Verifying linearizability: A comparative survey

Brijesh Dongol and John Derrick
University of Sheffield, UK

Abstract

Linearizability has become the key correctness criterion for concurrent data structures, ensuring that histories of the concurrent object under consideration are consistent, where consistency is judged with respect to a sequential history of a corresponding abstract data structure. Linearizability allows any order of concurrent (i.e., overlapping) calls to operations to be picked, but requires the real-time order of non-overlapping to be preserved. A history of overlapping operation calls is linearizable if at least one of the possible order of operations forms a valid sequential history (i.e., corresponds to a valid sequential execution of the data structure), and a concurrent data structure is linearizable iff every history of the data structure is linearizable. Over the years numerous techniques for verifying linearizability have been developed, using a variety of formal foundations such as refinement, shape analysis, reduction, etc. However, as the underlying framework, nomenclature and terminology for each method differs, it has become difficult for practitioners to judge the differences between each approach, and hence, judge the methodology most appropriate for the data structure at hand. We compare the major of methods used to verify linearizability, describe the main contribution of each method, and compare their advantages and limitations.

1 Introduction

Highly optimised fine-grained concurrent algorithms are increasingly being used in 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 and formal verification has uncovered subtle bugs in published algorithms that were previously thought correct [24, 14]. The main correctness criterion for concurrent algorithms is linearizability, which defines consistency conditions on the history of invocation and response events generated by an execution of the algorithm at hand [60, 62]. Assuming that operations start and end with invocation and response events, respectively, linearizability requires every operation call to take effect at some point between its invocation and response events. Concurrent calls may take effect in any order, but the order of sequential operation calls must be maintained. A (concurrent) history is linearizable iff there is some order for the effects of the operation calls that corresponds to a valid sequential history. A concurrent data structure is linearizable iff each of its histories is linearizable.
Over the years, an immense amount of research effort has been devoted to development of methods for verifying linearizability. Each of these has contributed to improving our understanding of linearizability and the conditions under which algorithms are linearizable. However, development of scalable approaches for verifying concurrent algorithms remains an open challenge, and as demand for fine-grained high performance concurrent algorithms has increased, the rate at which new methods for linearizability are being developed has accelerated. Unfortunately, each new method is presented using specialised formal frameworks, making it difficult to judge the merits of each approach. In this paper, we therefore present a comparative survey of the major techniques that have been developed thus far to examine the advantages and downfalls of each. Some questions to be asked about the different methods are:

- **Compositionality of the proof method.** How is linearizability verification localised?
- **Contribution of the framework.** Does the underlying framework contribute to simpler proofs?
- **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.** Can every linearizable algorithm be verified using the given 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 linearizability verification will be left out. The 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’s book [60] is already an excellent resource. Our hope is that the survey will enable one to better understand the fundamental challenges of linearizability verification and avenues of future work.

Linearizability is often verified by identifying a *linearization point* for each operation, which is an atomic statement whose execution causes the *effect* of the operation to be felt. After identification, one must then verify that execution of the linearization point has the same behaviour as the corresponding abstract operation. However, it turns out that identification of linearization points is non-trivial, and the difficulty of verifying linearizability of an algorithm depends on the location of the linearization points in each operation of the algorithm. Some algorithms such as the Treiber stack [98] have simple fixed linearization points. Others such as the Heller *et al* set [56] have linearization points that are determined by the execution of operations. Yet more complex algorithms include the Herlihy-Wing queue [62], where each linearization point relates to multiple possible abstract data types. Part of this survey aims to demonstrate properties of the algorithm at hand that causes difficulties when verifying linearizability. The desire to implement clever optimisations (to improve efficiency) leads to more complex behaviours of the underlying programs, which consequently increases the difficulty of formal verification.

We consider two case studies for comparison — an *optimistic set*, which consists of optimistic add and remove operations that insert and remove elements from the set, respectively, and a *lazy set* [56], which is the optimistic set together with a wait-free contains operation. This choice is based on the fact that the add and remove operations are lock-based operations with simple
push(v)
H1: n := new(Node);
H2: n.val := v;
H3: repeat
H4: ss := Head;
H5: n.next := ss;
H6: until CAS(Head,ss,n)
H7: return

pop: lv
P1: repeat
P2: ss := Head;
P3: if ss = null then
P4: return empty
P5: ssn := ss.next;
P6: lv := ssn.val
P7: until CAS(Head,ss,ssn);
P8: return lv

Figure 1: The Treiber stack

fixed linearization points. On the other hand, verifying linearizability of contains is complex, and requires consideration of other operations [15, 21].

2 Linearizability

Linearizability defines consistency conditions for data structures that allow multiple processes to concurrently execute operations, i.e., data structures for which the intervals of execution of operation calls may overlap. Informally speaking, sequential operations calls that (do not overlap) cannot be re-ordered, however, concurrent calls (that do overlap) may be linearized in any order.

2.1 Example: The Treiber stack

To make our forthcoming discussion more concrete, we introduce a simple stack example due to Treiber, which has become a standard case study from the literature. Initially the stack is described as a sequence of elements together with two operations push and pop. Operation push pushes its input \( v \neq \text{empty} \) on the stack, and pop removes one element that is returned in \( lv \). When the stack is empty, pop leaves the stack unchanged and returns the special value \( \text{empty} \).

The implementation uses a linked list of nodes which can be concurrently accessed by a number of processes. Nodes are cells in memory consisting of a value and a pointer to the next node. The pointer can also take the value \( \text{null} \). A variable Head is used to keep track of the current head of the list. Operations push and pop are split into several smaller operations: creating new nodes, swapping pointers etc. There is one operation atomically carrying out a comparison of values and an atomic assignment: \( \text{CAS}(\text{mem}, \text{exp}, \text{new}) \) (compare-and-swap), which compares mem to exp; if this succeeds (i.e. mem equals exp), mem is set to new and CAS returns true, otherwise the CAS fails, leaves mem unchanged and returns false. In pseudo-code the push and pop operations that one process executes are given in Figure 1.

The push operation first creates a new node with the value to be pushed onto the stack. It then repeatedly sets a local variable ss to Head and the pointer of the new node to ss. This ends once the final CAS detects that Head (still) equals ss upon which Head is set to the new node n. Note that the CAS in push does not necessarily succeed: in case of a concurrent pop, Head might have been changed in between. The pop is similar: it records the value of Head in ss, then determines
empty (i.e., stack is empty), the linearization point is the statement non-empty stack as at this statement the element is actually removed from the list. If the list is empty (i.e., stack is empty), the linearization point is the statement $P2$, when $Head = \text{null}$ is read (note that it is not line $P3$, which checks the local variable $ss$ only to be null. The global variable $Head$ might be non-null again at this point). Linearizability ensures every concurrent execution of the implementation data type is equivalent to some sequential execution of the abstract data type, where the sequential ordering is determined by the order of linearization points in the concurrent execution.

Processes executing these operations now have a shared global state (in this case $Head$) as well as local variables (e.g., $ss$). The criterion of linearizability ensures that despite concurrent access to variables, every operation seems to take effect at one point in between its invocation and return, the linearization point, e.g., the CAS in the $pop$ operation is a linearization point for $pop$ on a non-empty stack as at this statement the element is actually removed from the list. If the list is empty (i.e., stack is empty), the linearization point is the statement $P2$, when $Head = \text{null}$ is read (note that it is not line $P3$, which checks the local variable $ss$ only to be null. The global variable $Head$ might be non-null again at this point). Linearizability ensures every concurrent execution of the implementation data type is equivalent to some sequential execution of the abstract data type, where the sequential ordering is determined by the order of linearization points in the concurrent execution.

Notice, also, that this example illustrates the fact that an individual operation might have more than one linearization point depending on the actual values of its data structures. In $pop$ the linearization point depends on whether the stack is empty or not.

## 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 input and output events, each of which records invocations and response events of operation calls on a data structure. Each 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 [20]:

\[
\text{Event} ::= \text{inv} \langle P \times O \times V \rangle \mid \text{ret} \langle P \times O \times V \rangle
\]

For brevity, we use notation $op^I_p(x)$ and $op^R_p: r$ for events $\text{inv}(p, op, x)$ and $\text{ret}(p, op, r)$, respectively, and use $op^I_p$ and $op^R_p$ to respectively denote invoke and return events with no inputs or outputs. For an event $e = \langle p, op, x \rangle$, we assume projection functions $\text{proc}(e) = p$, $\text{oper}(e) = op$ and $\text{par}(e) = x$ that return the process, operation and parameter of event $e$, respectively. The definition of linearizability is formalised in terms of the history of events, which is represented formally by a sequence. Let $\text{seq}(X)$ denote sequences of type $X$, which we assume are indexed from 0 onward. Thus, $\langle a, b, c \rangle$ denotes the sequence $\{0 \mapsto a, 1 \mapsto b, 2 \mapsto c\}$ and $\langle \rangle$ denotes the empty sequence. A history is an element of $\text{History} \equiv \text{seq}(\text{Event})$, i.e., is a sequence of events. We assume sequences are indexed from 0 onward.

To motivate a history-based notion of linearizability, consider the following history of a concurrent stack, where execution starts with an empty stack.

\[
h_1 \equiv \langle \text{push}^I_1(a), \text{push}^I_2(b), \text{push}^R_1, \text{push}^R_2 \rangle
\]

processes 1 and 2 concurrently execute push operations, and hence, a linearization of this history is free to order these calls in either order, i.e., both linearizations below are valid.

\[
hs_1 \equiv \langle \text{push}^I_1(a), \text{push}^I_1, \text{push}^I_2(b), \text{push}^I_2 \rangle \quad hs_2 \equiv \langle \text{push}^I_2(b), \text{push}^I_2, \text{push}^I_1(a), \text{push}^I_1 \rangle
\]
The abstract stack at the ends of \( hs_1 \) and \( hs_2 \) will be \( \langle b, a \rangle \) (with \( b \) at the top) and \( \langle a, b \rangle \), respectively. Now suppose, history \( h_1 \) is extended with a sequential \( pop \) operation:

\[ h_2 \equiv h_1 \circ \langle pop^R_3, pop^R_3 : b \rangle \]

Note that a linearization of \( h_2 \) may not swap the order of the \( pop \) by process 3 with either of the \( push \) operations in \( h_1 \) because \( pop^I_3 \) occurs after the return of both \( push \)es, i.e., their executions are not concurrent. Because elements must be inserted and removed from a stack in a LIFO order, addition of the \( pop \) that returns \( b \) in \( h_2 \) restricts the valid linearizations. In particular, the only possible choice is one in which the push of \( b \) occurs after a push of \( a \), i.e.,

\[ hs_3 \equiv hs_2 \circ \langle pop^I_3, pop^R_3 : b \rangle \]

which results in an abstract stack \( \langle a \rangle \) at the end of execution. Note that \( hs_1 \circ \langle pop^I_3, pop^R_3 : b \rangle \) is an invalid linearization of \( h_2 \). Now suppose \( h_2 \) is appended with two more \( pop \) operations as follows:

\[ h_3 \equiv h_2 \circ \langle pop^I_3, pop^I_3, pop^R_3 : a, pop^R_3 : a \rangle \]

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 that are successfully able to remove \( a \) from the stack. A concurrent stack that generates \( h_3 \) would therefore be deemed incorrect.

A concurrent data structure is deemed linearizable iff every history of the data structure is linearizable. Thus, proving linearizability of the Treiber stack ensures that a history such as \( h_3 \) is never generated by the algorithm.

This concept is formalised as follows. For \( H \in History \), let \( H \mid p \) denote the subsequence of \( H \) consisting of all invocations and responses of process \( p \). Two histories \( H_1, H_2 \) are equivalent if for all processes \( p, H_1 \mid p = H_2 \mid p \). An invocation \( opi_p(x) \) matches a response \( opj^R_q : y \) iff \( opi = opj \) and \( p = q \). An invocation is pending in a history \( H \) iff there is no matching response to the invocation in \( H \) and is completed otherwise. We let \( complete(h) \) denote the maximal subsequence of history \( H \) consisting of all invocations and 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 \( opi <_H opj \) iff the response event of \( opi \) occurs before the invocation event of \( opj \) in \( H \), i.e., \( opi \) and \( opj \) do not execute concurrently and \( opi \) occurs before \( opj \). 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.

**Definition 1** (Linearizability [62]). A history \( HC \) is linearizable 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 some sequential history \( HS \) and \( <_{HC} \subseteq <_{HS} \).

The definition of linearizability allows histories to be extended with matching responses to pending invocations. This is necessary because some operations may be past their linearization point, but not yet responded. For example consider the following history, where the stack is initially empty.

\[ \langle push^I_p(x), pop^I_p, pop^R_q(x) \rangle \]  

(1)
The effect of operation $\text{push}_p^I(x)$ has clearly occurred in [1] because the $\text{pop}_q$ returns $x$, i.e., the linearization point of $\text{push}_p^I(x)$ occurs before that of $\text{pop}_q^I$, i.e., (1) is linearizable because it can be extended with a matching response to $\text{push}_p^I(x)$, and linearized by $\langle \text{push}_p^I(x), \text{push}_p^R, \text{pop}_q^I, \text{pop}_q^R(x) \rangle$.

2.3 Using an explicit matching function

It is clearly infeasible to work directly with the definition of linearizability, therefore, as we will see in (Section 3), numerous methods that imply Definition 1 have been developed. The approach of Derrick et al. goes a step further; and reformulates linearizability to enable it to be explicitly proved as part of the refinement [20, 21, 90]. In this section, we present Derrick et al.’s definition of linearizability for comparison with the original definition by Herlihy and Wing.

