Verifying linearizability: A comparative survey
BRIJESH DONGOL and JOHN DERRICK, University of Sheffield

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 if 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. In this paper we compare the major of methods used to verify linearizability, describe the main contribution of each method, and compare the advantages and limitations of each.

Categories and Subject Descriptors: D.2.4 [Software Engineering]: Software/Program Verification
General Terms: Algorithms, Theory, Verification
Additional Key Words and Phrases: Linearizability, concurrent data structures, refinement

ACM Reference Format:
DOI: http://dx.doi.org/10.1145/0000000.0000000

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 [Doherty 2003; Colvin and Groves 2005]. The main correctness criterion for concurrent algorithms is linearity, which defines consistency conditions on the history of invocation and response events generated by an execution of the algorithm at hand [Herlihy and Shavit 2008; Herlihy and Wing 1990]. 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 if there is some order for the effects of the operation calls that corresponds to a valid sequential history. A concurrent data structure is linearizable if 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 underlying framework. How does the underlying framework contribute to the proof, if at all?
— Algorithms verified. What are the algorithms that 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 [Herlihy and Shavit 2008] 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 [Treiber 1986] have simple fixed linearization points. Others such as the Heller et al set [Heller et al. 2007] have linearization points that are determined by the execution of operations. Yet more complex algorithms include the Herlihy-Wing queue [Herlihy and Wing 1990], 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 [Heller et al. 2007], 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 fixed linearization points. On the other hand, verifying linearizability of contains is complex, and requires consideration of other operations [Colvin et al. 2006a; Derrick et al. 2011b].

This paper is structured as follows. In Section 2, we describe linearizability, present its original definition using the nomenclature of Herlihy/Wing, and Derrick et al.’s alternative definition that uses matching functions. In Section 3, we present an overview of the different methods that have been developed for verifying linearizability, which includes simulation, data refinement, auxiliary variables, shape analysis, etc. Section 4 presents our first case study, which is a simplified version of the set algorithm with add and remove operations but without a contains operation. Section 5 presents a more complicated algorithm, namely, the lazy set case by Heller et al. [Heller et al. 2007]. Section 6 concludes the paper.

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. To motivate our discussion, we present Treiber’s stack algorithm [Treiber 1986] in Section 2.1. The formal definition of linearizability is given in Section 2.2 and an alternative formulation that enables its formal verification in Section 2.3. We discuss different classifications of algorithms in terms of the degree of difficulty of their linearizability verification in Section 2.4.

### 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 \textit{push} and \textit{pop}. Operation \textit{push} pushes its input \( v \neq \text{empty} \) on the stack, and \textit{pop} removes one element that is returned in \( lv \). When the stack is empty, \textit{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 \textit{Head} is used to keep track of the current head of the list. Operations \textit{push} and \textit{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: \textit{CAS} \((\text{mem}, \text{exp}, \text{new})\) (compare-and-swap), which compares \text{mem} to \text{exp}; if this succeeds (i.e. \text{mem} equals \text{exp}), \text{mem} is set to \text{new} and \textit{CAS} returns \text{true}, otherwise the \textit{CAS} fails, leaves \text{mem} unchanged and returns \text{false}. In pseudo-code the push and pop operations that one process executes are given in Figure 1.

The \textit{push} operation first creates a new node with the value to be pushed onto the stack. It then repeatedly sets a local variable \textit{ss} to \textit{Head} and the pointer of the new node to \textit{ss}. This ends once the final \textit{CAS} detects that \textit{Head} (still) equals \textit{ss} upon which \textit{Head} is set to the new node \textit{n}. Note that the \textit{CAS} in \textit{push} does not necessarily succeed: in case of a concurrent pop, \textit{Head} might have been changed in between. The \textit{pop} is similar: it records the value of \textit{Head} in \textit{ss}, then determines the remaining list and the output value. If \textit{Head} is still equal to \textit{ss} in the end, the \textit{pop} takes effect and the output value is returned. If not, the \textit{pop} repeats its steps with a new value of \textit{Head}.

Note that processes executing these operations now have a shared \textit{global} state (in this case \textit{Head}) as well as \textit{local} variables (e.g., \textit{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 \textit{linearization point}. For example, the \textit{CAS} in the \textit{pop} operation is a linearization point for \textit{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 in line 2, when \textit{Head = null} is read (note that it is not line 3, which checks the \textit{local} variable \textit{ss} only to be null. The global variable \textit{Head} might be non-null again at this point). Linearizability ensures that every concurrent execution of the implementation data type is equivalent to some sequential execution of the abstract data type, where this 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 [Derrick et al. 2011a]:

$$Event = \text{inv}(P \times O \times V) \mid \text{ret}(P \times O \times V)$$

For brevity, we use notation $op_p^i(x)$ and $op_p^r: r$ for events $\text{inv}(p, op, x)$ and $\text{ret}(p, op, r)$, respectively, and use $op_p^i$ and $op_p^r$ to respectively denote invoke and return events with no inputs or outputs. For an event $e = (p, op, x)$, 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, $(a, b, c)$ 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 $\text{History} \equiv \text{seq}(Event)$, 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.

$$h_1 \equiv \langle \text{push}_1^1(a), \text{push}_2^2(b), \text{push}_3^3 \rangle$$

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.

$$hs_1 \equiv \langle \text{push}_1^1(a), \text{push}_3^3, \text{push}_2^2(b) \rangle \quad hs_2 \equiv \langle \text{push}_2^2(b), \text{push}_3^3, \text{push}_1^1(a) \rangle$$

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:

$$h_2 \equiv h_1 \bowtie (\text{pop}_3^3, \text{pop}_3^r: b)$$

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 $\text{pop}_3^3$ occurs after the return of both pushes, 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.,

$$hs_3 \equiv hs_2 \bowtie (\text{pop}_3^r, \text{pop}_3^r: b)$$

which results in an abstract stack $\langle a \rangle$ at the end of execution. Note that $hs_4 \bowtie (\text{pop}_3^r, \text{pop}_3^r: b)$ is an invalid linearization of $h_2$. Now suppose $h_2$ is appended with two more pop operations as follows:

$$h_3 \equiv h_2 \bowtie (\text{pop}_4^3, \text{pop}_4^r: a, \text{pop}_5^r: a)$$

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 \( H \in \text{History} \), 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 \( opq_p(x) \) iff \( opi = opq \) 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 \( \text{complete}(h) \) 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 2.1 (Linearizability [Herlihy and Wing 1990]). 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 \( \text{complete}(HC') \) 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.

\[
\langle \text{push}_p^l(x), \text{pop}_q^l, \text{pop}_q^R(x) \rangle
\]

The effect of operation \( \text{push}_p^l(x) \) has clearly occurred in (1) because the \( \text{pop}_q \) returns \( x \), i.e., the linearization point of \( \text{push}_p^l(x) \) occurs before that of \( \text{pop}_q^l \). More formally (1) is linearizable because it can be extended with a matching response to \( \text{push}_p^l(x) \), then linearized by the following history.

\[
\langle \text{push}_p^l(x), \text{push}_p^R, \text{pop}_q^l, \text{pop}_q^R(x) \rangle
\]

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 2.1 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 [Derrick et al. 2011a; 2011b; Schellhorn et al. 2012]. 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 \( \text{is_inv}(e) \) 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 \( \text{h}(m) \) and \( \text{h}(n) \) belong to the same operation call and there are no other invocations by \( \text{proc}(\text{h}(m)) \) between \( m \) and \( n \). An index \( m \) is a pending invocation in \( h \) iff \( \text{h}(m) \) is an invocation and there are no other events by process \( \text{proc}(\text{h}(m)) \) after index \( m \). A history \( h \) is legal iff for each index \( n \), if \( \text{h}(n) \) is an invoke event then either \( \text{h}(n) \) is pending or \( n \) forms a matching pair with an index \( m > n \), otherwise (i.e., if \( \text{h}(n) \) 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 2.2 (Legal history). Let \( h: \text{seq}(\text{Event}) \) be a sequence of events. Two positions \( m, n \in \text{dom}(h) \) form a matching pair, denoted \( mp(m, n, h) \) iff

\[
m < n \land \text{proc}(\text{h}(m)) = \text{proc}(\text{h}(n)) \land \text{oper}(\text{h}(m)) = \text{oper}(\text{h}(n)) \land \\
\forall k \cdot m < k < n \Rightarrow \text{proc}(\text{h}(k)) \neq \text{proc}(\text{h}(m))
\]

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

\[
is\text{_inv}(\text{h}(m)) \land \forall n: \text{dom}(h) \land m < n \Rightarrow \text{proc}(\text{h}(m)) \neq \text{proc}(\text{h}(n))
\]
We say \( h \) is legal, denoted \( \text{legal}(h) \), if
\[
\forall n: \text{dom}(h) \land (\text{if } \text{is_inv}(h(n)) \land \text{pi}(n, h) \lor \exists m: \text{dom}(h) \land \text{mp}(m, n, h)) \Rightarrow \exists m: \text{dom}(h) \land \text{mp}(m, n, h)
\]

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 \( h_0 \in \text{seq}(\text{Ret}) \) 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 \( h \triangle h_0 \) 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, \( \text{complete}(h \triangle h_0) \) and \( hs \) have to be equivalent, and
\( L2. \) the ordering of operation executions in \( h \) needs to be preserved in \( hs \).