For a function $f$, we let $\text{dom}(f)$ denote the domain of $f$. Furthermore, assume that $\text{isInv}(e)$ holds iff $e \in \text{Event}$ is an invoke; and $\text{Ret}$ denotes the set of all return events. Two indices $m$ and $n$ form a matching pair in a history $h$ iff $h(m)$ and $h(n)$ belong to the same operation call and there are no other invocations by $\text{proc}(h(m))$ between $m$ and $n$. An index $m$ is a pending invocation in $h$ iff $h(m)$ is an invocation and there are no other events by process $\text{proc}(h(m))$ after index $m$. A history $h$ is legal iff for each index $n$, if $h(n)$ is an invoke event then either $h(n)$ is pending or $n$ forms a matching pair with an index $m > n$, otherwise (i.e., if $h(n)$ is a return event), there exists an index $m < n$ that forms a matching pair with $n$. These concepts are formalised by the following definition.

**Definition 2 (Legal history).** For $h : \text{seq(Event)}$, two positions $m, n \in \text{dom}(h)$ form a matching pair, denoted $\text{mp}(m, n, h)$ iff

\[
\begin{align*}
& \forall k \cdot m < k < n \Rightarrow \text{proc}(h(k)) \neq \text{proc}(h(m)) \\
& \forall k \cdot \text{isInv}(h(m)) \land \forall n : \text{dom}(h) \cdot m < n \Rightarrow \text{proc}(h(m)) \neq \text{proc}(h(n))
\end{align*}
\]

A position $n$ : $\text{dom}(h)$ in $h$ is a pending invocation, denoted $\text{pi}(n, h)$, iff

\[
\text{isInv}(h(m)) \land \forall n : \text{dom}(h) \cdot m < n \Rightarrow \text{proc}(h(m)) \neq \text{proc}(h(n))
\]

We say $h$ is legal, denoted $\text{legal}(h)$, iff

\[
\forall n : \text{dom}(h) \cdot \text{if isInv}(h(n)) \text{ then } \text{pi}(n, h) \lor \exists m : \text{dom}(h) \cdot \text{mp}(n, m, h) \\
\text{else } \exists m : \text{dom}(h) \cdot \text{mp}(m, n, h)
\]

Linearizability of a concurrent history $h$ is given by comparing it with the abstract sequential histories. First of all, $h$ might need to be extended by some returns $h_0 \in \text{seq(Ret)}$ that match the pending invokes. This is the case when $h$ contains operations where the effect has already taken place, though they have not returned. An example for this is the stack history [1], where the effect of the $\text{push}$ has already occurred (we are after the linearization point) but the return has not yet occurred. The obtained thus history $h \bowtie h_0$ is now compared to a sequential history $hs$ according to the two conditions below, where the function $\text{complete}(h)$ removes all pending invocations from $h$.

$L1.$ when projected onto processes, $\text{complete}(h \bowtie h_0)$ and $hs$ have to be equivalent, and

$L2.$ the ordering of operation executions in $h$ needs to be preserved in $hs.$
Definition 3. Given two histories \( h, hs \), we define \( h \) to be in lin-relation with \( hs \), denoted \( \text{lin}(h, hs) \), iff
\[
\exists f: \text{dom}(h) \mapsto \text{dom}(hs) \land \forall n: \text{dom}(h) \land h(n) = hs(f(n))
\land \forall m, n: \text{dom}(h) \land m < n \land mp(m, n, h) \Rightarrow f(n) = f(m) + 1
\land \forall m, n, m', n': \text{dom}(h) \land n < m' \land mp(m, n, h) \land mp(m', n', h)
\Rightarrow f(n) < f(m')
\]

Definition 4 (Linearizable histories [20]). A concurrent history \( h \) is linearizable with respect to some sequential history \( hs \) iff there exists some \( h_0 \in \text{seq}(\text{Ret}) \) such that \( \text{linearizable}(h, h_0, hs) \equiv \text{legal}(h \cap h_0) \land \text{lin}(\text{complete}(h \cap h_0), hs) \) holds. A concrete implementation \( C \) is linearizable with a sequential specification \( A \) if for all histories \( h \) of \( C \) there is a sequential history \( hs \) of \( A \) and a history \( h_0 \in \text{seq}(\text{Ret}) \) such that \( \text{linearizable}(h, h_0, hs) \).

For example, histories \( h_1, h_2 \) and \( h_3 \) are all legal. In addition, \( h_1 \) is linearizable with respect to \( hs_1 \) using mapping \( f_1 \equiv \{0 \mapsto 0, 1 \mapsto 2, 2 \mapsto 1, 3 \mapsto 3\} \), \( h_1 \) is linearizable with respect to \( hs_2 \) using mapping \( f_2 \equiv \{0 \mapsto 2, 1 \mapsto 0, 2 \mapsto 3, 3 \mapsto 1\} \), \( h_2 \) is linearizable with respect to \( hs_3 \) using mapping \( f_2 \cup \{4 \mapsto 4, 5 \mapsto 5\} \). In contrast, there is no mapping from \( h_3 \) to a valid sequential history.

2.4 Difficulties in verifying linearizability via linearization points

When verifying linearizability, one must first identify the linearization points in the implementation code and prove correspondence between execution of the linearization points and the execution of a sequential specification. Typically, this correspondence relation is refinement [18], and using refinement terminology, we refer to the implementation as the concrete system and the sequential specification as the abstract system. To verify refinement, one must identify and establish appropriate invariants to prove the correspondence relation. The difficulties in verifying an algorithm is mostly dictated by the complexity of the correspondence relation, which in turn is dictated by the type of linearization point of the algorithm.

A summary of algorithms and properties of their linearization points is given in Table 1. Of course, there exist several other algorithms than have not yet been formally verified correct, and hence, the list of algorithms in Table 1 is only partial. In particular, many complicated algorithms have not yet been formally verified, and there is a possibility that yet more complex classes of algorithms can be identified (e.g., [97]).

An operation may have several linearization points, and some operations (e.g., contains operation of the lazy set) may even be linearized multiple times over its interval of execution. This is allowed as long as at least one of these is correctly chosen. Schellhorn et al. have shown that by adding histories of invocations and responses to concrete histories, using backwards reasoning (e.g., backward simulation, or prophecy variables) is complete [90], i.e., provided an algorithm is

---

1 Original version contained an error, which was corrected and verified by [14].
2 Also known as the lock-coupling [9] set.
3 Here, this refers to the lazy set without the contains operation (Section 4). Herlihy and Shavit [60] give an optimistic set with an optimistic contains operation.
4 Algorithm is based on [53].
Linearization points | Example algorithms | Appears in
--- | --- | ---
Fixed | Treiber stack | [98]
MS queue | [75]
Array-based queue | [91]
Pessimistic set | [60]
DCA S set | [105]
Optimistic set | [60]
External | Lazy set | [56]
Elimination stack | [57]
Future behaviour | HW queue | [62]
RDCSS (Restricted Double Compare Single Swap) | [54]
MCAS (Multi-word Compare and Swap) | [54]
CCAS (Conditional Compare and Swap) | [39]
Elimination queue | [77]
Snark-double ended queue | [25]
HM lock-free set | [76]

Table 1: Classification of algorithms

linearizable, there exists a backward simulation relation that can be used to verify linearizability. In fact, it turns out that one can construct a “maximal” backward simulation relation so that each operation is linearized at its invocation. However, such maximal relations are highly non-deterministic and its general verification is often infeasible. For many algorithms, simpler solutions have been found that make it possible to verify refinement without constructing the maximal simulation relation.

Algorithms such as the Treiber stack have fixed (or static) linearization points, whose execution only linearizes the operation to which the linearization point belongs. 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 \( \text{Head} = \text{null} \) holds when P2 is executed — at this point, if \( \text{Head} = \text{null} \) 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, without having to refer to the potential future behaviour. 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 many cases reasoning can even be automated [102].

Unfortunately, not all algorithms can be verified in this manner. For example, some algorithms consist of external linearization points, where an operation may potentially be linearized by the execution of other operations (by other processes). Here, the operation executing in isolation must set its own linearization points, but interference from other processes may cause the operation to be linearized externally. We present a detailed comparison of different approaches to verifying one such example (the lazy set) in Section 5, whose verification requires use of advanced methods such as backward simulation and potential linearization points.

A third, yet more complicated class of algorithms are those whose linearization points depend on the future behaviour of the algorithm, and hence each state of concrete system potentially corre-
Verifying linearizability

Verifying linearizability by considering Definition 1 directly is infeasible because the concurrent data structures of interest generate infinitely many histories. Furthermore, the formal definition is at a level of abstraction that does not consider the internal specifics of any implementation. In practice, one is interested in generating a finite number of localised proof obligations of the implementation at hand that guarantees every history extracted from every possible trace is linearizable.

Following [13], a concrete system may be modelled using a finite unbounded set of processes that executes each transition by non-deterministically picking a process \( p \) from the set of processes and executing \( p \)'s next atomic statement. A transition may invoke or return an operation call, or execute the next step of an operation. Each transition is often non-blocking with fine-grained atomicity, and hence, the traces of the system generated by the concurrent execution contain a high degree of interleaving. Figure 2 shows an interleaved trace of the Treiber stack from Section 2.1, where \( L_p \) denotes the atomic statement labelled \( L \) executed by process \( p \). The concurrent history corresponding to the trace only considers the relative ordering of invokes and responses, and for the example in Figure 2 allows one to infer the concurrent execution of \( \text{Push}(a) \), \( \text{Push}(b) \) and \( \text{Pop}: b \) (a pop that returns \( b \)). The internal transitions of each process (e.g., \( H_{1p}, H_{2p}, H_{1q} \)) cannot be recovered from the history. Assuming that execution starts with an empty stack, the history in Figure 2 is clearly linearizable: due to the concurrent execution of \( \text{Push}(a) \) and \( \text{Push}(b) \), a pop operation is free to observe either \( a \) or \( b \) (or even \( \text{empty} \)) at the top of the stack. In the example, we assume the \( \text{Pop} \) observes \( b \) at the top of the stack.

The question now is: Can we infer a sequential history \( hs(tr) \) from each trace \( tr \) of the concurrent data structure, such that the concurrent history \( hc(tr) \) corresponding to \( tr \) is linearized by \( hs(tr) \)? For a solution using linearization points, the ordering of the linearization points in the interleaved trace precisely determines the sequential order of the operations. For the example trace in Figure 2, suppose the \( \text{push} \) operations are linearized by the statement labelled \( H6 \), and \( \text{pop} \)
operations linearized by statements labelled \( P7 \). This results in linearization points marked by the crosses and a sequential history in which the ordering of operations is \( \langle \text{Push}(a), \text{Push}(b), \text{Pop}:b \rangle \), allowing one to infer that the choice of linearization points is in fact correct.

As discussed in Section 2.4, a concurrent algorithm can be categorised according to the location of the linearization points in the algorithm. Table 2 presents an summary of methods for verifying linearizability, together with the algorithms that have been verified, and people involved with each method. We describe these methods in more detail in the sections below.

### 3.1 Refinement-based verification

For any concrete trace \( tr \), directly reasoning about whether the concurrent history corresponding to \( tr \) (i.e., generated from the real-time order of the invoke and return events of \( tr \)) is linearized by the

---

5This is the only method known to have found two bugs in existing algorithms [24, 14].

6Including a variation in [26].

7Set algorithm in [76], which is based on [53].

8Rely-guarantee Interval Temporal Logic

9The use of atomicity brackets prohibits behaviours that are permitted by the full fine-grained.

---

### Table 2: Methods for verifying linearizability

| Method                                      | Algorithms verified                                      | Reference |
|---------------------------------------------|---------------------------------------------------------|-----------|
| Simulation with canonical abstraction       | Treiber stack, MS queue, Array-based queue, Lazy set    | 48        |
|                                             | Elimination stack, Snark double-ended queue             | 15        |
| Data refinement with sequential abstraction | Treiber stack, Pessimistic set, Lazy set                | 20        |
|                                             | HW queue                                                | 21        |
| Rely-guarantee with separation logic        | RDCSS, Pessimistic set, Optimistic set, Lazy set        | 100 [67]  |
|                                             | MCAS                                                    | 100 [67]  |
|                                             | CCAS, Elimination stack, Two-lock queue, MS queue\(^6\), HM lock-free set\(^7\) |           |
| Reduction                                   | Treiber stack, MS queue, Elimination stack, Simplified multiset | 36        |
| RGITL\(^8\)                                 | Treiber stack, MS queue\(^6\)                           | 8         |
|                                             | Treiber stack with hazard pointers                      | 96        |
| Shape analysis                              | Treiber stack, MS queue\(^6\)                           | 6         |
|                                             | Numerous algorithms from [60]                           | 102       |
| Construction-based                          | Treiber stack, MS queue, Elimination stack, Optimistic set | 65        |
|                                             |                                                        | 8 and 50  |
| Hindsight lemma                             | Optimistic set, Lazy set\(^9\)                          | 82, 83    |
sequential history corresponding to \( tr \) (i.e., generated from the real-time order of the linearization points of \( tr \)) is infeasible. One therefore appeals to known techniques to infer the properties of the traces from the statements of the algorithm at a higher level of abstraction. In particular, the behaviours of the concrete (concurrent) data structure implementation in question are compared with the behaviours of its abstract (sequential) counterpart. Formally, this relationship is given by data refinement, which holds between an abstract and concrete system iff every observable behaviour of the concrete implementation is a possible observable behaviour of the abstract \[13\]. Typically, the internal representation of data within concrete and abstract algorithms differ, requiring one to use a representation relation to link the two state spaces. For example, in the Treiber stack (Section 2.1), the abstract representation is a sequence of values, whereas the concrete implementation is a linked list.