Here, two operations are ordered if the second one starts after the first one has returned.

\textbf{Definition 2.3.} Given two histories \( h, hs \), we define \( h \) to be in \( \text{lin}-\text{relation with } hs \), denoted \( \text{lin}(h, hs) \), iff
\[
\exists f: \text{dom}(h) \Rightarrow \text{dom}(hs) \land \forall n: \text{dom}(h) \land h(n) = hs(f(n)) \\
\land \forall m, n: \text{dom}(h) \land m < n \land \text{mp}(m, n, h) \Rightarrow f(n) = f(m) + 1 \\
\land \forall m, n, m', n': \text{dom}(h) \land n < m' \land \text{mp}(m, n, h) \land \text{mp}(m', n', h) \\
\Rightarrow f(n) < f(m')
\]

\textbf{Definition 2.4 (Linearizable histories [Derrick et al. 2011a]).} A concurrent history \( h \) is \textit{linearizable} with respect to some sequential history \( hs \) iff there exists some \( h_0 \in \text{seq}(\text{Ret}) \) such that \( \text{linearizable}(h, h_0, hs) \) holds, where
\[
\text{linearizable}(h, h_0, hs) \triangleq \text{legal}(h \triangle h_0) \land \text{lin}(\text{complete}(h \triangle h_0), hs)
\]

A concrete implementation \( C \) is \textit{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 \( f_1 \triangleq \{0 \mapsto 0, 1 \mapsto 2, 2 \mapsto 1, 3 \mapsto 3\} \), \( h_1 \) is linearizable with respect to \( hs_2 \) using mapping \( f_2 \triangleq \{0 \mapsto 2, 1 \mapsto 0, 2 \mapsto 3, 3 \mapsto 1\} \), \( h_3 \) 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.

\textbf{2.4. Difficulties in verifying linearizability via linearization points}

When verifying linearizability, one must first identify the \textit{linearization points} in the implementation code and prove correspondence between execution of the linearization points and the execution of a \textit{sequential specification}. Typically, this correspondence relation is \textit{refinement} [de Roever and Engelhardt 1996], and using refinement terminology, we refer to the implementation as the \textit{concrete system} and the sequential specification as the \textit{abstract system}. To verify refinement, one must identify and establish appropriate invariants to prove the \textit{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 I. Of course, there exist several other algorithms than have not yet been formally verified correct, and hence, the list of algorithms in Table I 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., [Travkin et al. 2012]).

\footnote{Original version contained an error, which was corrected and verified by [Colvin and Groves 2005].}
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 [Schellhorn et al. 2012], i.e., provided an algorithm is 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 `Head = null` holds when P2 is executed — at this point, if `Head = null` holds, one can be guaranteed that the `pop` operation will return `empty` and in addition that the corresponding abstract stack is empty.

Proving correctness of such algorithms is relatively straightforward, because reasoning may be performed in a forward manner, 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 [Vafeiadis 2010].

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 a concrete system potentially corre-
sponds to several possible states of the abstract data type for each of the possible future outcomes. Hence, for each step of the concrete, one must check that each potential abstract data type is correctly transformed. In order to refer to the future behaviour of the operations, such algorithms can only be verified using backward simulation, or (the equivalently expressive) prophecy variables. These constitute the most difficult class of algorithms that have been verified correct. Recently, aspect-oriented approaches to verification have been developed [Henzinger et al. 2013b], which has been used to verify Herlihy/Wing’s queue, however, it is currently unclear whether every data structure can be completely categorised by its aspects.

3. VERIFYING LINEARIZABILITY

Verifying linearizability by considering Definition 2.1 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. For practical purposes, 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 [Colvin et al. 2005], 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., $H_1_p, H_2_p, H_1_q$) 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 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 push operations are linearized by the statement labelled H6, and 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 ⟨Push(a), Push(b), Pop: b⟩, 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 II 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. Section 3.1 covers refinement-based methods, and Section 3.2 describes methods that augment the state space. Compositional approaches to linearizability verification is described in Section 3.3, proofs that allow linearizability to be proved without considering linearization points are given in Section 3.4 and construction-based approaches to verifying linearizability is given in Section 3.5.

### 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 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 be-

### Table II. Methods for verifying linearizability

<table>
<thead>
<tr>
<th>Method</th>
<th>Algorithms verified</th>
<th>Reference</th>
</tr>
</thead>
</table>
| Simulation with canonical abstraction<sup>5</sup> | Treiber stack, MS queue, Array-based queue, Lazy set, Elimination stack, Snark double-ended queue | [Groves 2009]  
<sup>6</sup>Including a variation in [Doherty et al. 2004b].  
<sup>7</sup>See algorithm in [Michael 2002], which is based on [Harris 2001].  
<sup>8</sup>Rely-guarantee Interval Temporal Logic  
<sup>9</sup>The use of atomicity brackets prohibits behaviours that are permitted by the full fine-grained. | [Doherty et al. 2004a]  
<sup>5</sup>This is the only method known to have found two bugs in existing algorithms [Doherty 2003; Colvin and Groves 2005]. |
| Data refinement with sequential abstraction | Treiber stack, Lazy set, HW queue | [Derrick et al. 2011a]  
<sup>7</sup>Set algorithm in [Michael 2002], which is based on [Harris 2001].  
<sup>8</sup>Rely-guarantee Interval Temporal Logic  
<sup>9</sup>The use of atomicity brackets prohibits behaviours that are permitted by the full fine-grained. | [Derrick et al. 2011b]  
<sup>6</sup>Including a variation in [Doherty et al. 2004b].  
<sup>5</sup>This is the only method known to have found two bugs in existing algorithms [Doherty 2003; Colvin and Groves 2005]. |
| Rely-guarantee with separation logic | Treiber stack, Pessimistic set, Lazy set, HW queue, Optimistic set, Two-lock queue, MS queue<sup>6</sup>, HM lock-free set<sup>7</sup> | [Vafeiadis 2007]  
<sup>7</sup>Set algorithm in [Michael 2002], which is based on [Harris 2001].  
<sup>8</sup>Rely-guarantee Interval Temporal Logic  
<sup>9</sup>The use of atomicity brackets prohibits behaviours that are permitted by the full fine-grained. | [Liang and Feng 2013a]  
<sup>6</sup>Including a variation in [Doherty et al. 2004b].  
<sup>5</sup>This is the only method known to have found two bugs in existing algorithms [Doherty 2003; Colvin and Groves 2005]. |
| Reduction | Treiber stack, MS queue, Elimination stack | [Elmas et al. 2010; Groves 2009]  
<sup>7</sup>Set algorithm in [Michael 2002], which is based on [Harris 2001].  
<sup>8</sup>Rely-guarantee Interval Temporal Logic  
<sup>9</sup>The use of atomicity brackets prohibits behaviours that are permitted by the full fine-grained. | [Groves 2008b; Elmas et al. 2010]  
<sup>6</sup>Including a variation in [Doherty et al. 2004b].  
<sup>5</sup>This is the only method known to have found two bugs in existing algorithms [Doherty 2003; Colvin and Groves 2005]. |
| RGITL<sup>8</sup> | Treiber stack, MS queue<sup>6</sup>, Treiber stack with hazard pointers | [Baumler et al. 2011]  
<sup>7</sup>Set algorithm in [Michael 2002], which is based on [Harris 2001].  
<sup>8</sup>Rely-guarantee Interval Temporal Logic  
<sup>9</sup>The use of atomicity brackets prohibits behaviours that are permitted by the full fine-grained. | [Tofan et al. 2011]  
<sup>6</sup>Including a variation in [Doherty et al. 2004b].  
<sup>5</sup>This is the only method known to have found two bugs in existing algorithms [Doherty 2003; Colvin and Groves 2005]. |
| Shape analysis | Treiber stack, MS queue<sup>6</sup>, Numerous algorithms from [Herlihy and Shavit 2008] | [Amit et al. 2007]  
<sup>7</sup>Set algorithm in [Michael 2002], which is based on [Harris 2001].  
<sup>8</sup>Rely-guarantee Interval Temporal Logic  
<sup>9</sup>The use of atomicity brackets prohibits behaviours that are permitted by the full fine-grained. | [Vafeiadis 2010]  
<sup>6</sup>Including a variation in [Doherty et al. 2004b].  
<sup>5</sup>This is the only method known to have found two bugs in existing algorithms [Doherty 2003; Colvin and Groves 2005]. |
| Construction-based | Treiber stack, MS queue, Elimination stack, Optimistic set | [Jonsson 2012]  
<sup>7</sup>Set algorithm in [Michael 2002], which is based on [Harris 2001].  
<sup>8</sup>Rely-guarantee Interval Temporal Logic  
<sup>9</sup>The use of atomicity brackets prohibits behaviours that are permitted by the full fine-grained. | [Abrial and Cansell 2005; Groves and Colvin 2009]  
<sup>6</sup>Including a variation in [Doherty et al. 2004b].  
<sup>5</sup>This is the only method known to have found two bugs in existing algorithms [Doherty 2003; Colvin and Groves 2005]. |
| Hindsight lemma | Optimistic set<sup>5</sup>, Lazy set<sup>9</sup> | [O’Hearn et al. 2010a; 2010b]  
<sup>7</sup>Set algorithm in [Michael 2002], which is based on [Harris 2001].  
<sup>8</sup>Rely-guarantee Interval Temporal Logic  
<sup>9</sup>The use of atomicity brackets prohibits behaviours that are permitted by the full fine-grained. | [Vechev and Yahav 2008]  
<sup>6</sup>Including a variation in [Doherty et al. 2004b].  
<sup>5</sup>This is the only method known to have found two bugs in existing algorithms [Doherty 2003; Colvin and Groves 2005]. |
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 [de Roever and Engelhardt 1996]. 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 [Colvin et al. 2006a; Colvin and Groves 2005; Colvin et al. 2005; Doherty et al. 2004b; Doherty 2003], using the framework of Input/Output Automata [Lynch and Tuttle 1989; Lynch 1996]. 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 [de Roever and Engelhardt 1996].

The abstract data structures in Groves et al.’s proofs [Colvin et al. 2006a; Colvin and Groves 2005; Colvin et al. 2005; Doherty et al. 2004b; Doherty 2003] are canonical constructions as described by Lynch [Lynch 1996], 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 statement. The $DoPush$ 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 [Lynch 1996]. This guarantees linearizability of the implementation because every trace of an implementation is also a trace of the canonical construction. The methods

ACM Computing Surveys, Vol. V, No. N, Article A, Publication date: January YYYY.
in [Colvin et al. 2006a; Colvin and Groves 2005; Colvin et al. 2005; Doherty et al. 2004b; Doherty 2003] all rely on this fundamental result [Doherty 2003].

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 $H6$ and $P7$ 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 $H6_p$ 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. [Schellhorn et al. 2012; Derrick et al. 2011b; 2007; 2011a] 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. In addition, a number of 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. [Derrick et al. 2011a], 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 $push_h(a)$ updates $hc$ to $hc \vdash \langle push_h(a) \rangle$. Every abstract transition updates the $ha$ with matching invocation and response pairs, e.g., $APush_p$ updates the $ha$ to
Fig. 5. Derrick et al.’s refinement proofs for linearizability.

$ha \sim \langle \text{push}_a^b(a), \text{push}_b^a \rangle$. 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.

These methods have been used to verify a number of algorithms including the lazy set [Heller et al. 2007; Derrick et al. 2011b] and the Herlihy/Wing queue [Herlihy and Wing 1990; Schellhorn et al. 2012], both of which have complex linearization points. Furthermore, their work includes a completeness result that shows backwards simulation for history enhanced data types alone is enough to verify linearizability via data refinement [Schellhorn et al. 2012]. This is a stronger result than the result for canonical automata [Lynch 1996], which requires a combination of forwards and backwards simulation to prove completeness of simulation to verify data refinement.

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 [Vafeiadis 2007; Vafeiadis et al. 2006; Liang and Feng 2013b; 2013a]. 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 $\langle v \rangle \dashv Stack$ whenever the CAS is successful [Vafeiadis 2007]. 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 [O’Hearn et al. 2001] and rely-guarantee [Jones 1983] over the unified layer. In particular, Vafeiadis et al.’s approach [Vafeiadis 2007; Vafeiadis et al. 2006] use a framework that combines rely-guarantee for compositionality and separation logic for pointer-based reasoning [Vafeiadis and Parkinson 2007]. Liang and Feng [Liang and Feng 2013b; 2013a] develop a related framework, which is used to prove a number of algorithms from the literature.

Applying their techniques 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 in fact exploited in the forward simulation proof in [Colvin et al. 2006a], where auxiliary variables that encode the abstract state are introduced to 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. For example, the proof of an operation with non-fixed linearization is left informal in [Vafeiadis et al. 2006], despite the fact that the framework allows (two-state) rely-guarantee predicates [Jones 1983].

Vafeiadis’ thesis [Vafeiadis 2007] 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 [Abadi and Lamport 1991] are used to reason about operations whose linearization points depend on future behaviour. Recently, Liang and Feng [Liang and Feng 2013b] have consolidated these ideas into a framework based on separation logic. Their augmentations include \texttt{linself}, (which performs the same function as the augmentations of Vafeiadis by linearizing the currently executing process [Vafeiadis et al. 2006]) and \texttt{lin(p)}, (which performs the linearization of process \texttt{p} different from \texttt{self} that may be executing a different operation [Vafeiadis 2007]). Unlike Vafeiadis, they also allow augmentations that use \texttt{try} and \texttt{commit} pairs, where the \texttt{try} is used to guess potential linearization points, and the \texttt{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 [Jones and Muchnick 1982], 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. [Amit et al. 2007], who consider implementations using singly linked lists and fixed linearization points. The following paraphrases Amit et al. [Amit et al. 2007, pg 480], by clarifying their nomenclature with the terminology used in this paper.
The proof method uses a correlating semantics, which simultaneously manipulates two memory states: a so-called candidate state [i.e., 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 [Vafeiadis 2009], 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 methods is used to verify several algorithms including the RDCSS algorithm, which has external linearization points.

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 [Vafeiadis 2010], 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 [de Roever et al. 2001], 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 [Jones 1983], where a rely condition, states assumptions about a component’s environment, and a guarantee condition describes the behaviour a component under the assumption that the rely condition holds. A detailed overview of compositional verification techniques lies outside the scope of this paper; we refer the interested reader to [de Roever et al. 2001; Vafeiadis 2007].

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 [Moszkowski 2000], which enables one to reason over the interval of time in which a program executes, as opposed to single state transition [Schellhorn et al. 2011]. The proofs are carried out using the KIV theorem prover [Drexler et al. 1993], which is combined with symbolic execution [Burstall 1974; Bäumler et al. 2010] to enable guarantee conditions to be checked 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 [Bäumler et al. 2011]. These methods have been applied to verify the Treiber stack and the Michael/Scott queue [Bäumler et al. 2011]. Our own methods [Dongol and Derrick 2013] 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 of the methods re-
lies on the correct identification of linearization points. For many interesting (complex) algorithms, the location of such linearization points is unclear, and may depend on the future behaviour of an algorithm. 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.

3.4.1. Reduction. Reduction [Lipton 1975] 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 [Groves 2008b; 2007; Elmas et al. 2010], 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 [Wang and Stoller 2005] 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 [Groves 2008b; 2007] and separately by Elmas et al. [Elmas et al. 2010]. The methods of include a tool support [Elmas 2010] via their QED verifier.

3.4.2. 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 [O’Hearn et al. 2010a], 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 [O’Hearn et al. 2010a] only considers a simple optimistic set. The extended technical report [O’Hearn et al. 2010b] 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 [Henzinger et al. 2013b]. Automation has been achieved for algorithms with helping mechanisms and internal linearization points such as the elimination stack [Dragoi et al. 2013]. These techniques require the algorithms to satisfy so-called R-linearizability [Pacull and Sandoz 1993], a stronger condition than linearizability, and hence, verification of algorithms with linearization points based on future behaviour are excluded.

3.4.3. 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 [Moszkowski 2000; 1997]), to reason about linearizability [Bäumler et al. 2011; Schellhorn et al. 2011; Bäumler et al. 2010; Dongol and Derrick 2013; 2012]. Of these, our own methods [Dongol and Derrick 2013; 2012] verify behaviour refinement between a coarse-grained abstraction and fine-grained implementation. The basic motivation resembles the reduction-based approaches
(Section 3.4.1) and hence, unlike [Bäumler et al. 2011; Schellhorn et al. 2011; Bäumler et al. 2010],
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 implemen-
tation 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 [Abrial and Cansell 2005],
within the Event-B framework [Abrial 2010], 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
[Michael and Scott 1996]), and it is not clear whether such a scheme really is implementable.

Groves [Groves 2008a] presents a derivation of the Michael/Scott queue using trace reduction to justify each refinement step [Lipton 1975]. In particular, atomicity decomposition is justified if the new actions commute with those in concurrent processes. This is extended by Groves and Colvin
[Groves and Colvin 2009; 2007], where the derivation of a more complicated stack by Hendler et al. [Hendler et al. 2010] 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) [Groves and Colvin 2009; 2007]. These proofs are not yet mechanised, and the effort required to mechanically perform these reduction-based proofs is unclear.

Gao and Hesselink et al. also present a number of derivations of non-blocking algorithms [Gao et al. 2009; Gao and Hesselink 2007; Gao et al. 2007; 2005], including the development of special-purpose reduction theorems [Gao and Hesselink 2007]. However, these derivations are focused on the preservation of the lock-free progress property [Massalin and Pu 1992; Dongol 2006], as opposed to linearizability.

Vechev and Yahav [Vechev and Yahav 2008; Vechev et al. 2009] 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 [Jonsson 2012] has presented a derivation of the Treiber stack and Michael/Scott queue in a refinement calculus framework [Morgan and Vickers 1992]. Jonsson defines linearizability using the following statement [Jonsson 2012, Definition 3.1]:

A program $P$ is linearizable if and only if $\text{atomic}\{P\}$ is refined by $P$.

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 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 [Turon and Wand 2011], 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. [Heller et al. 2007], which is a simple algorithm with add and remove operations that have fixed linearization points and a contains operation that is potentially linearized by the execution of other operations. We first present a verification of a simplified version with only an add and remove operation. An overview of the different approaches to verifying set algorithms is given in Table III. 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.