Compositional proofs of data refinement are achieved via simulation, which allows one to reason about each transition of the concrete data structure individually (see Figure 3). The first formal proofs of linearizability are simulation-based and given by Groves et al. \[15, 14, 13, 26, 24\], using the framework of Input/Output Automata \[72, 73\]. Over the years, several improvements and variations of this work have been developed, but the core idea for refinement-based linearizability verification remains the same; behaviours of the concrete algorithm, which executes with fine-grained atomicity are compared to behaviours of a coarse-grained abstract algorithm. Due to these differences in granularity, most concrete transitions correspond to stuttering transitions with respect to the abstract algorithm, and non-stuttering transitions of the concrete typically correspond to the linearization point of some operation.

Figure 3 shows four typical simulation diagrams where AInit, AOp and AFin are abstract initialisation, operation and finalisation steps (and similarly CInit, COp and CFin), \( \sigma, \sigma' \) are abstract states, \( \tau, \tau' \) are concrete states, and \( rep \) is a representation relation between abstract and concrete states. The proof obligations require that the statements of the concrete algorithm be characterised as initialisation, non-stuttering, stuttering, and finalisation transitions. 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. Neither forwards nor backwards simulation alone is complete for verifying data refinement, but the combination of the two is known to be complete \[18\].

The abstract data structures in Groves et al.’s proofs \[15, 14, 13, 26, 24\] are canonical constructions as described by Lynch \[73\], where each operation call consists of an invocation, a single atomic transition to perform the operation (and possibly modify the shared data structure), and a return transition. For the Treiber Stack, the canonical push operation would be of the form PushInv; DoPush; PushRes, where each PushInv, DoPush and PushRes is an atomic state-
DoPush is the main transition, which performs the operation at the abstract level. Other concurrent operations may be interleaved between an invocation and the main transition of an operation, as well as between the main transition and the return transition, but the main action of each operation is performed atomically. The histories that canonical automata generate are concurrent, however, due to the atomicity of the main action, every history corresponding to a canonical program is linearizable [73]. This guarantees linearizability of the implementation because every trace of an implementation is also a trace of the canonical construction. The methods in [15, 14, 13, 26, 24] all rely on this fundamental result [24].

To see how simulation with canonical automata are used to verify linearizability, consider once again the trace from Figure 3. Using Groves et al.’s methods, recalling that we assume statements labelled $H_6$ and $P_7$ linearize the push and pop statements respectively, after establishing simulation between the concrete implementation and canonical specification, the concrete trace from Figure 3 is mapped to a canonical trace as shown in Figure 4. Each invocation and response of the concrete maps to the corresponding invocation and response of the abstract, and each non-stuttering statement maps to the corresponding main statement. For example, executing $H_6_p$ corresponds to the an abstract main operation $APush_p$ that performs the push on the abstract data structure atomically. As can be seen in Figure 4, the canonical history is still concurrent but the linearization points are trivially identifiable as those that execute the main operation at the abstract level.

Although Groves et al. use Lynch’s results to link linearizability between concurrent fine-grained and canonical (coarse-grained) programs, linearizability is ultimately about a relationship between a concurrent implementation and sequential abstraction. Therefore, a fundamental question about the link between linearizability and data refinement between concurrent and sequential programs remains. That is: Can linearizability be formulated as an instance of data refinement between a concurrent implementation and a sequential abstract program? To answer this, Derrick et al. [90, 21, 19, 20] develop a method using history-enhanced data types, where the (sequential) abstract and (concurrent) concrete data types are extended with an auxiliary history variables that the record invocations and responses of each operation. Like Groves et al., simulation-based methods are used to verify the data refinement. Additionally thread-local proof obligations that dispense with histories are generated whose satisfaction implies linearizability.

For a concrete example, once again consider the stack trace from Figure 3 but with history enhanced data types. Using the methods of Derrick et al. [20], one would obtain a refinement shown in Figure 5, 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 \(hc \uparrow \langle push_p^I(a) \rangle\). Every abstract transition updates the \(ha\) with matching invocation and response pairs, e.g., \(APush_p\) updates the \(ha\) to \(ha \uparrow \langle push_p^I(a), push_p^R \rangle\). 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 (which is achieved using an explicit matching function from Section 2.3), as opposed to relying on a canonical formulation to generate linearizable traces.

3.2 Augmented states: Separation logic, shape analysis and automation

Another method for verifying linearizability is via the introduction of auxiliary variables to the concrete algorithm that represents the abstract data structure being implemented [100, 103, 68, 67]. For example, in the Treiber stack, one would introduce an abstract sequence, say \(Stack\), to the program in Section 1. At each linearization point of the Treiber stack, a corresponding operation is performed on \(Stack\), e.g., the CAS at \(H6\) is augmented so that \(Stack\) is updated to \(\langle v \rangle \uparrow Stack\) whenever the CAS is successful [100]. The linearizability proof involves proving an invariance of an abstraction relation between the concrete and abstract representations. For the Treiber stack, the concrete implementation is a linked list, whereas the abstract is \(Stack\), which is a sequence of values.

Unlike the refinement-based approaches (Section 3.1), which require use of a separate refinement relation to link the abstract and concrete states, extending the concrete state with the abstract state allows one to express the refinement relation as an invariant over the augmented space. This has the advantage of flattening the state space of two different levels of abstraction into a single layer, and the ability to use separation logic [81] and rely-guarantee [63] over the unified layer. In particular, Vafeiadis et al.'s approach [100, 103] use a framework that combines rely-guarantee for compositionality and separation logic for pointer-based reasoning [104]. Liang and Feng [68, 67] develop a related framework, which is used to prove a number of algorithms from the literature.

Applying this technique to the example trace from Section 2, we obtain the augmented trace in Figure 6. Note that the augmented state can be partitioned into disjoint substates corresponding to the concrete and abstract states. The abstract state is updated according to the corresponding abstract operation each linearization point.

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
observation is exploited in the forward simulation proof in \[15\], 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. This reduction in expressive power becomes important for proofs of operations with external and non-fixed linearization points, which is a common characteristic of many interesting algorithms, e.g., the proof of an operation with non-fixed linearization is left informal in \[103\], despite the fact that the framework allows (two-state) rely predicates \[63\].

Vafeiadis’ thesis \[100\] addresses these shortcomings via the use of more sophisticated auxiliary statements that are able to linearize both the currently executing operation as well as other executing processes. In addition, prophecy variables \[1\] are used to reason about operations whose linearization points depend on future behaviour. Recently, Liang and Feng \[68\] have consolidated these ideas into a framework based on separation logic. Their augmentations include linself, (which performs the same function as the augmentations of Vafeiadis by linearizing the currently executing process \[103\]) and lin(p), (which performs the linearization of process \(p\) different from self that may be executing a different operation \[100\]). Unlike Vafeiadis, they also allow augmentations that use try and commit pairs, where the try is used to guess potential linearization points, and the commit used to pick a from the linearization points that have been guessed thus far.

Augmented state spaces also form the basis for shape analysis \[64\], which is a static analysis technique for verifying properties of data structures with dynamically allocated memory. One of the first shape-analysis-based linearizability proofs is that of Amit et al. \[6\], who consider implementations using singly linked lists and fixed linearization points. The following paraphrases Amit et al. \[6, pg 480\], 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., the 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 \[101\], combining Section 3.2 with shape analysis. Unlike Amit et al., a distinction is made between shape abstraction (describing the structure of data structure) and value abstraction (describing the values contained within the data structure). The method is used to verify several algorithms, including the
Although the behaviours of concurrent data structures are complex, many algorithms that implement them are 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 presents a fully automated method that considers all linearization points in a single transition [102], and infers the required abstraction maps based on the given program and abstract specification of the data structure. The method thus far is only able to handle so-called *logically pure* algorithms, i.e., those that do not logically modify the corresponding abstract state. Due to the focus on automation, the possibility of a proof for logically impure algorithms is only conjectured, provided the verifier supplied the required abstraction maps.

### 3.3 Compositional methods

Compositional frameworks allow one to reason about a concurrent program in a modular manner by separating the behaviour of a single component from that of its environment [17], where the behaviour of the environment is formalised abstractly by a two-state relation. For shared-variable concurrency, a popular approach to compositionality is Jones’ rely-guarantee framework [63], 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 overview of compositional verification techniques lies outside the scope of this paper; we refer the interested reader to [17, 100].

As already mentioned in Section 3.2, rely-guarantee reasoning has been applied to linearizability verification. In particular, rely conditions are used to encode the transitions that modify the global state, e.g., lock acquisition, pointer modification. Schellhorn *et al.* combine rely-guarantee reasoning with interval temporal logic [80], which enables one to reason over the interval of time in which a program executes, as opposed to single state transition [89]. The proofs are carried out using the KIV theorem prover [34], which is combined with symbolic execution [12, 7] to enable guarantee conditions to be checked by inductively stepping through the program statements within KIV, simplifying mechanised verification. Furthermore, using interval temporal logic enables one to reason about the future behaviour of an operation, which sometimes allows one to avoid backward simulation proofs [8]. These methods have been applied to verify the Treiber stack and the Michael/Scott queue [8]. Our own methods [30] also use rely-guarantee reasoning, but allow more general rely conditions that describe properties over an interval of time as opposed to over a pre and post state.

### 3.4 Proofs without linearization points

The verification methods in Sections 3.1–3.3 allow localised reasoning so that the transitions of only one (arbitrarily chosen) process needs to be considered at a time. However, each method relies on the correct identification of linearization points, which as seen in 2.4 is non-trivial and may depend on future behaviour. Therefore, expert understanding of the algorithm at hand is often necessary for them to be verified correct. Many researchers have examined methods that allow...
verification of linearizability without requiring the linearization points to be identified. Here, we present a survey of some of these techniques.

Reduction. This [69] defines conditions for constructing coarse-grained atomic blocks from finer-grained atomic statements in a manner that does not modify the global behaviour of the algorithm. After this transformation, the remaining proof only needs to focus on verifying linearizability of the coarse-grained abstraction [47, 45, 36], which is simpler than verifying the original program because fewer statements need to be considered. In essence, reduction enables one to ensure trace equivalence of the fine-grained implementation and its abstraction by verifying commutativity properties. For simple example, in a program $S_1; S_2$ if $S_2$ performs purely local modifications, $S_2_p ; T_q = T_q ; S_2_p$ will hold for any statement $T$ and processes $p, q$ such that $p \neq q$. Therefore, $S_1; S_2$ in the program code may be treated as atomic${\{S_1; S_2\}}$.

Wang and Stoller [107] present static analysis techniques together with reduction theorems to reduce the atomicity. Automation of their results is also conjectured. Reduction has been used by Groves [47, 45] and separately by Elmas et al. [36]. The methods of include a tool support [35] via their QED verifier.

Problem-specific techniques. Concurrency research has also focused on problem-specific methods for simplifying proofs. One such method for non-blocking algorithms is the Hindsight Lemma [82], which has been used to verify different linked-list-based implementations of the lazy set. The lemma applies to linked list implementations of concurrent sets (e.g., the lazy set) and characterises conditions under which one a node is guaranteed to have been in or out of a set. However, the original paper [82] only considers a simple optimistic set. The extended technical report [83] presents a proof of the Heller et al.’s lazy set. Unfortunately, they choose to model the locks within the add and remove operations using atomicity brackets, which has the unwanted side effect of disallowing concurrent reads of the locked nodes. That is, although O’Hearn et al. claim to have a proof of the lazy set, the modifications they have made to the algorithm (in particular the atomicity brackets), mean that the algorithm they have verified is in fact not the lazy set. Overall, the ideas behind problem specific simplifications such as the Hindsight Lemma are interesting, but the logic used and the data structures considered are highly specialised. Therefore, it is not clear whether such techniques can be applied to other data structures in the literature.

Some data structures like queues and stacks can be uniquely identified by their aspects (properties that ensure the data structure in question has been implemented). This is exploited by Henzinger et al., who show present an aspect-oriented proof of the Herlihy/Wing queue [59]. Automation has been achieved for algorithms with helping mechanisms and internal linearization points such as the elimination stack [33]. These techniques require the algorithms to satisfy so-called $R$-linearizability [86], a stronger condition than linearizability, hence, verification of algorithms with linearization points based on future behaviour are excluded.

Interval-based abstraction. Linearizability is a property over the intervals in which operations execute, requiring a linearization point at some point between the operation’s invocation and response. Some methods therefore aim to exploit this more directly using interval logics (based on ITL [80, 79]), to reason about linearizability [8, 89, 7, 50, 29]. Of these, our own methods [30, 29] verify behaviour refinement between a coarse-grained abstraction and fine-grained implementation. The basic motivation resembles the reduction-based approaches and hence, unlike [8, 89, 7],
aims to simplify linearizability proofs by allowing abstraction to be proved without having to identify linearization points in the concrete code (see Section 5.5).

### 3.5 Linearizability by construction

Several researchers have also proposed development of linearizable algorithms via incremental refinement, starting with an abstract specification. Due to the transitivity of refinement, and because the initial program is an atomic program that is guaranteed to be 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. Furthermore, this work provides insight into the possibility developing algorithms for data structures that more complex than those currently found in the literature (e.g., stacks, queues, hash tables).