An early proof attempt of the lazy set using auxiliary variables is that of Vafeiadis et al. [Vafeiadis et al. 2006], who extend the concrete state space with additional variables corresponding to the abstraction of the data structure in question, then augment linearization points of the concrete code with statements corresponding to the abstract operation. The methods in [Vafeiadis et al. 2006] are unable to verify the contains operation. This is addressed in Vafeiadis’ thesis [Vafeiadis 2007] where augmentations are able to reason about the linearization points of other system processes, enabling a proof of the lazy set algorithm.

The first complete proof is that of Colvin et al. [Colvin et al. 2006a], who show that the concrete implementation simulates an abstract canonical algorithm. Their proofs use the framework of Input/Output Automata [Lynch and Tuttle 1989] and are mechanised in the PVS theorem prover [Owre et al. 1996]. Their setup allows invariants to be checked in a model checker, providing some level of assurance that the final proof will be successful. Verification of contains requires the use of backwards simulation [de Roever and Engelhardt 1996] because the proof must refer to future behaviour of the operation.

Vafeiadis presents a tool-supported method that allows linearization points to be automatically inferred [Vafeiadis 2010]. Although a proof of the lazy set is not given (due to difficulties in automatically inferring correct abstractions), the method is able to verify other set algorithms that requires reference to linearization points outside the operation. The clear advantage of this method is that when an algorithm is within the scope of CAVE [Vafeiadis 2010], no input additional input from the verifier is needed.

O’Hearn et al. present a methodology specific to sets that are implemented concretely by linked lists is given by [O’Hearn et al. 2010a; 2010b], however, simplifications made about the atomicity of locked sections of code mean that (contrary to the claims in the paper), the algorithms verified are in fact not the optimistic nor lazy sets. In particular, by making locked sections of code atomic, the algorithms in [O’Hearn et al. 2010a; 2010b] do not allow traversal of the locked sections while the actual addition/removal is being performed.

Elmas et al. present a method based on reduction [Lipton 1975] that enables incremental replacement of fine-grained code in an implementation by statements with coarser-grained atomicity. Their methods use the QED verifier [Elmas 2010] and are used to prove a multi-set algorithm consisting of add and remove operations, but without a contains operation [Elmas et al. 2009]. Reduction is aimed at simplifying linearizability proofs and reducing the complexity of an implementation, and therefore do not need to identify the linearization points in the concrete code. Note that the multiset algorithm is also used as a case study by Travkin et al. [Travkin et al. 2012], but here, the only operations considered are add and contains. Linearizability could not be proved with the inclusion of a delete operation. Using the framework of Rely-Guarantee Interval Temporal Logic [Schellhorn et al. 2011], which allows compositional reasoning. An advantage of their framework is that it enables symbolic execution of a algorithm and its properties within the KIV theorem prover [Drexler et al. 1993].

Derrick et al. return to refinement-based proofs [Derrick et al. 2011b], presenting a verification of the lazy set. Non-atomic refinement [Derrick and Wehrheim 2005] and potential linearization points are encode future behaviours of the contains operation and the proof have been mechanised in KIV. Their techniques build on [Derrick et al. 2011a] and use history enhanced data types,
Table III. Overview of methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Lin. point identification</th>
<th>Tool</th>
<th>Complete</th>
<th>Linked to HW defn</th>
<th>Additional notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>[Vafeiadis et al. 2006]</td>
<td>Manual</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Operation contains not verified</td>
</tr>
<tr>
<td>[Colvin et al. 2006a]</td>
<td>Manual</td>
<td>PVS</td>
<td>Yes(^{10})</td>
<td>No</td>
<td>Allows model checking</td>
</tr>
<tr>
<td>[Vafeiadis 2007]</td>
<td>Manual</td>
<td>—</td>
<td>Yes(^{11})</td>
<td>No</td>
<td>Auxiliary code can</td>
</tr>
<tr>
<td>[Vafeiadis 2010](^{12})</td>
<td>Automatic</td>
<td>CAVE</td>
<td>—</td>
<td>Yes</td>
<td>Full automation via shape analysis</td>
</tr>
<tr>
<td>[O’Hearn et al. 2010a](^{12})</td>
<td>N/A(^{13})</td>
<td>—</td>
<td>No</td>
<td>No</td>
<td>Only applicable to list-based sets</td>
</tr>
<tr>
<td>[Elmas et al. 2009](^{12})</td>
<td>N/A(^{14})</td>
<td>QED</td>
<td>—</td>
<td>Yes</td>
<td>Technique based on reduction</td>
</tr>
<tr>
<td>[Derrick et al. 2011b]</td>
<td>Manual</td>
<td>KIV</td>
<td>Yes(^{15})</td>
<td>Yes</td>
<td>Data refinement-based proofs</td>
</tr>
<tr>
<td>[Liang and Feng 2013a]</td>
<td>Manual</td>
<td>—</td>
<td>—</td>
<td>Yes(^{12})</td>
<td>Separation logic encoding</td>
</tr>
<tr>
<td>[Dongol and Derrick 2013]</td>
<td>N/A(^{14})</td>
<td>—</td>
<td>—</td>
<td>Yes(^{17})</td>
<td>Interval-based reasoning</td>
</tr>
</tbody>
</table>

and hence, are the first to link data refinement of implementations to Herlihy and Wing’s original definition of linearizability [Herlihy and Wing 1990].

Liang and Feng [Liang and Feng 2013b; 2013a] present an approach that extends the original ideas of Vafeiadis et al. [Vafeiadis et al. 2006], where linearization points of the algorithm in question are augmented with auxiliary statements that describe behaviour at the linearization points.

The final approach we consider is our own method, which uses a framework based on interval predicates [Dongol and Derrick 2013]. Here, we consider the behaviour of the algorithm in question over its interval of execution, then show that this behaviour implies the behaviour of a coarse-grained algorithm over the same interval. Like the reduction-based methods, our approach is aimed at simplifying proofs of linearizability by transforming the fine-grained algorithm to another algorithm with coarser-grained atomicity, and hence, requires a second proof that the coarse-grained algorithm is itself linearizable.

The optimistic set algorithm is presented in Section 4.1, its linearization points are discussed in Section 4.2, and three different proofs of linearizability are presented in Sections 4.3, 4.4, and 4.5. The formalisation in this section are aimed at highlighting the main ideas behind each method to elucidate their differences. We refer readers interested in reproducing each proof accurately to the original papers.

4.1. An optimistic set

In this section, we present a simplified version of Heller et al.’s concurrent set algorithm [Heller et al. 2007] (see 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

\(^{10}\)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 [Heller et al. 2007] 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 [Schellhorn et al. 2012].
\(^{16}\)Link to linearizability is via results of Filipović et al. [Filipović et al. 2010].
\(^{17}\)Using history-enhanced data types [Dongol and Derrick 2012].
\(^{18}\)The mark bit is not strictly necessary to implement the optimistic set (e.g., [Vafeiadis 2007]), however, we use it here to simplify the lead up to the lazy set in Section 5.
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
endif
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;
endif
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 := curr.next;
L6: pred.lock();
L7: curr.lock();
L8: if !pred.mark and !curr.mark and pred.next = curr
L9: return pred, curr
L10: pred.unlock();
L11: curr.unlock();
enddo

Fig. 7. Lazy set algorithm operations

Fig. 8. Execution of add(42) by process p

As a concrete example, consider the linked list in Figure 8 (a), which represents the set \{3, 18, 77\}, and an execution add(42) by process p without interference. Execution starts by calling locate(42), which searches for the successor (curr) of the node to
be added, ignoring any other locks (lines L3-L5). After exiting the loop, the executing process locks both $\text{pred}$ and $\text{curr}$, which prevents their modification by other processes (lines L6-L7). At L8, the process checks to ensure that both $\text{pred}$ and $\text{curr}$ are unmarked (which ensures that they have not been removed since the end of the loop) and that $\text{pred.next}$ is still $\text{curr}$ (which ensures that no new nodes have been inserted between $\text{pred}$ and $\text{curr}$ since the end of the loop). If the test at L8 fails, the locks on $\text{pred}$ and $\text{curr}$ are released and the process $\text{locate}$ is re-executed. In our example execution, we assume that the test is successful, which causes $n_1$ and $n_2$ 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 $n_3$ with value 42 and next pointer $n_3_p$ (see Figure 8 (c)). Then by executing A4, the executing process sets the next pointer of $n_1$ to $n_2_p$ causing a successful add operation (that returns $\text{true}$) 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 $\text{false}$. Following the linearization, process $p$ releases the locks on $n_1$ and $n_3$ and returns $\text{true}$ 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 $\text{true}$ (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 $\text{false}$. 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 $\text{true}$ 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.

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.

We present outlines of the proofs using the simulation-based methods of Colvin et al. [Colvin et al. 2006a] (Section 4.3), refinement-based method of Derrick et al. [Derrick et al. 2011b] (Section 4.4) and auxiliary variable method of Vafeiadis’ [Vafeiadis 2007] (Section 4.5). The other meth-
ods described in this paper only conjecture the possibility of a proof, and hence, we do not attempt to develop a proof ourselves. 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 [Colvin et al. 2006b] and the Vafeiadis’ RGSe notation [Vafeiadis and Parkinson 2007; Vafeiadis 2007] into Z [Bowen 1996], which is the notation used by Derrick et al. Inevitably, this causes some of the benefits of a proof method to be lost; we discuss the effect of the translation and the benefits provided by the original framework, where necessary.

Details on representing concurrent algorithms in Z are given in [Derrick et al. 2011a]. 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 $p \in P$.

Using the Object-Z [Smith 1999] convention, we assume that variables $v' = v$ for every variable $v$ unless $v' = k$ is explicitly defined for some 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 (in which it is not executing any operation) and an active state (in which it has invoked an operation but has not yet returned). A process in an idle state may become active by non-deterministically invoking one of the set operations with a non-deterministically chosen parameter. After completion of the operation, the process returns to an idle state.

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:

- $A2$ if the test fails, i.e., $n3.val = x$ holds, and
- $A7$, 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

```
addA(x):
    ...
    A7a: local_stmt_a;
    A7: res := false;
    A7b: local_stmt_b;
    ...
```

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 $x$. 

A:21
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 response operations 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.

\[
\text{AState}
\]

\[
S: \mathbb{P} \times V
\]

\[
\text{pc: } P \rightarrow \text{Label}
\]

\[
\text{v: } P \rightarrow V
\]

The canonical operations corresponding to the add are given by the following Z schema:

\[
\begin{array}{c|c|c|c}
\text{AddInv}_p(x) & \text{AddOK}_p & \text{AddFail}_p & \text{AddRes}_p(r) \\
\hline
\Delta \text{AState} & \Delta \text{AState} & \Delta \text{AState} & \Delta \text{AState} \\
pc(p) = \text{id} & pc(p) = \text{addi} & pc(p) = addi & pc(p) = addlo \\
p'(p) = \text{addi} & \text{v(p)} \in S & \text{v(p)} \notin S & r = \text{res}(p) \\
v'(p) = x & S' = S \cup \{v(p)\} & \text{res}'(p) = \text{false} & p'(p) = \text{id} \\
\text{pc}'(p) = \text{addo} & \text{v}'(p) = \text{true} & \text{pc}'(p) = \text{addo} &
\end{array}
\]

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

As highlighted in Section 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 AOOp is the abstract operation corresponding to the COOp in process p, and ‘‘ denotes relational composition.

\[
\forall p: P \cdot \text{rep} \sqsubseteq \text{COOp} \subseteq \text{AOOp} \sqsubseteq \text{rep}
\]

Thus, for any abstract state \( \sigma \) and concrete state \( \tau \) linked by the representation relation \( \text{rep} \), if the concrete statement \( \text{COOp} \) is able to transition from \( \tau \) to \( \tau' \), then there must exist an abstract state \( \sigma' \) such that \( \text{AOOp} \) 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 auxS, which stores the set of elements currently in the set. The set auxS is updated whenever a node is inserted into the list, or is marked.
for deletion. Correctness of this encoding is verified by proving

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

where \( c \) 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\(^{19} \) [Colvin et al. 2006a; 2006b].

\[
\forall p: P \cdot pc(p) \in \{ A5, R7 \} \Rightarrow v(p) \notin aux_S \tag{2}
\]

\[
\forall p: P \cdot pc(p) \in \{ A7, R3 \} \Rightarrow v(p) \in aux_S \tag{3}
\]

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 \( cs \).

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

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 [Owre et al. 1996].

4.4. Method 2: Proofs against sequential specifications

Derrick et al. present a proof of the lazy set algorithm [Derrick et al. 2011b], 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.
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:

\[
\begin{align*}
\text{Add}_p & \quad \underline{\Delta AState} \\
x: V & \quad r: B \\
S' = S \cup \{x\} & \quad r = (S' \neq S)
\end{align*}
\]

\[
\begin{align*}
\text{Remove}_p & \quad \underline{\Delta AState} \\
x: V & \quad r: B \\
S' = S \setminus \{x\} & \quad r = (S' \neq S)
\end{align*}
\]

The proofs rely on \textit{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

\(^{19}\)In [Colvin et al. 2006b] \( A5 \) and \( A7 \) are labelled \( add_6 \) and \( add_8 \), respectively.
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:

\[
\text{AddH}_p(x, r) \equiv \text{Add}_p(x, r) \land \\
[hs, hs': \text{seq}(\text{EVENT}) \mid hs' = hs \land \langle \text{inv}(p, \text{add}, x), \text{ret}(p, \text{add}, r) \rangle]
\]

\[
\text{RemoveH}_p(x, r) \equiv \text{Remove}_p(x, r) \land \\
[hs, hs': \text{seq}(\text{EVENT}) \mid hs' = hs \land \langle \text{inv}(p, \text{rem}, x), \text{ret}(p, \text{rem}, r) \rangle]
\]

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

\[
\text{AddInvH}_p(x) \equiv \text{AddInv}_p(x) \land [h, h': \text{seq}(\text{EVENT}) \mid h' = h \land \langle \text{inv}(p, \text{add}, x) \rangle]
\]

\[
\text{AddRetH}_p(r) \equiv \text{AddRet}_p(r) \land [h, h': \text{seq}(\text{EVENT}) \mid h' = h \land \langle \text{ret}(p, \text{add}, r) \rangle]
\]

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 [Derrick and Wehrheim 2003; 2005] 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 2.4. 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 \( \text{AOOp}_p \). 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:

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

A process \( p \) has status \( \text{IDLE} \) if \( p \) is not executing any operation, \( \text{IN} \langle x \rangle \) if \( p \) is executing an operation with input \( x \), but has not passed the linearization point of the operation, and \( \text{OUT}(r) \) if \( p \) is executing an operation and has passed the linearization point with return value \( r \). This is combined with functions:

\[
\begin{align*}
\text{runs}: & \text{CState} \times P \to O \cup \{ \text{none} \} \\
\text{status}: & \text{CState} \times P \to \text{STATUS}
\end{align*}
\]

where \( \text{runs} \) denotes the operation the given process is executing in a given state (\( \text{none} \) if the process is idle) and \( \text{status} \) determines whether or not the process contributes a return event in a given state. The encoding of the \( \text{status} \) is such that

\[
\text{status}(cs, p) = \begin{cases} 
\text{IDLE} & \text{if runs}(cs, p) = \text{none} \\
\text{IN}(x) & \text{if runs}(cs, p) = \text{op} \land \text{cs}(pc(p)) \text{ is not past the lin. point of op} \\
\text{OUT}(r) & \text{if runs}(cs, p) = \text{op} \land \text{cs}(pc(p)) \text{ is past the lin. point of op}
\end{cases}
\]

The forward simulation relation \( \text{rep} \) is then of the form:

\[
\text{rep}((hs, h), (cs, h)) \equiv \\
\text{ABS}(as, cs) \land \text{INV}(cs) \land (\forall n \cdot \text{pi}(n, h) \Rightarrow \text{runs}(cs, \text{proc}(h(n))) = \text{oper}(h(n))) \land (\forall p, x \cdot \text{status}(cs, p) = \text{IN}(x) \Rightarrow \exists n \cdot \text{pi}(n, h) \land h(n) = \text{inv}(p, \text{runs}(cs, p), x)
\]

ACM Computing Surveys, Vol. V, No. N, Article A, Publication date: January YYYY.
Thus (4) states that the abstraction $\text{ABS}$ and invariant $\text{INV}$ hold, and (5) states that if $b(n)$ is a pending invocation, then function $\text{run}$ is accurate. Conjunct (6) states that whenever process $p$’s status is $\text{IN}(x)$ for some $x$, there must exist an index $n \in \text{dom}(h)$ such that $b(n)$ is a pending invocation, and corresponds to an invocation that is executing $\text{run}(cs,p)$ with input $x$. Finally conjunct (7) relates $h$ to $hs$ using the set of processes with status $\text{OUT}$. It requires that there exist a set $R$ of events corresponding to processes that have executed a linearized statement, but not yet returned such that for any bijective sequence $h_0$ generated from $R$, linearizable$(h, h_0, hs)$ holds.

Finally, a number of process-local\footnote{Thread-local in the terminology of [Derrick et al. 2011a].} proof obligations that do not need to refer to $hs$ and $h$ are generated, and a theorem that ensures these satisfaction of the process-local properties that imply rep holds. These proof obligations use information from the status function to determine the correct simulation condition. For example, the proof obligation below is for steps of process $p$ that transition from a status $\text{IN} \langle \text{in} \rangle$, where $\text{CO}p_p$ potentially corresponds

\[
\forall as: \text{AS}, cs, cs': \text{CState}, p; P \bullet
\]

\[
\text{rep}(as, cs) \land \text{status}(cs, p) = \text{IN} \langle \text{in} \rangle \land \text{CO}p_p(cs, cs') \Rightarrow
\]

\[
\text{status}(cs', p) = \text{IN} \langle \text{in} \rangle \land \text{rep}(as, cs') \lor
\]

\[
(\exists as', \text{out} \bullet \text{A}Op_p(\text{in}, as, as', \text{out}) \land \text{status}(as', p) = \text{OUT} \langle \text{out} \rangle \land \text{rep}(as', cs'))
\]

Verification of invocation and response transitions are straightforward because the abstraction is not modified, and stuttering transitions are straightforward because the histories are not modified. The non-stuttering transitions linearize the abstract data structure. This is reflected in the status function, whose value changes from $\text{IN}(x)$ before the transition to $\text{OUT}(r)$ after the transition.