The first constructive approach to linearizability is Abrial and Cansell [3], within the Event-B framework [2], and the proofs are mechanised using an Event-B tool. However, the final algorithm they obtain requires counters on the nodes (as opposed to pointers [75]), and it is not clear whether such a scheme really is implementable.

Groves [46] presents a derivation of the Michael/Scott queue using trace reduction to justify each refinement step [69]. In particular, atomicity decomposition is justified if the new actions commute with those in concurrent processes. This is extended by Groves and Colvin [50, 49], where the derivation of a more complicated stack by Hendler et al. [57] is derived. This stack uses an additional backoff array in the presence of high contention for the shared central stack. Groves and Colvin’s derivation allows 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) [50, 49]. These proofs are not yet mechanised, and the effort required to mechanically perform these reduction-based proofs is unclear.

Gao and Hesselink et al. present a number of derivations of non-blocking algorithms [41, 44, 43, 42], including the development of special-purpose reduction theorems [44]. However, these derivations aim to preserve lock-freedom (a progress property) [74], as opposed to linearizability.

Vechev and Yahav [105, 106] present tool-assisted derivation methods based using bounded model checking to obtain assurances that a derived algorithm is linearizable. Starting with a sequential linked-list set, they derive variations of the set algorithm implemented using DCAS 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 are checked; linearizability verification requires potentially infinite executions to be verified.

More recently, Jonsson [65] has presented a derivation of the Treiber stack and Michael/Scott queue in a refinement calculus framework [78]. Jonsson defines linearizability using: A program $P$ is linearizable if and only if atomic{$P$} is refined by $P$ [65, Definition 3.1]. With such an interpretation of linearizability, one is able to start by treating the entire concrete operation as a single atomic transition, then incrementally split its atomicity into finer-grained portions. Like Groves and Elmas et al., reduction-style commutativity checks are used to justify splitting
[103] Manual — No No  Operation contains not verified
[15] Manual PVS Yes No  Allows model checking
[100] Manual — Yes No  Auxiliary code can
[274] Linked to HW defn
Table 3: Overview of methods

the atomicity at each stage. This method does not consider development of alternatives to the algorithm at hand, and algorithms with non-fixed linearization points have not been considered.

More recently, Turon and Wand [99], have presented a constructive approach to development of fine-grained data structures, but dispense with linearizability as a proof obligation. Instead, they focus on maintenance of the observable behaviour of the abstract data structure.

4 Case study 1: An optimistic set algorithm

Set algorithms have become standard case studies for showing applicability of a particular theory to verifying linearizability. Of particular interest is the lazy set by Heller et al. [56], 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 with only an add and remove operation. An overview of the different approaches to verifying set algorithms is given in Table 3. Further details of each method are provided in the sections that follow. We present the approaches in historical order to highlight the contributions and advances of each method. The formalisation in this section aims to highlight the main ideas behind each method to elucidate their differences. We refer readers interested in reproducing each proof to the original papers.

10 Via completeness of forwards and backwards simulation for proving refinement.
11 Via completeness of auxiliary and prophecy variables for proving refinement.
12 The lazy set [56] is not yet verified in the method.
13 Uses Hindsight Lemma to generate proof obligations.
14 Linearizability proofs are performed at a higher level of abstraction.
15 Backward simulation for history-enhanced data types shown to be complete for linearizability [90].
16 Link to linearizability is via results of Filipović et al. [38].
17 Using history-enhanced data types [29].
4.1 An optimistic set

In this section, we present a simplified version of Heller et al.’s concurrent set algorithm [56] (see Figure 7) operating on a shared linked list, which is sorted in strictly ascending values order. The algorithm consists of operations add and remove, where both add and remove operations use an auxiliary operation locate to optimistically determine the position of the node to be inserted/deleted from the linked list, i.e., the 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 at L8. If the validation fails, the locks are released and the search restarted.

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 marked bit[18] 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 $\infty$ are used at the start (Head) and end (Tail) of the list. All values v inserted are assumed to satisfy $-\infty < v < \infty$.

Operation locate(x) is used to obtain pointers to two nodes whose values may be used to determine whether or not x is in the list — when locate(x) returns, the value of the predecessor node pred must be less than x, and the value of the current node 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, the add(x) operation 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 Figure 8(a), which represents the (abstract) set {3, 18, 77}, and an execution add(42) by process p without interference. Execution starts by calling operation locate(42), which searches for the predecessor (pred) and successor (curr) of the node to be added, ignoring any other locks (lines L3-L5). After exiting the loop, the executing process locks both pred and curr, which prevents their modification by other processes (lines L6-L7). At L8, the process checks to ensure that both pred and curr are unmarked (which ensures that they have not been removed since the end of the loop) and that pred.next is still curr (which ensures that no new nodes have been inserted between pred and curr since the end of the loop). If the test at L8 fails, the locks on pred and curr are released and the process locate is re-executed. In our example execution, we assume that the test is successful, which causes n1p and n2p to be set as shown in Figure 8(b). Having found and locked the correct location for the insertion, the process executing add tests to see that the value is not already in the set (line A2), then creates a new unmarked node n3p with value 42 and next pointer n3p (see Figure 8(c)).

[18] The mark bit is not strictly necessary to implement the optimistic set (e.g., [100]), however, we use it here to simplify the lead up to the lazy set in Section 5.

19
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
L9: then return pred,
L10: pred.unlock();
L11: curr.unlock()
else
L12: curr := pred.next
L13: if !curr.mark
L14: and pred.next =
L15: then return false;
L16: curr.unlock();
L17: pred.unlock();
L18: curr.unlock();
L19: return res

by executing A4, the executing process sets the next pointer of \( n_{1_p} \) to \( n_{2_p} \) causing a successful add operation (that returns \textit{true}) to be linearized (see Figure 8(d)) — provided no remove(42) operations are executed, any other add(42) operation that is started after A4 has been executed will return \textit{false}. Following the linearization, process \( p \) releases the locks on \( n_{1_p} \) and \( n_{3_p} \) and returns \textit{true} to indicate that the operation was successful.

Now consider the execution of remove(18) by process \( p \) on the set \{3, 18, 77\} depicted by the linked list in Figure 9(a), where the process executes without interference from the other processes. Like add, the remove operation calls locate(18), which returns the state depicted in Figure 9(b). At R2, a check is made that the element to be removed (given by node n2) is actually in the set. Then, the node n2 is removed logically by setting its marked value to \textit{true} (line R3), which is the linearization point of remove (see Figure 9(c)) — provided no add(18) operations are executed, any other concurrent remove(18) operation that starts after R3 has been executed will return \textit{false}. After execution of the linearization point, operation remove sets n3 to be the next pointer of the removed node (line R4), and then node n2 is physically removed by setting the next pointer of n1 to n3 (see Figure 9(d)). Then, the held locks are released and \textit{true} is returned to indicate that the remove operation was been successful. Note that although 18 has been logically removed from the set in Figure 9(c), no other process is able to insert 18 to the set until the marked node has also been physically removed as depicted in Figure 9(d), and the locks held by \( p \) have been released.
Figure 8: Execution of add(42) by process p

Figure 9: Execution of remove(18) by process p

4.2 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. Therefore, the refinement-based methods (Section 3.1) are able to verify correctness using the relatively straightforward simulation rules and the state augmentation methods (Section 3.2) are able to modify the abstract state directly. As we will see in Section 5, verification of an algorithm that includes contains considerably increases the complexity of the invariants and refinement relations.

Comparing how the different frameworks are used to verify these the simpler optimistic set is instructive, highlighting the similarities between the underlying methods and additionally, enables us to introduce the concepts behind each method using a simpler example. To unify the presentation, we translate the PVS formulae from [16] and the Vafeiadis’ RGSep notation [104, 100] into Z [11], 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.

Details on representing concurrent algorithms in Z are given in [20]. To reason about linked
lists, the memory model must be explicitly modelled, and hence, the concrete state \( CState \) is defined as follows. Below, \( Label \) and \( Node \) are assumed to be the types of a program counter label, and node, respectively. Each atomic statement is represented by a schema. For example, the schema for the statements in Figure 7 labelled \( A5 \) and \( A7 \) are given below. Each \( p \) below is a process (of type \( P \)) and \( pc(p) \) is an auxiliary variable for the program counter for process \( p \in P \).

\[
\begin{align*}
\text{CState} & \quad \text{Add}_5^p \quad \text{Add}_7^p \\
\text{pred, curr: } P \to Node & \quad \Delta CState \\
n1, n2, n3: P \to Node & \quad pc(p) = A5 \\
\text{pc: } P \to Label & \quad next'(n1(p)) = n2(p) \\
\text{lock: } Node \to \mathcal{P} P & \quad \Delta CState \\
\text{next: } Node \to Node & \quad pc(p) = A7 \\
\text{mark: } Node \to \mathbb{B} & \quad res'(p) = false \\
\text{res: } P \to V & \quad
\end{align*}
\]

Using the Object-Z [94] convention, we assume that variables \( v' = v \) for every variable \( v \) unless \( v' = k \) is explicitly defined for some new value \( k \).

Execution of the lazy set is modelled by assuming the existence of an unbounded but finite set of processes, each of which alternates between an idle state (i.e., not executing any operation) and an active state (i.e., executing some operation). A process in an idle state may become active by non-deterministically invoking one of the set operations with a non-deterministically chosen parameter. Upon completing the operation, an active process returns to idle.

A valid linearization point for a successful add operation is \( A5 \) and a successful remove is \( R3 \). For an unsuccessful add(\( x \)) operation (which occurs if the element being added is already in the set), there are two possible choices for the linearization point; either line \( A2 \) if the test fails, i.e., \( n3.val = x \) holds, or line \( A7 \), which sets the result to false. The simpler choice here is \( A7 \), which unlike \( A2 \), is an unconditional linearization point. Conditional linearization points are not unique to the lazy set (c.f., the empty dequeue case for Treiber’s stack and Michael/Scott’s queue), nor is the ability to pick from one of several possible linearization points. In fact, suppose the add operation was modified so that atomic local statements were introduced before and after \( A7 \) as follows

\[
\text{addA}(x):
\]

\[
\ldots A7a: \ local\_stmt\_a; \ A7: \ res := false; \ A7b: \ local\_stmt\_b; \\
\ldots
\]

Then, either \( A7a, A7, \) or \( A7b \) could be selected as the linearization point for the addA operation — due to the locks held by the process executing add(\( x \)), the linked list is guaranteed to contain \( x \) when \( A7a, A7, \) and \( A7b \) are executed. A similar argument applies to the remove operation; below we assume that \( R7 \) has been chosen to be the linearization point.

### 4.3 Method 1: Proofs against canonical automata

Following Colvin et al., one is required to perform the following steps:

1. The linearization points of each concrete operation must be identified and fixed.
2. A canonical abstraction must be defined, together with a representation relation that describes the link between the abstract and concrete representations.

3. Simulation between the concrete program and canonical abstraction must be proved, where the concrete initialisation and responses are matched with abstract initialisation and response operations, respectively. The linearization points are matched with the main canonical operations.

4. The simulation may be performed in a forwards or backwards manner, and in some cases, both are required. The proof may involve introduction of additional invariants.

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

\[
AState \\
S: \mathbb{P} V \\
pc: P \rightarrow Label \\
v: P \rightarrow V
\]

The canonical operations corresponding to the \( \text{add} \) are given by the following Z schema:

\[
\begin{align*}
\text{AddInv}_p(x) & \quad \text{AddOK}_p \\
\Delta AState & \quad \Delta AState \subseteq AOp_p \circ \text{rep} \\
\begin{array}{l}
pc(p) = \text{idle} \\
v(p) \in S \\
v'(p) = x
\end{array} & \begin{array}{l}
pc(p) = \text{addi} \\
v(p) \in S \\
S' = S \cup \{v(p)\}
\end{array} & \begin{array}{l}
pc(p) = \text{addi} \\
v(p) \notin S \\
res'(p) = \text{false}
\end{array} & \begin{array}{l}
p'c(p) = \text{addi} \\
r = \text{res}(p)
\end{array}
\end{align*}
\]

Similar schema are generated for the canonical form of the \( \text{remove} \) operation. Following Lynch [73], 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 3, 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 state the following, where \( AOp_p \) is the abstract operation corresponding to the \( COp_p \) in process \( p \), and \( \circ \) denotes relational composition.

\[
\forall p: P \cdot \text{rep} \circ AOp_p \subseteq AOp_p \circ \text{rep}
\]

Thus, for any abstract state \( \sigma \) and concrete state \( \tau \) linked by the representation relation \( \text{rep} \), if the concrete statement \( COp_p \) is able to transition from \( \tau \) to \( \tau' \), then there must exist an abstract state \( \sigma' \) such that \( AOp_p \) can transition from \( \sigma \) to \( \sigma' \) and \( \sigma' \) is related to \( \tau' \) via \( \text{rep} \).

Checking two-state predicates is often cumbersome, and furthermore, Colvin et al. aim to set up a framework that enables model checking of possible invariants prior to its formal verification in a theorem prover. To this end, they introduce auxiliary variables to the concrete state space that
reflect the abstract space, and verify 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. Correctness of this encoding is verified by proving

\[
\text{cs}(aux_S) = \{ k \in V \mid \text{InList}(cs, k) \}
\]

where \( cs \) is an arbitrarily chosen concrete state and \( \text{InList} \) 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). The main invariants that Colvin et al. prove are given below\(^{19}\)\(^{15} \)[16].

\[
\forall p: P \cdot pc(p) \in \{ A5, R7 \} \Rightarrow v(p) \notin aux_S \\
\forall p: P \cdot pc(p) \in \{ A7, R3 \} \Rightarrow v(p) \in aux_S
\]

Thus, by (2), 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 (3) is similar. The representation relation between an abstract state \( as \) and concrete state \( cs \) is defined as follows, where \( \text{step}_\text{rel} \) is a relation between the program counters of \( as \) and \( cs \).

\[
\text{rep}(as, cs) \equiv as(S) = cs(aux_S) \land \text{step}_\text{rel}(as, cs)
\]

Proofs of these conditions require a number of additional invariants to be established, e.g., those that state the list is sorted. A substantial portion of these invariants relate to the proof of full lazy set. These proofs are carried out entirely within PVS \(^{85}\).

### 4.4 Method 2: Proofs against sequential specifications

Derrick et al. present a proof of the lazy set algorithm \(^ {21} \), from which we extract the proofs of the add and remove operations. Unlike the simulation-based methods in Section 4.3 and the method using auxiliary variables in Section 4.5 below, the proofs here capture data refinement between an abstract sequential specification and a concrete concurrent implementation to explicitly prove linearizability between the the sequential and concrete histories.

Verification using this method consists of the following steps:

1. The linearization points of each operation must be identified and fixed.

2. The proofs have to ensure that individual concrete runs actually correctly implement the abstract operations, i.e., every sequence of fine-grained transitions correctly produce the corresponding abstract coarse-grained transition.

3. To tackle concurrency, one must guarantee that other processes running in parallel maintain the refinement. To this end, \textit{interference freedom} and \textit{disjointness} proof obligations are encoded within the invariants.

4. Using a \textit{status} function, the global proof is decomposed into process-local proof obligations; the proof obligation that must be satisfied is dependant on the status of the current process.

\(^{19}\)In \(^ {16}\) \( A5 \) and \( A7 \) are labelled \textit{add6} and \textit{add8}, respectively.
5. Finally, correct initialisation must be guaranteed.

Unlike Colvin et al., whose abstraction is a canonical specification, the abstraction here is a fully sequential data type. The abstract add and remove operation are hence:

\[
\begin{align*}
\text{Add}_p & \equiv \Delta AState, x: V, r: \mathbb{B} \mid S' = S \cup \{x\} \land r = (S' \neq S) \\
\text{Remove}_p & \equiv \Delta AState, x: V, r: \mathbb{B} \mid S' = S \setminus \{x\} \land r = (S' \neq S)
\end{align*}
\]

The proofs rely on history-enhanced data structures, which introduce the sequential and concrete histories as auxiliary variables. Executing operations append events to a history, e.g., an invoke operation \( op \) with input \( x \) executed by process \( p \), appends \( \text{inv}(p, op, x) \) to the history. The abstract data types execute all operations atomically, and hence, their invocation and return occur in a single transition. Given that \( hs \) is the auxiliary sequential histories, the following formalises the history-enhanced add and remove operations:

\[
\begin{align*}
\text{AddH}_p(x, r) & \equiv \text{Add}_p(x, r) \land [hs, hs': \text{seq}(EVENT) \mid hs' = hs \setminus \{\text{inv}(p, \text{add}, x), \text{ret}(p, \text{add}, r)\}] \\
\text{RemoveH}_p(x, r) & \equiv \text{Remove}_p(x, r) \land [hs, hs': \text{seq}(EVENT) \mid hs' = hs \setminus \{\text{inv}(p, \text{rem}, x), \text{ret}(p, \text{rem}, r)\}]
\end{align*}
\]

Similarly, if \( h \) is the concrete history variable, the invocation and return schema of the add operation are extended as follows:

\[
\begin{align*}
\text{AddInvH}_p(x) & \equiv \text{AddInv}_p(x) \land [h, h': \text{seq}(EVENT) \mid h' = h \setminus \{\text{inv}(p, \text{add}, x)\}] \\
\text{AddRetH}_p(r) & \equiv \text{AddRet}_p(r) \land [h, h': \text{seq}(EVENT) \mid h' = h \setminus \{\text{ret}(p, \text{add}, r)\}]
\end{align*}
\]

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, and this is done using the encoding from Section 2.3.

The proofs here involve showing that each process is a non-atomic refinement [22, 23] of the abstract data type. To relate the concrete history \( h \) to an abstract history \( hs \), Derrick et al. use an additional set \( R \) that stores a set of return events for pending invokes whose effects have taken place, and therefore contributes to \( hs \). In particular, assuming \( \text{bseq}, X \) denotes bijective sequences of type \( X \), some \( h_0 \in \text{bseq}(R) \) can be used as the \( h_0 \) in Definition 4. Each process \( p \) may or may not contribute a return event \( \text{ret}(p, i, out) \) to the set \( R \). If it does, then there must be a pending invoke \( \text{inv}(p, i, x) \) in concrete history \( h \), and \( p \) must have already passed the linearization point and therefore modified the representation of the abstract data structure to implement \( AOp_p \). If it does not, then it either does not execute an operation at all, or it has a pending invoke \( \text{inv}(p, i, x) \) in its history.

To develop local proof obligations, each process is related with a status, which is of type:

\[
STATUS ::= \text{IDLE} \mid \text{IN} \langle V \rangle \mid \text{OUT} \langle V \rangle
\]

A process \( p \) has status \( \text{IDLE} \) iff \( p \) is not executing any operation, \( \text{IN}(x) \) iff \( p \) is executing an operation with input \( x \), but has not passed the linearization point of the operation, and \( \text{OUT}(r) \) iff \( p \) is executing an operation and has passed the linearization point with return value \( r \). This is combined with functions:
that transition from a status in the correct simulation condition. For example, the proof obligation below is for steps of process p
are generated, and a theorem that ensures these satisfaction of the process-local properties that 
hR a set 

The method of [100], consists of the following steps.

4.5 Method 3: Auxiliary variables

The method of [100], consists of the following steps.

20Thread-local in the terminology of [20].
1. Introduce auxiliary variables to the existing program, at least one of which is an abstraction of the data type in question. 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 as an augmentation to 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. This involves introducing appropriate annotations, and verification using the RGSep framework [104].

In this paper, for comparison, we present a translation of the RGSep framework into Z.

For the add operation, a state space is extended with a fresh variables AbsRes and S representing the abstract result and set, respectively to obtain an augmented state AugState. In addition the fixed linearization points A5 and A7 are augmented as follows, where the brackets <stmt> denote that stmt is executed atomically.

\[
\text{add}(x):
\]
\[
\ldots
\]
\[
A5: \langle n1.next := n2; \text{AbsRes} := (x \not\in S); S := (S \cup \{x\}) >
\]
\[
A7: \langle \text{res} := \text{false}; \text{AbsRes} := (x \not\in S); S := (S \cup \{x\}) > \ldots
\]

Note that for A7, the auxiliary code will set AbsRes to false (i.e., \(x \not\in S\)), and therefore the abstract set S remains unchanged. The remove operation is similar, and hence, its details are elided.

Translating Vafeiadis' separation logic notation into Z, and simplifying the notational overhead, we obtain the following relations. Function \(\text{lock}(n)\) returns the id of the process that currently holds the lock on node \(n\), where \(\text{lock}(n) = \emptyset\) holds if no process has locked \(n\). Assuming that \(\text{val}(n)\), \(\text{next}(n)\) and \(\text{mark}(n)\), denote the value, next and mark fields of \(n\), we define:

\[
\text{lvn}(n) \equiv (\text{lock}(n), \text{val}(n), \text{next}(n)) \quad \text{lvnm}(n) \equiv (\text{lock}(n), \text{val}(n), \text{next}(n), \text{mark}(n))
\]

The non-stuttering actions of a program’s environment are abstracted by rely conditions, which are relations on the pre-post states of an environment transitions that modify the global state (see below [21]). Because the abstract and concrete state spaces are disjoint, we replace all instances of separating conjunction ‘∗’ by logical conjunction ‘∧’, which enables simpler comparison among the different methods. We discuss the differences that arise from this translation where needed.

\[
\text{Lock}_p \equiv [\Delta \text{AugState}, n: \text{Node} \mid \text{lock}(n) = \emptyset \land p \neq 0 \land \text{lock}'(n) = p]
\]

\[
\text{Unlock}_p \equiv [\Delta \text{AugState}, n: \text{Node} \mid \text{lock}(n) = p \land p \neq 0 \land \text{lock}'(n) = \emptyset]
\]

\[
\text{Mark}_p \equiv [\Delta \text{AugState}, n, n1: \text{Node}, v: \text{V} \mid \text{lvn}(n1) = (p, v, n) \land \text{lvnm}'(n1) = (p, v, n, \text{true}) \land S' = S \setminus \{v\}]
\]

\[21[100]\] defines provisos for some of these actions, which suggests that they be interpreted as implication. For consistency with the Z formalism, we formalise these proviso predicates as preconditions of each action.
\[\text{Add}_p \equiv \begin{cases} \Delta \text{AugState}, \\ n_1, n_2, n_3: \text{Node}, u, v: \text{Val} \end{cases} \]
\[\begin{align*}
(u < v < \text{val}(n3)) & \land \text{lvn}(n1) = (p, u, n3) \\
\text{lock}(n3) & = p \land \text{lvn}(n2) = (\emptyset, v, n3) \\
\text{next}'(n1) & = n2 \land S' = S \cup \{v\}
\end{align*}\]
\[\text{Remove}_p \equiv \begin{cases} \Delta \text{AugState}, \\ n_1, n_2, n_3: \text{Node}, u, v: \text{Val} \end{cases} \]
\[\begin{align*}
\text{lvn}(n1) & = (p, u, n2) \land \text{lvn}(n2) = (p, v, n3, \text{true}) \\
\text{lvn}'(n1) & = (p, u, n3)
\end{align*}\]

The rely condition for process \(p\) is
\[\text{Rely}_p \equiv \bigvee_{q \in P \setminus \{p\}} \text{Lock}_q \lor \text{Unlock}_q \lor \text{Add}_q \lor \text{Mark}_q \lor \text{Remove}_q\]

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.

The methods of [100] require annotation of the code using separation logic-style assertions. In addition, building on the framework of [63], these assertions must be stable with respect to the rely conditions. The proof outlines for the lazy set are elided in [100], however, may be reconstructed from the other list examples in his thesis. We further adapt the proof outlines using Z-style notation. The invariants are formalised using the following predicates:

\[\begin{align*}
\text{ls}(x, A, y) & \equiv (x = y \land A = \langle \rangle) \lor \\
& \exists v, z, B \cdot x \neq y \land A = \langle v \rangle \cap B \land \text{val}(x) = v \land \text{next}(x) = z \land \text{ls}(z, B, y)
\end{align*}\]
\[\begin{align*}
\text{sorted}(A) & \equiv \text{if } A \in \{\langle \rangle, \langle a \rangle\} \text{ then true} \\
& \text{else if } A = \langle a, b \rangle \cap B \text{ then } (a < b) \land \text{sorted}(\langle b \cap B \rangle \text{ else false}
\end{align*}\]
\[s(A) \equiv S = \text{ran}(A) \setminus \{\pm \infty, \infty\}\]

Vafeiadis uses notation \([7]\) to denote that \(P\) that must hold under interference from the environment. This can notation can be avoided by checking all predicates against interference from the environment (e.g., as done in \([84]\)). The majority of the assertions refer to the global state, and hence would have been written as \([7]\) anyhow. Note that due to a typographical error, the failed case of the add operation is missing in \([100]\), however, it can be reconstructed from the remove operation (see Figure \([10]\). Of course, such annotations are not available in Z, but can easily be encoded as invariants on the overall specification by explicitly introducing a program counter variable \([28]\). 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:

\[\begin{align*}
\text{POA}_7_p & \equiv pc(p) = A7 \Rightarrow \exists n, A, B \cdot \text{ls}(\text{Head}, A, n_3p) \land \text{lvn}(n_3p) = (p, x, n) \land \\
& \text{ls}(n, B, \text{Tail}) \land s(A \cap \langle x \rangle \cap B)
\end{align*}\]

Such proof obligations must be resilient to interference from other processes (as in \([84]\)), hence, one must verify that the following holds for each \(p, q \in P\) such that \(p \neq q\),

\[\text{POA}_7_p \models \text{Env}_q \Rightarrow \text{POA}_7'_p\]

where \(\text{Env} \in \{\text{Lock}, \text{Unlock}, \text{Mark}, \text{Add}, \text{Remove}\}\).

\([67]\) provide outlines for the remove and contains operations albeit using a different framework, and define a number of additional predicates prior 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. Due to the similarities between the proof methods, we elide the details of such a transformation in this paper.
add(x) :
...
{∃u, v • ∃n, A, B • ls(Head, A, n1p) ∧ lvn(n1p) = (p, u, n3p)
  ∧ lvn(n3p) = (p, v, n) ∧ ls(n, B, Tail) ∧ s(A ∩ ⟨u, v⟩ ∩ B)}
  ∧ lvn(n2p) = (∅, x, n3p) ∧ u < x ∧ x < v

A5: <n1.next := n2; AbsRes := (x /∈ S); S := (S ∪ {x})>
{∃u, v • ∃n, A, B • ls(Head, A, n1p) ∧ lvn(n1p) = (p, u, n2p)
  ∧ lvn(n2p) = (p, x, n) ∧ ls(n, B, Tail) ∧ s(A ∩ ⟨u, x⟩ ∩ B)}
...
{∃n, A, B • ls(Head, A, n3p) ∧ lvn(n3p) = (p, x, n) ∧ ls(n, B, Tail) ∧ s(A ∩ ⟨x⟩ ∩ B)}
A7: <res := false; AbsRes := (x /∈ S); S := (S ∪ {x})>
{∃A • ls(Head, A, Tail) ∧ s(A)}
...