4.5. Method 3: Auxiliary variables

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

(1) Introduce auxiliary variables to the existing program, at least one of which is an abstraction of the data type in question. 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 the abstraction of each statement.

(4) Define and prove an invariant that links the abstract and concrete representations. This involves introducing appropriate annotations, and verification using the RG-Sep framework [Vafeiadis and Parkinson 2007].

In this paper, for comparison, we present a translation of the RG-Sep framework into $Z$.

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

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

\[
A5:\langle n1.\text{next} := n2; \text{AbsRes} := (x \notin S); S := (S \cup \{x\})\rangle
\]

\[
A7:\langle \text{res} := \text{false}; \text{AbsRes} := (x \notin S); S := (S \cup \{x\})\rangle
\]

\[
\ldots
\]
Note that for \( A7 \), the auxiliary code will set \( \text{AbsRes} \) to \( \text{false} \) (i.e., \( x \not\in S \)), and therefore the abstract set \( S \) remains unchanged. The \( \text{remove} \) 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 ‘\( \land \)’, which enables simpler comparison among the different methods. We discuss the differences that arise from this translation where needed.

\[
\begin{array}{ll}
\text{Add}_{p} & \Delta \text{AugState} \\
\text{n}: \text{Node} & \text{n1, n2, n3}: \text{Node} \\
\text{u, v}: \text{Val} & \text{u, v}: \text{V} \\
\begin{align}
(u < v < \text{val}(n3)) \land \text{lvn}(n1) = (p, u, n3) \\
\text{lock}(n3) = p \land \text{lvn}(n2) = (\emptyset, v, n3) \\
\text{next}(n1) = n2 \land S' = S \cup \{v\}
\end{align}
\end{array}
\]

\[
\begin{array}{ll}
\text{Lock}_{p} & \Delta \text{AugState} \\
n: \text{Node} & \text{n}, n1 \in \text{Node} \\
\text{v}: \text{V} & \text{lvn}(n1) = (p, v, n) \\
\text{lvnm}(n1) = (p, v, n, \text{true}) \\
S' = S \setminus \{v\}
\end{array}
\]

\[
\begin{array}{ll}
\text{Unlock}_{q} & \Delta \text{AugState} \\
n: \text{Node} & \text{n1}: \text{Node} \\
\text{lvn}(n1) = (p, u, n2) \\
\text{lvnm}(n2) = (p, v, n3, \text{true}) \\
\text{lvnm}(n1) = (p, u, n3)
\end{array}
\]

\[
\begin{array}{ll}
\text{Remove}_{p} & \Delta \text{AugState} \\
n1, n2, n3: \text{Node} & \text{n1, n2, n3}: \text{Node} \\
u, v: \text{Val} & \text{lvn}(n1) = (p, u, n2) \\
\text{lvnm}(n2) = (p, v, n3, \text{true}) \\
\text{lvnm}(n1) = (p, u, n3)
\end{array}
\]

The rely condition for process \( p \) is

\[
\text{Rely}_{p} \equiv \bigvee_{q \in P(\{p\})} \text{Lock}_{q} \lor \text{Unlock}_{q} \lor \text{Add}_{q} \lor \text{Mark}_{q} \lor \text{Remove}_{q}
\]

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 [Vafeiadis 2007] require annotation of the code using separation logic-style assertions. In addition, building on the framework of [Jones 1983], these assertions must be stable with respect to the rely conditions. The proof outlines for the lazy set are elided in [Vafeiadis 2007], 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:

\[
\text{ls}(x, A, y) \equiv (x = y \land A = \langle \rangle) \lor \exists v, z, B: x \neq y \land A = \langle v \rangle \land B \land \text{val}(x) = v \land \text{next}(x) = z \land \text{ls}(z, B, y)
\]

\[
\text{sorted}(A) \equiv \begin{cases}
\text{true} & \text{if } A = \langle \rangle \lor A = \langle a \rangle \\
(a < b) \land \text{sorted}(\langle b \rangle \cap B) & \text{if } A = \langle a, b \rangle \cap B
\end{cases}
\]

\(^{21}\)[Vafeiadis 2007] 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.
\[ s(A) \equiv S = \text{ran}(A) \setminus \{-\infty, \infty\} \]

Vafeiadis uses notation \( \mathbb{P} \) 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 [Owicki and Gries 1976]). The majority of the assertions refer to the global state, and hence would have been written as \( \mathbb{P} \) anyhow. Note that due to a typographical error, the failed case of the add operation is missing in [Vafeiadis 2007], however, it can be reconstructed from the remove operation.

**add** \((x)\) :

\[
\exists u, v \cdot \exists n, A, B \cdot \text{ls}(\text{Head}, A, n_1) \land \text{lnv}(n_1) = (p, u, n_3) \\
\quad \land \text{lnv}(n_2) = (p, v, n) \land \text{ls}(n, B, \text{Tail}) \land s(A \cap \{u, v\} \cap B) \\
\quad \land \text{lnv}(n_2^p) = (\varnothing, x, n_3) \land u < x \land x < v
\]

\( A_5: \langle n1.next := n2; \text{AbsRes} := (x \notin S); S := (S \cup \{x\}) > \\
\exists u, v \cdot \exists n, A, B \cdot \text{ls}(\text{Head}, A, n_1) \land \text{lnv}(n_1) = (p, u, n_2) \\
\quad \land \text{lnv}(n_2) = (p, x, n) \land \text{ls}(n, B, \text{Tail}) \land s(A \cap \{u, x\} \cap B)
\]

\[
\ldots
\]

\( A_7: \langle \text{res} := \text{false}; \text{AbsRes} := (x \notin S); S := (S \cup \{x\}) > \\
\exists A \cdot \text{ls}(\text{Head}, A, \text{Tail}) \land s(A) \\
\ldots
\]

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 [Dongol 2009]. For example, given that \( pc(p) \) denotes the program counter for process \( p \), whose value is a program label, the assertion at \( A_7 \) can be encoded as a predicate of the form:

\[ POA_{7_p} \equiv pc(p) = A7 \Rightarrow \exists n, A, B \cdot \text{ls}(\text{Head}, A, n_3) \land \text{lnv}(n_3) = (p, x, n) \land \text{ls}(n, B, \text{Tail}) \land s(A \cap \{x\} \cap B) \]

Such proof obligations must be resilient to interference from other processes (as in [Owicki and Gries 1976]), and hence, one must verify that the following holds for each \( p, q \in P \) such that \( p \neq q \):

\[ POA_{7_p} \circ Env_q \Rightarrow POA_{7_p} \]

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

[Liang and Feng 2013a] provide outlines for the remove and contains operation 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.

### 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 [Vafeiadis 2010; Dragoi et al. 2013]). We have presented a detailed account of three methods that manually identifies the linearization points, as well as abstraction relations and invariants. These

---

Note that the optimistic set in [Vafeiadis 2010] does not use a marking scheme, and hence, is different from the algorithm in Section 4.1. Furthermore, the methods in [Dragoi et al. 2013] can only automatically verify algorithms with helping mechanisms.
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 refinement-based approaches (methods 1 and 2) require a two-state refinement relations, which are avoided in method 3, although method 1 simplifies the refinement relation via the introduction of auxiliary variables that reflect the abstract state at the concrete level. It is noteworthy that auxiliary variables can be introduced to method 2.

— The abstraction in method 1 is a canonical program that generates concurrent traces and differs from the abstraction in methods 2 and 3, where the abstract system 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.

— 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 [Heller et al. 2007], which consists of a contains operation in addition to the add and remove operations from the optimistic set from Section 4. Its verification using the three methods examined in 4 are presented in Sections 5.2, 5.3, and 5.4. A fourth method that uses an altogether different interval-based framework is presented in Section 5.5. Despite the simplicity of the contains operation, its verification introduces significant complexity in the proof methods, requiring the use of more advanced verification techniques.

5.1. The contains operation

In this section, we present Heller et al.’s concurrent set algorithm [Heller et al. 2007]. The algorithm consists of the add and remove operations from Figure 10 together with a contains operation, given in Figure 7. 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 resa as follows:
contains(x) :
C1: curr := Head;
C2: while curr.val < x do
   C3: curr := curr.next
   enddo;
C4: res := !curr.mark and (curr.val = x)
C5: return res

Fig. 10. The contains operation

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

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 follows\(^{23}\).

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

To verify linearizability of the original operation in Figure 10, 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 [Hayes et al. 2013; Dongol and Derrick 2013] (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, where process p is executing contains(50) and has just exited its loop because curr.val ≥ 50, but has not yet executed statement C4a.

![Diagram of linked list during execution](image)

Now suppose another process q executes an add(50) operation to completion. This results in the following state of the linked list, which corresponds to an abstract state \{3, 18, 50\}.

![Diagram of linked list after add(50)](image)

Execution of process p from this state will set resa\(_p\) to false, and hence the contains(50) will return false, even though the element 50 is in the set (corresponding to the shared linked list) when C4a is executed.