Figure 10: Reconstructed proof outline for add

4.6 Discussion

With the advances in linearizability verification correctness of the optimistic set is straightforward, and there is even the possibility of automating the verification procedure (e.g., by extending methods in [102,33][22]). We have presented a detailed account of three methods that manually identifies 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 is non-local and hence, the loss of this feature does not overly affect the proof. After this exercise, we discover some fundamental differences in the three methods.

- The abstraction in method 1 is a canonical program that generates concurrent traces unlike the abstraction in methods 2 and 3, where the abstraction is a sequential abstract specification.

- Method 2 directly relates proofs to Herlihy and Wing’s definition of linearizability. This link is indirectly established in methods 1 and 3.

- The proofs using methods 1 and 2 are mechanised. 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.

Note that the optimistic set in [102] does not use a marking scheme, and hence, is different from the algorithm in Section 4.1. The methods in [33] 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

Figure 11: The contains operation

contains(x) :
... contains(x) :
... C4a: r1 := !curr.mark;
C4b: res := r1 and (curr.val = x); C4c: r1 := (curr.val = x);
C4d: res := !curr.mark and r1;

Figure 12: Splitting atomicity (a) Figure 13: Splitting atomicity (b)

- Exploiting the symmetry between processes, each of the methods allows verification by only considering the execution a single process. Of these, 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 uses rely-guarantee reasoning, where the rely condition describes the potential interference from the environment.

The length of each proof and complexity of the assertions and invariants generated for each method is similar. Prior knowledge of a proof using one method can be transferred to a proof in another. The underlying challenge in verifying linearizability manifests itself 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 the corresponding abstract changes that occur at each linearization point.

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 from the optimistic set from Section 4.

5.1 The contains operation

A process executing contains(x) traverses the list (ignoring locks) and if a node with value greater or equal to x is found, it returns true if the node is unmarked and its value is equal to x, otherwise returns false. The contains operation is different from both add and remove in that it does not use any locks and performs a single traversal of the list (ignoring locks) looking for a given element. The operation returns true if an unmarked node with the value being searched is found, and returns false otherwise.

To reason about linearizability of the contains operation, one must first split the complex expression in C4 into statements C4a and C4b using a fresh local variable r1 (see Figure12). Note
that because the order in which the variables within a non-atomic expression are accessed is not fixed, at second possibility for ensuring atomicity within contains is to split C4 into C4c and C4d as in Figure 13. To verify linearizability of the original operation in Figure 11 both orders of evaluation must be verified. Derrick et al. and Vafeiadis prove the first variation (see Sections 5.3 and 5.4), while Groves et al. prove the second (see Section 5.2). It is however, also possible to directly verify the original algorithm by using frameworks that capture the non-determinism in expression evaluation under concurrency [55, 30] (see Section 5.5).

Unlike the add and remove operation, 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 data structure when C4a or C4b is executed, and this is impossible. To see this, suppose C4a is chosen as the linearization point of the contains operation. Now consider the following state of the shared linked list in Figure 14 (a), where process p is executing contains(50) and has just exited its loop because curr.val ≥ 50, but has not yet executed statement C4a. Now suppose another process q executes an add(50) operation to completion. This results in the linked list in Figure 14 (b), which corresponds to an abstract state {3, 18, 50}. Execution of process p from this state will set r1p 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.

Similarly, suppose C4b is chosen to be the linearization point of contains operation. Assume there are no other concurrent operations and that process p is executing contains(77) on the linked list in Figure 9 (a), and execution has reached (but not yet executed) statement C4b. This results in the state of the linked list in Figure 14 (c). Now suppose another process q executes a remove(77) operation to completion. This results in Figure 14 (d), corresponding to the abstract queue {3, 18}. Now, when process p executes C4b, it will set resp 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 point for contains.

An immediate question that now arises is whether contains is at all linearizable. The answer it turns out is yes. 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 [15]. 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 \texttt{contains}(x) returns \texttt{false}, there is some point in the interval of execution such that the corresponding abstract set does not contain $x$. However, as we have already seen, this point cannot be determined statically, i.e., by examining the statements within the \texttt{contains} operation alone. It turns out that \texttt{contains} is in fact linearized by the execution of an \texttt{add} or a \texttt{remove} operation.

5.2 Method 1: Proofs against canonical automata

The schemata corresponding to the canonical version of the \texttt{contains} operation are given by:

|                  | $ContInv_p(x)$ | $ContOK_p$ | $ContFail_p$ | $ContRes_p(r)$ |
|------------------|----------------|------------|--------------|----------------|
| $\Delta AState$ | $pc(p) = idle$| $pc(p) = CIn$ | $pc(p) = CIn$ | $pc(p) = COut$ |
|                  | $pc'(p) = CIn$| $v(p) \in S$ | $v(p) \notin S$ | $r = res(p)$  |
|                  | $v'(p) = x$   | $res'(p) = true$ | $res'(p) = false$ | $pc'(p) = idle$ |

Forward simulation cannot be used to prove that the \texttt{contains} operation implements the canonical abstraction (see [15] for details). Instead proofs are performed using backward simulation [18], which for a non-stuttering transition generates a proof obligation of the form:

$$\forall p: P \cdot COp_p \triangleright rep \subseteq rep \triangleright AOp_p$$

Thus, if $COp_p$ can transition from $\tau$ to $\tau'$ and $\tau'$ is related by $rep$ to some abstract state $\sigma'$, then there must exist an abstract state $\sigma$ such that $rep$ holds between $\tau$ and $\sigma$ and $AOp_p$ can transition from $\sigma$ to $\sigma'$. Such proofs involve reasoning from the end of computation to the start, and hence, are more complicated than just forward simulation.

To cope with this complexity, Colvin et al. split their simulation proofs by introducing an intermediate specification, that “eliminates the need to know the future [15, pg 481]”. They then prove backward simulation between the abstract 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 abstract as possible.

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

$$IContInv_p(x) \equiv ContInv_p(x) \land seen\_out'(p) = (x \notin S)$$
$$IRemOK_p(x) \equiv RemOK_p(x) \land$$
$$\forall q: P \cdot pc(q) = CIn \land v(q) = v(p) \Rightarrow seen\_out'(q) = true$$

The intermediate \texttt{contains}(x) operation is allowed to return \texttt{false} whenever $seen\_out(x)$ holds, therefore, $ContFail_p$ is replaced by $IContFail_p$ below:
Unlike ContFail_p, transition IContFail_p is enabled, and hence, res(p) can be set to false even if v(p) ∈ S holds in the current state. This is allowed because whenever pc(p) = Cin ∧ seen_out(p) holds, a state for which v(p) ∉ S holds must have occurred at some point since the invocation of the contains operation. When pc(p) = Cin ∧ seen_out(p) ∧ v(p) ∈ 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 abstract and intermediate state spaces is relatively straightforward because there is no data refinement between intermediate state is and abstract state as. In particular, one obtains:

\[
bsr(is, as) \equiv is(S) = as(S) \land \\
∀p \cdot is(pc(p)) = as(pc(p)) \lor \left(\begin{array}{l}
is(pc(p)) = Cin \land is(seen_out(p)) \land \\
as(pc(p)) = COut \land as(res(p)) = false
\end{array}\right)
\]

The second disjunct within the universal quantification is needed because p may have already executed ContFail_p in the abstract, and decided to return false, whereas the corresponding intermediate operation has not yet made its choice. This delay in choice 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 Section 4.3, this proof is simplified by introducing an auxiliary set auxS to the concrete code, which is updated in the same way as in Section 4.3. The proof allows the same forward simulation to be used, but additional invariants related to the contains operation must be introduced. For example:

\[
∀p \cdot \left(\begin{array}{l}
(pc(p) = 4c \land val(curr(p)) \neq v(p)) \lor \\
(pc(p) ∈ \{4c, 4d\} \land mark(curr(p)))
\end{array}\right) \Rightarrow seen_out(p)
\]

(8)

\[
∀p \cdot pc(p) = 4d \land \neg mark(curr(p)) \Rightarrow v(p) ∈ aux_out
\]

(9)

By (8), 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 (9), if the concrete program is in a position to return true, the value being searched must be in the set.

\subsection{5.3 Method 2: Proofs against sequential specifications}

Unlike the simulation against a canonical specification, the abstraction here is a sequential queue. The abstraction of the contains operation is therefore given by a single operation AbsCont_p.

\[
Contains_p \equiv [\Delta AState, x: V, r: Λ | r' = (x ∈ S)]
\]

To cope with the non-determinism in the linearization points, yet allow locality in the proof obligations generated, 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, where in and out denote the input and output parameters, respectively. Thus, in this new setting:

\[
STATUS ::= IDLE \mid IN\langle V \rangle \mid OUT\langle V \rangle \mid INOUT\langle V \times V \rangle
\]
For example, in the lazy set, process $p$ with status $INOUT(3, true)$ denotes a process that is potentially after its linearization point, has 3 as input and will return $true$.

As in Section 4.4, the proof proceeds by identifying the status for a concrete state and a process. For a contains($x$) operation executed by process $p$, given that $cs$ is a concrete state,

\[
\begin{align*}
    cs.pc(p) &= C1 \Rightarrow \text{status}(cs, p) = \text{IN}(x) \\
    cs.pc(p) &= \{C2, C3, C4a\} \Rightarrow \text{status}(cs, p) = \text{INOUT}(x, x \in cs.aux_S) \\
    cs.pc(p) &= C4b \Rightarrow \text{status}(cs, p) = \text{OUT}(cs.(\text{val}(\text{curr}(p)))) = x \\
    cs.pc(p) &= C5 \Rightarrow \text{status}(cs, p) = \text{OUT}(cs.(\text{res}(p)))
\end{align*}
\]

While executing $C2$, $C3$ or $C4a$, a contains operation may now “change its mind” about the linearization point and its outcome as often as necessary. The proof obligation requires that every 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, since the value being searched by $p$ is also removed from the set representation. A process $q$ adding a cell with $x$ after curr($p$) will change $p$’s status to $true$. Again this is justified, because the element being searched is also added to the abstract set.

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 Figure 3 (see [21] for details).

### 5.4 Method 3: Auxiliary variables

The auxiliary variable 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. As with the methods in Sections 5.2 and 5.3, a contains operation may linearize several times before returning, however, the output returned must be consistent with the state of the queue during the execution of contains.

The augmented state introduces a further auxiliary variable OutSet of type $P \times V \times B$, where $(p, v, r) \in \text{OutSet}$, 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 4.5. 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 4.5) and the auxiliary OutSet. 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 $C \subseteq \text{OutSet}$ that are currently executing a contains($v$) to false, which linearizes each of the processes in $C$.

| Mark$_p$ |
|----------|
| $\Delta \text{AugState}$ |
| $n, n1$: Node, $B, C$: $P \times V \times B$, $v$: $V, r$: $B$ |

| lvn(n1) = (p, v, n) \land \text{OutSet} = B \cup C \land (\forall b: B \cdot b.2 \neq v) \land (\forall c: C \cdot c.2 = v) |
| lvn'(n1) = (p, v, n, true) \land S' = S - \{v\} \land |
| OutSet' = B \cup \{(q, v, false) | \exists r \cdot (q, v, r) \in C\} |
In addition, two environment operations that add and remove triples of type $P \times V \times B$ to/from the auxiliary variable $OutSet$ are also introduced. These represent environment processes that start and complete a contains operation.

\[
\begin{align*}
AddOut_p & \equiv \Delta AugState, v: V, r: B \quad (\forall o: OutSet \cdot o.1 \neq p) \land \\
RemOut_p & \equiv \Delta AugState, v: V, r: B \\
\text{OutSet}' &= \text{OutSet} \cup \{(p, v, r)\} \\
\text{OutSet}' &= \text{OutSet} \setminus \{(p, v, r)\}
\end{align*}
\]

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

\[
\text{remove}(x):\]

\[
\text{R3: } <\text{n2.mark := true; AbsRes(this) := (x} \in S); \\
\quad \text{for each } q \in \text{OutSet do if } q.2 = \text{n2.val then AbsRes(q) := false}> ...
\]

The augmented version of the contains operation is given below. The presentation in [100] suffers from a few typos, which are confirmed by the proof in [67]. 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 in [100]. Below, like [100], details of the annotation for the proof outline are elided, but the interested reader may consult [67].

\[
\text{contains}(x) : \]

\[
\text{C1: curr := Head; } \\
\text{C4a: } <r1 := \text{curr.marked; } \\
\quad \text{AbsRes(this) := (not r1 and curr.val = x); OutOps := OutOps } \setminus \{\text{this}\}>
\]

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 OutSet. This allows other processes executing remove($x$) to set $p$’s linearization point when $x$ is marked (and 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 Method 4: Verification without linearization points

The proof methods above have considered the effects of a single atomic transition on the concrete state on the abstract state. When reasoning about concurrency however, one must inevitably reason about the interference from other processes. In particular, linearizability is concerned with the behaviour of an operation over time, where one considers the interval of execution of an operation from its invocation to its response, and requires that the linearization point occurs at some point within this interval. Interval-based methods aim to exploit such reasoning directly using logics such as ITL [79 80] and its variations (e.g., [31 32 89 7]). Schellhorn et al. have exploited rely-guarantee reasoning over intervals to mechanically verify a number of algorithms [89 7]. These
however, in many ways resemble the methods described in Sections 5.2-5.4 in that they involve identification of linearization points as well as the instantiation of invariants, rely conditions and representation relations between the abstract and concrete state spaces. As we have already seen, for the lazy set algorithm identification of linearization points and appropriate abstractions so that verification may be simplified is non-trivial.

Our methods in [29, 30] take inspiration from the reduction-based approaches, where proofs of linearizability are decomposed into two steps. The first step proves that a fine-grained implementation refines a program that executes the same operations but with coarse-grained atomicity. The second step of the proof is to show that the abstraction is linearizable. Because behaviour refinement is established, any behaviour of the fine-grained implementation is a possible behaviour of the coarse-grained abstraction, and hence, an implementation is linearizable whenever the abstraction is linearizable. The advantage of this technique is that it does not require identification of the linearizing statements in the implementation.

Unlike reduction, which is a non-compositional method, the methods in [30, 29] allow rely-guarantee reasoning. Furthermore, verification of the abstraction is possible by appealing to a variation of Interval Temporal Logic. In particular, the basic semantic model uses interval predicates, which allows formalisation of a program’s behaviour with respect to an interval (which is a contiguous set of times), and an infinite stream (that maps each time to a state). Unlike Jones [63], who assumes rely conditions are two-state relations, rely conditions in our framework are interval predicates that are able to refer to an arbitrary number of states. The logic allows non-deterministic evaluation of expressions [55], and hence both orders of evaluation $C_4a ; C_4b$ and $C_4c ; C_4d$ (see Section 5.1) are considered simultaneously. For example, consider Figure 15 which depicts an execution of the lazy set over interval $\Delta$ in stream $s$, a process $p$ that executes a $\text{contains}(x)$ that returns $true$ over $\Delta_p$, a process $q$ that executes $\text{remove}(x)$ and $\text{add}(y)$ over intervals $\Delta_q$ and $\Delta'_q$, respectively, and a process $u$ that executes $\text{add}(x)$ over interval $\Delta_u$. The shared data structure may be changing over $\Delta_p$ while process $p$ is checking $x \in S$.

A possible coarse-grained abstraction of $\text{contains}(x)$ is an operation that tests whether $x$ is in the set in a single atomic step, unlike the implementation in Figure 11 which uses a sequence of atomic steps to iterate through the list to search for a node with value $x$. Therefore, as depicted in Figure 15 an execution of $\text{contains}$ that returns $true$, i.e., given that $C^\omega$ denotes possibly infinite iteration of $C$, one must prove both

$$\langle x \in S \rangle ; \text{return true} \sqsubseteq C_1 ; (C_2 ; C_3)^\omega ; C_4 ; \text{return true} \quad (R1)$$

$$\langle x \in S \rangle ; \text{return false} \sqsubseteq C_1 ; (C_2 ; C_3)^\omega ; C_4 ; \text{return false} \quad (R2)$$
where $C_1 - C_4$ are the labels of contains in Figure 11 and $(x \in S)$ is a guard that is atomically able to test whether $x$ is in the abstract set $S$. In particular, $(x \in S)$ holds in an interval $\Omega$ and stream $s$ iff $x \in S$ holds the state corresponding to time $t$, where $t \in \Omega$. Note that both $(x \in S)$ and $(x \notin S)$ may hold for $\Delta_p$; the refinement (R1) above would only be invalid if $x \notin S$ holds for each time $t \in \Delta_p$.

### 5.6 Discussion

The lazy set represents a class of algorithms that can be verified by allowing an operation to set the linearization point of another, and its proof is therefore more involved than, say the Treiber stack (Section 2.1) or optimistic set (Section 4), which can be verified by allowing each operation to only linearize itself. The four 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 methods 1-3 are similar, and experience in verification using one of these methods can aid in the proof in another. For each of these methods, identification of the linearization points, and developing an understanding of its effects on the data structure remains the difficult part of the proof.

Method 4 aims to alleviate these issues by presenting an altogether different approach, where the proof is decomposed into multiple steps, separating out a proof of linearizability from a proof of atomicity refinement. The atomicity refinement proof need not refer to the linearization points of the implementation. In addition, the linearization point of the coarse-grained contains operation is within the contains itself, unlike the fine-grained program, which may be linearized by the execution of other operations.

### 6 Conclusions

There has been remarkable progress since Herlihy and Wing’s original paper on linearizability [62], and with the increasing necessity for concurrency (in particular shared-variable 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. Numerous alternative definitions of linearizability have been developed (e.g., Section 2.3 and speculative linearizability [51]). In some cases linearizability and atomicity may be used synonymously [88]. Filipović et al. [38] have also formulated linearizability of concurrent object systems (which are collections of concurrent data structures) is an instance of observational refinement; a concrete system observationally refines an abstract system if any observation made on the concrete can also be made on the abstract. Linearizability has also been applied to distributed systems [10], databases [87] and fault-tolerant systems [52]. Stronger notions of linearizability such as $R$-linearizability have also been developed [86], which is used to define consistency of replicated databases.

With the increasing popularity of multi-core, multi-process architectures, weaker consistency conditions than linearizability have also been defined in order to allow greater flexibility in an implementation, and therefore increase the scope for optimisation. Such is the necessity for optimised
algorithms within these architectures that researchers have begun questioning whether linearizability is itself becoming a bottleneck to optimisation. Part of the problem is that linearizability insists on sequential consistency \cite{66, 60}, 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 a number of weaker only ensure weaker guarantees \cite{4}. Shavit purports quiescent consistency as the correctness criteria for data structures in the multicore age \cite{93}. Quiescent consistency is a weaker condition than linearizability requiring a valid linearization for operation calls separated by a period of quiescence (which is a period in without any pending operation invocations). Unlike linearizability, quiescent consistency does not guarantee sequential consistency. To increase the potential for optimisation, researchers have also considered weakening the linearizability criteria itself, with the introduction of properties such as quasi-linearizability \cite{5}, \(k\)-linearizability \cite{58}, and eventual consistency \cite{92}.

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 \cite{60, 61, 28, 27, 95}). Progress properties are not the main focus of this paper, and hence, discussion of methods for verifying them have been elided.

This paper considered verification of linearizability only, and the associated proof methods that have been developed for it in the context of concurrent data structures. Necessity of such proofs is alluded to by the subtleties in the behaviours of the algorithms that implement concurrent data structures, and by the fact that it errors have been found in algorithms that were previously believed to be correct \cite{25, 14}. 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 \cite{25}. However, number of lines of code is not an indicator of complexity, with even simple algorithms like Herlihy and Wing’s queue \cite{62} posing immense challenges \cite{90, 59} due to the fact that future behaviour must be considered.

An important strand of research is model checking, which does not prove linearizability, but can provide assurances that linearizability holds. This paper has focused on verification methods, and hence, a detailed comparison of model checking methods have been elided. However, like Colvin et al. \cite{15}, we believe model checking can play a complementary role in verification, allowing invariants to be model checked to provide assurances that they can be proved correct. Methods for model checking linearizability may be found in \cite{106, 40, 70, 71}; a detailed comparison of these is beyond the scope of this survey.

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

**Compositionality of the proof method.** Each of the refinement-based methods 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. If a verification fails, modification of an invariant or refinement relation would require all previously verified proof obligations to be reproved. Reduction of this proof load is achieved by frameworks that use rely/guarantee reasoning (e.g., RGSep and RGITL) allow further
compositionality by using a rely condition to abstractly define the behaviour of a component’s environment. Here, an additional step of reasoning is required to show that the rely condition does indeed capture the potential interference from other processes.

**Contribution of the underlying framework.** None of the existing frameworks thus far provide a silver bullet for linearizability verification, and the choice of framework that a verifier uses can come down to personal preference. 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 [102], which allows linearizability to be automatically verified. For more complex algorithms, using a setup such as the one provided by Colvin et al. [15] would allow invariants to be model checked prior to verification. On the other hand, Derrick et al. have defined a systematic method for constructing representation relations, invariants and interference freedom conditions as well as proof obligations that enables process-local verification [20].

**Algorithms verified.** A survey of these has been given in Section 2.4. There exist several other algorithms in the literature whose linearizability has been conjectured, but a fully formal verification has not yet been performed.

**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 [90]. Furthermore, a combination of forwards and backwards simulations is known to be complete for data refinement [73, 18], and combining auxiliary and prophecy variables is known to be complete for reasoning about past and future behaviour [1]. However, completeness of a method does not guarantee simpler proofs, as evidenced by the maximal backwards simulation used in [90].

Overall, verifying concurrent algorithms in a scalable manner remains an open problem. For algorithms with linearization points that depend on future behaviour, scalability can only be achieved with approaches capable of dealing with the non-determinism of future behaviours. 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 (see Section 5.5). Until a scalable general solution is found, it is worthwhile pursuing problem specific approaches (e.g., [59, 33, 83]).

**References**

[1] M. Abadi and L. Lamport. The existence of refinement mappings. *Theor. Comput. Sci.*, 82(2):253–284, 1991.

39
[2] J.-R. Abrial. *Modeling in Event-B - System and Software Engineering*. Cambridge Univ. Press, 2010.

[3] J.-R. Abrial and D. Cansell. Formal construction of a non-blocking concurrent queue algorithm. *Journal of Universal Computer Science*, 11(5):744–770, May 2005.

[4] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Comp.*, 29(12):66–76, 1996.

[5] Y. Afek, G. Korland, and E. Yanovsky. Quasi-linearizability: Relaxed consistency for improved concurrency. In C. Lu, T. Masuzawa, and M. Mosbah, editors, *OPODIS*, volume 6490 of *LNCS*, pages 395–410. Springer, 2010.

[6] D. Amit, N. Rinetzky, T. W. Reps, M. Sagiv, and E. Yahav. Comparison under abstraction for verifying linearizability. In W. Damm and H. Hermanns, editors, *CAV*, volume 4590 of *LNCS*, pages 477–490. Springer, 2007.

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

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

[9] R. Bayer and M. Schkolnick. Concurrency of operations on b-trees. *Acta Inf.*, 9:1–21, 1977.

[10] K. P. Birman. Maintaining consistency in distributed systems. In *Proceedings of the 5th Workshop on ACM SIGOPS European Workshop: Models and Paradigms for Distributed Systems Structuring*, EW 5, pages 1–6, New York, NY, USA, 1992. ACM.

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

[12] R. M. Burstall. Program proving as hand simulation with a little induction. In *IFIP Congress*, pages 308–312, 1974.

[13] R. Colvin, S. Doherty, and L. Groves. Verifying concurrent data structures by simulation. *Electr. Notes Theor. Comput. Sci.*, 137(2):93–110, 2005.

[14] R. Colvin and L. Groves. Formal verification of an array-based nonblocking queue. In *ICECCS*, pages 507–516. IEEE Computer Society, 2005.

[15] R. Colvin, L. Groves, V. Luchangco, and M. Moir. Formal verification of a lazy concurrent list-based set algorithm. In T. Ball and R. B. Jones, editors, *CAV*, volume 4144 of *LNCS*, pages 475–488. Springer, 2006.

[16] R. Colvin, L. Groves, V. Luchangco, and M. Moir. Pvs proofs for lazy set algorithm, 2006.
[17] W. P. de Roever, F. de Boer, U. Hannemann, J. Hooman, Y. Lakhnech, M. Poel, and J. Zwiers. *Concurrency Verification: Introduction to Compositional and Noncompositional Methods*. Cambridge University Press, 2001.

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

[19] J. Derrick, G. Schellhorn, and H. Wehrheim. Proving linearizability via non-atomic refinement. In J. Davies and J. Gibbons, editors, *iFM*, volume 4591 of *LNCS*, pages 195–214. Springer, 2007.

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

[21] J. Derrick, G. Schellhorn, and H. Wehrheim. Verifying linearisability with potential linearisation points. In M. Butler and W. Schulte, editors, *FM*, volume 6664 of *LNCS*, pages 323–337. Springer, 2011.

[22] J. Derrick and H. Wehrheim. Using coupled simulations in non-atomic refinement. In *ZB 2003: Formal Specification and Development in Z and B*, number 2651 in *LNCS*, pages 127–147. Springer, 2003.

[23] J. Derrick and H. Wehrheim. Non-atomic refinement in z and csp. In H. Treharne, S. King, M. C. Henson, and S. A. Schneider, editors, *ZB*, volume 3455 of *LNCS*, pages 24–44. Springer, 2005.

[24] S. Doherty. Modelling and verifying non-blocking algorithms that use dynamically allocated memory. Master’s thesis, Victoria University of Wellington, 2003.

[25] S. Doherty, D. Detlefs, L. Groves, C. H. Flood, V. Luchangco, P. A. Martin, M. Moir, N. Shavit, and G. Steel. Dcas is not a silver bullet for nonblocking algorithm design. In *SPAA*, pages 216–224, 2004.

[26] S. Doherty, L. Groves, V. Luchangco, and M. Moir. Formal verification of a practical lock-free queue algorithm. In D. de Frutos-Escrig and M. Núñez, editors, *FORTE*, volume 3235 of *LNCS*, pages 97–114. Springer, 2004.