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

\(^{23}\)Of course, for a sequential program, C4a ; C4b is the same as C4c ; C4d.
linked list in Figure 9 (a), and execution has reached (but not yet executed) statement C4b. This results in the state of the shared linked list.

\[ \sim \rightarrow 3 \rightarrow 18 \rightarrow 77 \rightarrow \infty \]

Now suppose another process q executes a \texttt{remove(77)} operation to completion. This results in the following state, corresponding to the abstract queue \{3, 18\}.

\[ \sim \rightarrow 3 \rightarrow 18 \rightarrow \infty \]

Now, when process p executes C4b, it will set res\(_p\) to true, and hence, return true even though 77 is not in the abstract set corresponding to the shared linked list when C4b is executed. Therefore, neither C4a nor C4b are appropriate linearization point for contains.

An immediate question that now arises is whether contains is at all linearizable. The answer is yes. As Colvin \textit{et al.} point out:

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

That is, within any interval in which \texttt{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 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 add or a remove operation.

5.2. Method 1: Proofs against canonical automata

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

\[
\begin{align*}
\text{ContInv}_{p}(x) & : \Delta AState \\
\text{ContInv}_{p}(x) & : \Delta AState \\
\text{ContInv}_{p}(x) & : \Delta AState \\
\text{ContInv}_{p}(x) & : \Delta AState \\
\text{ContInv}_{p}(x) & : \Delta AState \\
\text{ContInv}_{p}(x) & : \Delta AState \\
\text{ContInv}_{p}(x) & : \Delta AState \\
\end{align*}
\]

Forward simulation cannot be used to prove that the \texttt{contains} operation implements the canonical abstraction (see [Colvin et al. 2006a] for details). Instead proofs are performed using \textit{backward simulation} [de Roever and Engelhardt 1996], which for a non-stuttering transition generates a proof obligation of the form:

\[ \forall p: P \cdot COp_p \not\subseteq rep \not\subseteq AOp_p \]

Thus, if \texttt{COp}_p can transition from \(\tau\) to \(\tau'\) and \(\tau'\) is related by \texttt{rep} to some abstract state \(\sigma'\), then there must exist an abstract state \(\sigma\) such that \texttt{rep} holds between \(\tau\) and \(\sigma\) and \texttt{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 \textit{et al.} split their simulation proofs by introducing an intermediate specification, that “eliminates the need to know the future [Colvin et al. 2006a, pg 481]”
They then prove backward simulation between the 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 \( \text{seen\textunderscore out}(p) \) that holds for a process \( p \) executing \( \text{contains}(x) \) iff \( x \) has been absent from the abstract set since \( p \) invoked \( \text{contains}(x) \). The invocation of the intermediate \( \text{contains}(x) \) operation sets \( \text{seen\textunderscore out}(p) \) to \( \text{false} \) if \( x \) is in \( S \) and to \( \text{true} \) otherwise. Furthermore, when the main transition of a \( \text{remove}(x) \) occurs, in addition to linearizing itself, it linearizes all invoked \( \text{contains}(x) \) operations that have not yet set their \( \text{res}(p) \) value. Therefore, \( I\text{ContInv}_p(x) \) and \( I\text{RemOK}_p(x) \), which respectively, invoke the contains and perform the main remove operation in the intermediate specification are defined as follows:

\[
\begin{align*}
I\text{ContInv}_p(x) & \triangleq \text{ContInv}_p(x) \land \text{seen\textunderscore out}'(p) = (x \notin S) \\
I\text{RemOK}_p(x) & \triangleq \text{RemOK}_p(x) \land \forall q: P \cdot pc(q) = \text{conti} \land v(q) = v(p) \Rightarrow \text{seen\textunderscore out}'(q) = \text{true}
\end{align*}
\]

The intermediate \( \text{contains}(x) \) operation is allowed to return \( \text{false} \) whenever \( \text{seen\textunderscore out}(x) \) holds, therefore, \( \text{ContFail} \) is replaced by \( I\text{ContFail}_p \) below:

\[
\begin{align*}
I\text{ContFail}_p & \triangleq \Delta I\text{State} \\
& \text{pc}(p) = \text{conti} \land \text{seen\textunderscore out}(p) \land \text{res}'(p) = \text{false} \land \text{pc}'(p) = \text{conti}
\end{align*}
\]

Unlike \( \text{ContFail} \), transition \( I\text{ContFail}_p \) is enabled, and hence, \( \text{res}(p) \) can be set to \( \text{false} \) even if \( v(p) \in S \) holds in the current state. This is allowable because whenever \( pc(p) = \text{conti} \land \text{seen\textunderscore out}(p) \) holds, a state for which \( v(p) \notin S \) holds must have occurred at some point since the invocation of the \( \text{contains} \) operation. Note that when \( pc(p) = \text{conti} \land \text{seen\textunderscore out}(p) \land v(p) \in S \) holds, both \( I\text{ContOK}_p \) and \( I\text{ContFail}_p \) are enabled and process \( p \) may non-deterministically chooses to linearize with \( \text{res}(p) = \text{true} \) (in the current state) or with \( \text{res}(p) = \text{false} \) (having been linearized at some point in the past).

The backward simulation relation \( \text{bsr} \) 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:

\[
\text{bsr}(is, as) \triangleq is(S) = as(S) \land \\
\forall p \cdot is(pc(p)) = as(pc(p)) \lor \left(is(pc(p)) = \text{conti} \land is(\text{seen\textunderscore out}(p)) \land \left(as(pc(p)) = \text{conti} \land as(\text{res}(p)) = \text{false}\right)\right)
\]

The second disjunct within the universal quantification is needed because \( p \) may have already executed \( \text{ContFail}_p \) in the abstract, and decided to return \( \text{false} \), whereas the corresponding intermediate operation has not yet made its choice. This delay in choice is only allowed if \( \text{seen\textunderscore out}(p) \) 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 \( \text{contains} \) operation must be introduced. For example:

\[
\forall p \cdot \left((pc(p) = 4c \land \text{val(curr)(p)} \neq v(p)) \lor (pc(p) \in \{4c,4d\} \land \text{mark(curr)(p)})\right) \Rightarrow \text{seen\textunderscore out}(p)
\]

\[
\forall p \cdot pc(p) = 4d \land \lnot \text{mark(curr)(p)} \Rightarrow v(p) \in auxS
\]

By (8), if the concrete program is in a position to return \( \text{false} \), 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.
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 \texttt{contains} operation is therefore given by a single operation \texttt{AbsCont} below.

\[
\begin{array}{c|c|c|c}
\text{Contains}_p & \Delta \text{AState} & x: V & r: B \\
\hline
r' &=& (x \in S)
\end{array}
\]

To cope with the non-determinism in the linearization points, yet allow locality in the proof obligations generated, Derrick \textit{et al.} generalise the notion of a status by introducing \texttt{INOUT}(in, out) that covers a situation in which an operation has potentially linearized, where \texttt{in} and \texttt{out} denote the input and output parameters, respectively. Thus, in this new setting:

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

For example, in the lazy set, process \( p \) with status \texttt{INOUT}(3, true) denotes a process that is potentially after its linearization point, has 3 as input and will return \texttt{true}. As in Section 4.4, the proof proceeds by identifying the status for a concrete state and a process. For a \texttt{contains}(x) operation executed by process \( p \), given that \( cs \) is a concrete state,

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

While executing \( C2, C3 \) or \( C4a \), a \texttt{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 \texttt{contains} to \texttt{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 \texttt{curr}(p) will change \( p \)’s status to \texttt{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. In particular, the left diagram of Figure 11 shows the case where the execution of operation \( COp_p \) definitely sets its own as well as the linearization point of process \( q \) that executes an operation that does not modify the global state (e.g., a \texttt{contains} operation). The right hand side depicts the case where the abstract operation of process \( p \) is either no or a potential LP for \( p \), and is therefore not allowed to modify the abstract state.
5.4. Method 3: Auxiliary variables

The auxiliary variable method of Vafeiadis also requires substantial changes to cope with verification of the \texttt{contains} operation. In particular, auxiliary statements that are able to linearize the currently executing \texttt{contains} operations must be introduced. As with the methods in Sections 5.2 and 5.3, a \texttt{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 \texttt{contains}.