[27] B. Dongol. Formalising progress properties of non-blocking programs. In Z. Liu and J. He, editors, *ICFEM*, volume 4260 of *LNCS*, pages 284–304. Springer, 2006.

[28] B. Dongol. *Progress-based verification and derivation of concurrent programs*. PhD thesis, The University of Queensland, 2009.

[29] B. Dongol and J. Derrick. Proving linearisability via coarse-grained abstraction. *CoRR*, abs/1212.5116, 2012.
[30] B. Dongol and J. Derrick. Verifying linearisability of a concurrent list-based lazy via layers of abstraction. In AVoCS, 2013. To appear.

[31] B. Dongol, J. Derrick, and I. J. Hayes. Fractional permissions and non-deterministic evaluators in interval temporal logic. ECEASST, 53, 2012.

[32] B. Dongol, I. J. Hayes, and J. Derrick. Deriving real-time action systems with multiple time bands using algebraic reasoning. Science of Computer Programming, (0):–, 2013.

[33] C. Dragoi, A. Gupta, and T. A. Henzinger. Automatic linearizability proofs of concurrent objects with cooperating updates. In N. Sharygina and H. Veith, editors, CAV, volume 8044 of LNCS, pages 174–190. Springer, 2013.

[34] R. Drexler, W. Reif, G. Schellhorn, K. Stenzel, W. Stephan, and A. Wolpers. The kiv system: A tool for formal program development. In P. Enjalbert, A. Finkel, and K. W. Wagner, editors, STACS, volume 665 of LNCS, pages 704–705, 1993.

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

[36] T. Elmas, S. Qadeer, A. Sezgin, O. Subasi, and S. Tasiran. Simplifying linearizability proofs with reduction and abstraction. In J. Esparza and R. Majumdar, editors, TACAS, volume 6015 of LNCS, pages 296–311. Springer, 2010.

[37] T. Elmas, S. Qadeer, and S. Tasiran. A calculus of atomic actions. In Z. Shao and B. C. Pierce, editors, POPL, pages 2–15. ACM, 2009.

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

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

[40] D. Friggens. On the Use of Model Checking for the Bounded and Unbounded Verification of Nonblocking Concurrent Data Structures. PhD thesis, Victoria University of Wellington, 2013.

[41] H. Gao, Y. Fu, and W. H. Hesselink. Verification of a lock-free implementation of multiword ll/sc object. In DASC, pages 31–36. IEEE, 2009.

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

[43] H. Gao, J. F. Groote, and W. H. Hesselink. Lock-free parallel and concurrent garbage collection by mark&sweep. Sci. Comput. Program., 64(3):341–374, 2007.
[44] H. Gao and W. H. Hesselink. A general lock-free algorithm using compare-and-swap. \textit{Inf. Comp.}, 205(2):225–241, 2007.

[45] L. Groves. Reasoning about nonblocking concurrency using reduction. In \textit{ICECCS}, pages 107–116. IEEE Comp. Soc., 2007.

[46] L. Groves. Trace-based derivation of a lock-free queue algorithm. \textit{Elect. Notes Theo. Comp. Sci.}, 201:69–98, 2008.

[47] L. Groves. Verifying Michael and Scott’s lock-free queue algorithm using trace reduction. In J. Harland and P. Manyem, editors, \textit{CATS}, volume 77 of \textit{CRPIT}, pages 133–142, 2008.

[48] L. Groves. Reasoning about nonblocking concurrency. \textit{JUCS}, 15(1):72–111, Jan 2009.

[49] L. Groves and R. Colvin. Derivation of a scalable lock-free stack algorithm. \textit{Electr. Notes Theor. Comput. Sci.}, 187:55–74, 2007.

[50] L. Groves and R. Colvin. Trace-based derivation of a scalable lock-free stack algorithm. \textit{Formal Asp. Comput.}, 21(1-2):187–223, 2009.

[51] R. Guerraoui, V. Kuncak, and G. Losa. Speculative linearizability. In J. Vitek, H. Lin, and F. Tip, editors, \textit{PLDI}, pages 55–66. ACM, 2012.

[52] R. Guerraoui and A. Schiper. Fault-tolerance by replication in distributed systems. In Al. Strohmeier, editor, \textit{Reliable Software Technologies Ada-Europe '96}, volume 1088 of \textit{LNCS}, pages 38–57. Springer Berlin Heidelberg, 1996.

[53] T. L. Harris. A pragmatic implementation of non-blocking linked-lists. In J. L. Welch, editor, \textit{DISC}, volume 2180 of \textit{LNCS}, pages 300–314. Springer, 2001.

[54] T. L. Harris, K. Fraser, and I. A. Pratt. A practical multi-word compare-and-swap operation. In D. Malkhi, editor, \textit{DISC}, volume 2508 of \textit{LNCS}, pages 265–279. Springer, 2002.

[55] I. J. Hayes, A. Burns, B. Dongol, and C. B. Jones. Comparing degrees of non-determinism in expression evaluation. \textit{Comput. J.}, 56(6):741–755, 2013.

[56] S. Heller, M. Herlihy, V. Luchangco, M. Moir, W. N. Scherer III, and N. Shavit. A lazy concurrent list-based set algorithm. \textit{Parallel Processing Letters}, 17(4):411–424, 2007.

[57] D. Hendler, N. Shavit, and L. Yerushalmi. A scalable lock-free stack algorithm. \textit{J. Parallel Distrib. Comput.}, 70(1):1–12, 2010.

[58] T. A. Henzinger, C. M. Kirsch, H. Payer, A. Sezgin, and A. Sokolova. Quantitative relaxation of concurrent data structures. In R. Giacobazzi and R. Cousot, editors, \textit{POPL}, pages 317–328. ACM, 2013.
[59] T. A. Henzinger, A. Sezgin, and V. Vafeiadis. Aspect-oriented linearizability proofs. In P. R. D’Argenio and H. C. Melgratti, editors, CONCUR, volume 8052 of LNCS, pages 242–256. Springer, 2013.

[60] M. Herlihy and N. Shavit. The art of multiprocessor programming. Morgan Kaufmann, 2008.

[61] M. Herlihy and N. Shavit. On the nature of progress. In A. Fernández Anta, G. Lipari, and M. Roy, editors, OPODIS, volume 7109 of LNCS, pages 313–328. Springer, 2011.

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

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

[64] N. D. Jones and S. S. Muchnick. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In R. A. DeMillo, editor, POPL, pages 66–74. ACM Press, 1982.

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

[66] L. Lamport. How to make a correct multiprocess program execute correctly on a multiprocessor. IEEE Trans. Computers, 46(7):779–782, 1997.

[67] H. Liang and X. Feng. Modular verification of linearizability with non-fixed linearization points. Technical report, USTC, March 2013.

[68] H. Liang and X. Feng. Modular verification of linearizability with non-fixed linearization points. In H.-J. Boehm and C. Flanagan, editors, PLDI, pages 459–470. ACM, 2013.

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

[70] Y. Liu, W. Chen, Y. A. Liu, and J. Sun. Model checking linearizability via refinement. In A. Cavalcanti and D. Dams, editors, FM, volume 5850 of LNCS, pages 321–337. Springer, 2009.

[71] Y. Liu, W. Chen, Y. A. Liu, J. Sun, S. J. Zhang, and J. Song Dong. Verifying linearizability via optimized refinement checking. IEEE Trans. Software Eng., 39(7):1018–1039, 2013.

[72] N. Lynch and M. Tuttle. An introduction to input/output automata. CWI-Quarterly, 2(3):219–246, 1989.

[73] N. A. Lynch. Distributed Algorithms. Morgan Kaufmann, 1996.
[74] H. Massalin and C. Pu. A lock-free multiprocessor os kernel (abstract). *Operating Systems Review*, 26(2):8, 1992.

[75] M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *PODC*, pages 267–275, May 1996.

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

[77] M. Moir, D. Nussbaum, O. Shalev, and N. Shavit. Using elimination to implement scalable and lock-free fifo queues. In P. B. Gibbons and P. G. Spirakis, editors, *SPAA*, pages 253–262. ACM, 2005.

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

[79] B. C. Moszkowski. Compositional reasoning using interval temporal logic and Tempura. In W. P. de Roever, H. Langmaack, and A. Pnueli, editors, *COMPOS*, volume 1536 of *LNCS*, pages 439–464. Springer, 1997.

[80] B. C. Moszkowski. A complete axiomatization of Interval Temporal Logic with infinite time. In *LICS*, pages 241–252, 2000.

[81] P. W. O’Hearn, J. C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In L. Fribourg, editor, *CSL*, volume 2142 of *LNCS*, pages 1–19. Springer, 2001.

[82] P. W. O’Hearn, N. Rinetzky, M. T. Vechev, E. Yahav, and G. Yorsh. Verifying linearizability with hindsight. In A. W. Richa and R. Guerraoui, editors, *PODC*, pages 85–94. ACM, 2010.

[83] P. W. O’Hearn, N. Rinetzky, M. T. Vechev, E. Yahav, and G. Yorsh. Verifying linearizability with hindsight. Technical report, Queen Mary University of London, 2010.

[84] S. Owicki and D. Gries. Verifying properties of parallel programs: An axiomatic approach. *Commun. ACM*, 19(5):279–285, 1976.

[85] S. Owre, S. Rajan, J. M. Rushby, N. Shankar, and M. K. Srivas. PVS: Combining specification, proof checking, and model checking. In R. Alur and T. A. Henzinger, editors, *CAV*, volume 1102, pages 411–414. Springer Verlag, / 1996.

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

[87] K. Ramamritham and P. K. Chrysanthis. In search of acceptability citeria: Database consistency requirements and transaction correctness properties. In M. T. Özsu, U. Dayal, and P. Valduriez, editors, *IWDOM*, pages 212–230. Morgan Kaufmann, 1992.
[88] M. Raynal. *Distributed Algorithms for Message-Passing Systems*, chapter Atomic Consistency (Linearizability), pages 427–446. Springer, 2013.

[89] G. Schellhorn, B. Tofan, G. Ernst, and W. Reif. Interleaved programs and rely-guarantee reasoning with itl. In C. Combi, M. Leucker, and F. Wolter, editors, *TIME*, pages 99–106. IEEE, 2011.

[90] G. Schellhorn, H. Wehrheim, and J. Derrick. How to prove algorithms linearisable. In P. Madhusudan and S. A. Seshia, editors, *CAV*, volume 7358 of *LNCS*, pages 243–259. Springer, 2012.

[91] C.-H. Shann, T.-L. Huang, and C. Chen. A practical nonblocking queue algorithm using compare-and-swap. In *ICPADS*, pages 470–475, 2000.

[92] M. Shapiro and B. Kemme. Eventual consistency. In L. Liu and M. T. Özsu, editors, *Encyclopedia of Database Systems*, pages 1071–1072. Springer US, 2009.

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

[94] G. Smith. *The Object-Z Specification Language*. Advances in Formal Methods. Springer US, 1999.

[95] B. Tofan, S. Bäumler, G. Schellhorn, and W. Reif. Temporal logic verification of lock-freedom. In C. Bolduc, J. Desharnais, and B. Ktari, editors, *MPC*, volume 6120 of *LNCS*, pages 377–396. Springer, 2010.

[96] B. Tofan, G. Schellhorn, and W. Reif. Formal verification of a lock-free stack with hazard pointers. In A. Cerone and P. Pihlajasaaari, editors, *ICTAC*, volume 6916 of *LNCS*, pages 239–255. Springer, 2011.

[97] O. Travkin, H. Wehrheim, and G. Schellhorn. Proving linearizability of multiset with local proof obligations. *ECEASST*, 53, 2012.

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

[99] A. J. Turon and M. Wand. A separation logic for refining concurrent objects. In T. Ball and M. Sagiv, editors, *POPL*, pages 247–258. ACM, 2011.

[100] V. Vafeiadis. *Modular fine-grained concurrency verification*. PhD thesis, University of Cambridge, 2007.

[101] V. Vafeiadis. Shape-value abstraction for verifying linearizability. In N. D. Jones and M. Müller-Olm, editors, *VMCAI*, volume 5403 of *LNCS*, pages 335–348. Springer, 2009.

[102] V. Vafeiadis. Automatically proving linearizability. In T. Touili, B. Cook, and P. Jackson, editors, *CAV*, volume 6174 of *LNCS*, pages 450–464. Springer, 2010.
[103] V. Vafeiadis, M. Herlihy, T. Hoare, and M. Shapiro. Proving correctness of highly-concurrent linearisable objects. In J. Torrellas and S. Chatterjee, editors, *PPOPP*, pages 129–136, 2006.

[104] V. Vafeiadis and M. J. Parkinson. A marriage of rely/guarantee and separation logic. In L. Caires and V. T. Vasconcelos, editors, *CONCUR*, volume 4703 of *LNCS*, pages 256–271. Springer, 2007.

[105] M. T. Vechev and E. Yahav. Deriving linearizable fine-grained concurrent objects. In R. Gupta and S. P. Amarasinghe, editors, *PLDI*, pages 125–135. ACM, 2008.

[106] M. T. Vechev, E. Yahav, and G. Yorsh. Experience with model checking linearizability. In C. S. Pasareanu, editor, *SPIN*, volume 5578 of *LNCS*, pages 261–278. Springer, 2009.

[107] L. Wang and S. D. Stoller. Static analysis of atomicity for programs with non-blocking synchronization. In K. Pingali, K. A. Yelick, and A. S. Grimshaw, editors, *PPOPP*, pages 61–71. ACM, 2005.