The augmented state introduces a further auxiliary variable \texttt{OutSet} of type \( P \times V \times \mathbb{B} \), where \((p,v,r) \in \texttt{OutSet}\), iff process \( p \) is executing a \texttt{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 \texttt{Lock\_p}, \texttt{Unlock\_p}, \texttt{Add\_p} and \texttt{Remove\_p} are as given in Section 4.5. The \texttt{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 \texttt{OutSet}. In addition to setting the marked value to \texttt{true} and removing \( v \) from the abstract set, the executing process \( p \) also sets the return value of all processes \( C \subseteq \texttt{OutSet} \) that are currently executing a \texttt{contains(v)} to \texttt{false}, which linearizes each of the processes in \( C \).

\[
\begin{align*}
Mark_p & \quad \Delta \text{AugState} \\
\text{n, n1: Node} & \\
B, C: P \times V \times \mathbb{B} & \\
v: V & \\
r: \mathbb{B} &
\end{align*}
\]

\[
\begin{align*}
\text{lvn}(n1) = (p,v,n) \land \texttt{OutSet} = B \cup C \land (\forall b: B \cdot b.2 \neq v) \land (\forall c: C \cdot c.2 = v) \\
lvnm'(n1) = (p,v,n,\text{true}) \\
S' = S\setminus \{v\} \\
\text{OutSet}' = B \cup \{(q,v,\text{false}) \mid \exists r \cdot (q,v,r) \in C\}
\end{align*}
\]

In addition, two environment operations that add and remove triples of type \( P \times V \times \mathbb{B} \) to/from the auxiliary variable \texttt{OutSet} are also introduced. These represent environment processes that start and complete a \texttt{contains} operation.

\[
\begin{align*}
\text{AddOut}_p & \quad \Delta \text{AugState} \\
v: V & \\
r: \mathbb{B} &
\end{align*}
\]

\[
\begin{align*}
(p,v) \notin \texttt{OutSet} & \\
\text{OutSet}' = \texttt{OutSet} \cup \{(p,v,r)\}
\end{align*}
\]

\[
\begin{align*}
\text{RemOut}_p & \quad \Delta \text{AugState} \\
v: V & \\
r: \mathbb{B} &
\end{align*}
\]

\[
\begin{align*}
(p,v,r) \in \texttt{OutSet} & \\
\text{OutSet}' = \texttt{OutSet}\setminus\{(p,v,r)\}
\end{align*}
\]

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

\[
\begin{align*}
\text{remove}(x): & \\
\text{...} & \\
\text{R3: } & <n2.mark := \text{true}; \text{AbsRes(this)} := (x \in S)\; \text{for each } q \in \texttt{OutSet} \text{ do} \\
& \quad \text{if } q.2 = n2.val \text{ then } \text{AbsRes}(q) := \text{false} \text{ endif} \text{ enddo}> & \\
\text{...} &
\end{align*}
\]

The augmented version of the \texttt{contains} operation is given below. The presentation in [Vafeiadis 2007] suffers from a few typos, which are confirmed by the proof in [Liang and Feng 2013a]. In particular, the auxiliary code that linearizes itself (in statement \texttt{C4a}) should only set the abstract...
result to true if both \textbf{not} res and curr.val = e hold, as opposed to only \textbf{not} res as indicated in [Vafeiadis 2007]. Below, like [Vafeiadis 2007], details of the annotation for the proof outline are elided, but the interested reader may consult [Liang and Feng 2013a].

\begin{verbatim}
contains(x) :
  <AbsRes(this) := (x \not\in S); OutSet := OutSet \cup \{this\}>
C1: curr := Head;
...
C4a: <resa := curr.marked;
    AbsRes(this) := (not resa \and curr.val = x); OutOps := OutOps \setminus \{this\}>
...
\end{verbatim}

The augmentation is such that any process \( p \) that invokes contains(x) initially linearizes to \textit{true} or \textit{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 \textit{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 [Moszkowski 1997; 2000] and its variations (e.g., [Dongol et al. 2012; Dongol et al. 2013; Schellhorn et al. 2011; Bäumler et al. 2010]). Schellhorn \textit{et al.} have exploited rely-guarantee reasoning over intervals to mechanically verify a number of algorithms [Schellhorn et al. 2011; Bäumler et al. 2010]. 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 [Dongol and Derrick 2012; 2013] take inspiration from the reduction-based approaches (see Section 3.4.1), 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 [Dongol and Derrick 2013; 2012] 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 [Jones 1983], 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 [Hayes et al. 2013], and hence both orders of evaluation (which is a contiguous set of times), and an infinite stream over intervals add(y) that executes a contains(x) that returns true over Δp, a process q that executes remove(x) and add(y) over intervals Δq and Δq′, respectively, and a process u that executes add(x) over interval Δu. The shared data structure may be changing over Δp while process p is checking to see whether x is in the set.

A possible coarse-grained abstraction of contains(x) is an operation that tests whether x is in the set in a single atomic step, unlike the implementation in Figure 10, 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 12, an execution of contains that returns true, i.e., given that Cω denotes possibly infinite iteration of C, one must prove both

\[
\begin{align*}
\langle x \in S \rangle \; \text{return true} & \sqsubseteq C1; (C2; C3)\omega; C4; \text{return true} & (R1) \\
\langle x \in S \rangle \; \text{return false} & \sqsubseteq C1; (C2; C3)\omega; C4; \text{return false} & (R2)
\end{align*}
\]

where C1 - C4 are the labels of contains in Figure 10 and \(\langle x \in S \rangle\) is a guard that is atomically able to test whether x is in the abstract set S. In particular, \(\langle x \in S \rangle\) holds in an interval Ω and stream s iff \(x \in S\) holds the state corresponding to time t, where \(t \in \Omega\). Note that both \(\langle x \in S \rangle\) and \(\langle x \not\in S \rangle\) may hold for \(\Delta_p\); the refinement (R1) above would only be invalid if \(x \not\in S\) 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 test 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 [Herlihy and Wing 1990], 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 [Guerraoui et al. 2012]). In some cases linearizability and atomicity may be used synonymously [Raynal 2013]. Filipović et al. [Filipović et al. 2010] 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 [Birman 1992], databases [Ramanritham and Chrysanthis 1992] and fault-tolerant systems [Guerraoui and Schiper 1996]. Stronger notions of linearizability such as R-linearizability have also been developed [Pacull and Sandoz 1993], 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 [Lamport 1997; Herlihy and Shavit 2008], i.e., that the order of events within a process is maintained. However, modern processors use local caches for efficiency and hence are not sequentially consistent. Instead, they only implement weak memory models a number of weaker only ensure weaker guarantees [Adve and Gharachorloo 1996]. Shavit purports quiescent consistency as the correctness criteria for data structures in the multicore age [Shavit 2011]. 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 [Afek et al. 2010], k-linearizability [Henzinger et al. 2013a], and eventual consistency [Shapiro and Kemme 2009]. Quasi linearizability formalises a notion of ‘almost linearizable’, where permutations of a linearization of a concurrent history up to some ‘quasi-factor distance’ are permitted, e.g., a linearization history \( \langle \text{Enq}(a), \text{Enq}(b), \text{Enq}(c), \text{Deq}(c) \rangle \) is a 2-quasi linearizable history of a queue because the enqueue of \( c \) may be ‘discarded’ with a cost of 2 by allowing \( \text{Enq}(c) \) to commute with both \( \text{Enq}(a) \) and \( \text{Enq}(b) \). The relaxation scheme k-linearizability weakens the semantics of the abstract data structure that is being implemented, e.g., a k-linearizable queue would allow either of the first \( k \) elements to be dequeued with the additional restriction that the first element is one of the first \( k \) elements dequeued. Unlike quasi-linearizability, which weakens the requirement on the histories generated by an implementation, k-linearizability weakens the abstract interpretation of the implementation, and hence are incomparable [Henzinger et al. 2013a]. Eventual consistency (a weaker condition than sequential consistency [Burckhardt et al. 2012]) states that all observations on a system will agree if there are no more updates to the system.

In many applications, one must often consider the progress properties that an algorithm guarantees. Here, like safety, several different types of progress conditions have been identified such as starvation freedom, wait freedom, lock freedom and obstruction freedom (see [Herlihy and Shavit 2008; 2011; Dongol 2009; 2006; Tofan et al. 2010]). 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 [Doherty et al. 2004a; Colvin and Groves 2005].

The central theme of each proof method has been the establishment of a refinement relation between the concrete algorithm at hand, and an abstract representation that the concrete program is required to implement. This is seen to be sufficient because if one can identify a linearization point in each concrete operation, which is an atomic statement whose execution causes the effect of the operation to take place. As we have seen, this task is non-trivial due to the possibility of operations being linearized by other operations, and often requires knowledge of potential future behaviours.
Current proof techniques continue to struggle with the scalability and only a handful of fine-grained algorithms have been formally verified to be linearizable. The longest fully verified algorithm (in terms of lines of code) is the Snark algorithm [Doherty et al. 2004a]. However, number of lines of code is not an indicator of complexity, with even simple algorithms like Herlihy and Wing’s queue [Herlihy and Wing 1990] posing immense challenges [Schellhorn et al. 2012; Henzinger et al. 2013b] due to the fact that future behaviour must be considered. As novel techniques for verifying linearizability continue to be developed (at an increasing pace) a survey of achievements of different the current state-of-the-art is necessary.

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. [Colvin et al. 2006a], 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 [Vechev et al. 2009; Friggens 2013; Liu et al. 2009; Liu et al. 2013]; 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 [Vafeiadis 2010], which allows linearizability to be automatically verified. For more complex algorithms, using a setup such as the one provided by Colvin et al. [Colvin et al. 2006a] 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 [Derrick et al. 2011a].

**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 [Schellhorn et al. 2012]. Furthermore, a combination of forwards and backwards simulations is known to be complete for data refinement [Lynch 1996; de Roever and Engelhardt 1996], and combining auxil-
iary and prophecy variables is known to be complete for reasoning about past and future behaviour [Abadi and Lamport 1991]. However, completeness of a method does not guarantee simpler proofs, as evidenced by the maximal backwards simulation used in [Schellhorn et al. 2012].

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., [Henzinger et al. 2013b; Dragoi et al. 2013; O’Hearn et al. 2010b]).

REFERENCES


