Automated Verification of Linearization Policies

Parosh Aziz Abdulla, Bengt Jonsson, and Cong Quy Trinh

Uppsala University

Abstract. We present a novel framework for automated verification of linearizability for concurrent data structures that implement sets, stacks, and queues. The framework requires the user to provide a *linearization policy*, which describes how linearization point placement in different concurrent threads affect each other; such linearization policies are often provided informally together with descriptions of new algorithms. We present a specification formalism for linearization policies which allows the user to specify, in a simple and concise manner, complex patterns including non-fixed linearization points. To automate verification, we extend thread-modular reasoning to bound the number of considered threads, and use a novel symbolic representation for unbounded heap structures that store data from an unbounded domain. We have implemented our framework in a tool and successfully used it to prove linearizability for a wide range of algorithms, including all implementations of concurrent sets, stacks, and queues based on singly-linked lists that are known to us from the literature.

1 Introduction

Data structures that can be accessed concurrently by many parallel threads are a central component of many software applications, and are implemented in several widely used libraries (e.g., `java.util.concurrent`). Linearizability [17] is the standard correctness criterion for such concurrent data structure implementations. It states that each operation on the data structure can be considered as being performed atomically at some point, called the *linearization point (LP)*, between its invocation and return. This allows client threads to understand the data structure in terms of atomic actions, without considering the complications of concurrency.

Linearizable concurrent data structures typically employ fine-grained synchronization, replacing locks by atomic operations such as compare-and-swap, and are therefore notoriously difficult to get correct, witnessed, e.g., by a number of bugs in published algorithms [8,20]. It is therefore important to develop efficient techniques for automatically verifying their correctness. This requires overcoming several challenges.

One challenge is that the criterion of linearizability is harder to establish than standard correctness criteria, such as control state reachability; in fact, proving linearizability with respect to a given data structure specification is undecidable, even in frameworks where verification of temporal safety properties is decidable [5]. This has lead to verification techniques that establish some form of simulation between concurrent and sequential executions, and whose mechanization requires an interactive theorem prover (e.g., [6,7,25,26]). Automation has been successful only under simplifying assumptions. A natural one is that LPs are *fixed*, i.e., can be affixed to particular statements in method implementations [1,3,30]. However, for a large class of linearizable implementations, the LPs are not fixed in the code of their methods, but depend on actions of other threads in each particular execution. This happens, e.g., for algorithms that employ various forms of *helping mechanisms*, in which the execution of a particular statement in one thread defines the LP for one or several other threads [13,14,34,35].

Another challenge is that verification techniques must be able to reason about fine-grained concurrent programs that are infinite-state in many dimensions: they consist of an unbounded number of concurrent threads, which operate on an unbounded domain of data values, and use unbounded dynamically allocated memory. This challenge has been addressed by bounding the number of accessing threads [2,33,32], restricting the class of algorithms that can be verified [15,31], or requiring auxiliary lemmas [24,36].

Contributions In this paper, we present a novel uniform framework for automatically verifying linearizability of concurrent data structure implementations, which handles the above challenges. In our framework, the user provides (i) a C-like description of the data structure implementation, (ii) a specification of sequential data structure semantics, using the simple technique of *observers* [1], and (iii) a *linearization policy*, which describes how LP placement in different concurrent threads affect each other; such linearization policies are often provided informally together with descriptions of new data structure implementations. Our framework then automatically checks that the requirement of linearizability (wrp. to the data structure specification) is satisfied when LPs are placed according to the given linearization policy. Ours is the first framework that can automatically verify all linearizable singly-linked list-based implementations of sets, stacks, and queues that have appeared in the verification literature, requiring only a small amount of user annotation (of linearization policies). Our framework relies on a number of advancements over the state-of-the-art.

1. We handle non-fixed LPs by a novel formalism for specifying linearization policies, by means of so-called *controllers*. Linearization policies capture inter-thread constraints on LP placement. They are often described informally by the algorithm designers together with each new data structure implementation when explaining why it is linearizable. Our controllers offer a way to express such policies in a simple and uniform manner. They can express complex patterns for linearization that are much more general than fixed LPs, including all the ones that are known to us from the literature, such as helping and flat-combining [14,13,34,35,23,24,12]. Each method is equipped with a controller, whose task is to announce the occurrence of potential linearization points for that method. The controller is defined by a few simple rules that are triggered by its thread, and may also interact with controllers of other threads, in order to properly announce LPs. We specify the data structure semantics by adapting the technique of *observers* [1], that check correctness of the sequence of LP announcements. We extend previous usage of observers by allowing a controller to announce an LP several times. This extension allow us to handle implementations where the occurrence of a (non-fixed) LP is conditional on some predicted future condition; a false prediction can then (under some restrictions) be corrected by a renewed LP announcement. In previous approaches, such situations were handled by prophecy variables or backward simulation (e.g., [25]), that make

automated verification significantly more difficult. The use of observers and controllers reduces verification to establishing that each method invocation generates a sequence of LP announcements, the last of which may change the state of the observer and conforms to the method's call and return parameters, such that that the total sequence of LP announcements is not rejected by the observer. The establishment of these conditions can be reduced to control state reachability using standard techniques; our framework accomplishes this by automatically generated *monitors*.

2. We handle the challenge of an unbounded number of threads by extending the successful thread-modular approach which verifies a concurrent program by generating an invariant that correlates the global state with the local state of an arbitrary thread [3]. We must extend it to handle global transitions where several threads synchronize, due to interaction between controllers. We show that the number of synchronizing threads that need to be considered in the abstract postcondition computation is bounded by $3 \cdot (\#\mathcal{O}) + 2$ where $\#\mathcal{O}$ is the diameter of the observer. Furthermore, we define a condition, which we call *stuttering*, that allows to reduce the number of synchronizing threads to only two. The stuttering condition can be verified through a simple syntactic check and is indeed satisfied by all the examples that we have considered.
3. In order to reason about concurrent programs that operate on an unbounded data domain via dynamically allocated memory, we present a novel symbolic representation of singly-linked heap structures.

For stacks and queues Chakraborty et al. [15], and later Boujjani et al.[4], showed that linearizability can be checked by observing only the sequence of method calls and returns, without considering potential LP placement. This technique cannot check linearizabilitiy of sets. We can adapt their technique to our framework. By using observers as defined in [4,15], and adapting controllers accordingly, we can use our symbolic verification technique to automatically verify stacks and queues without considering potential LP placement.

We have implemented our technique in a tool, and applied it to specify and automatically verify linearizability of all the implementations of concurrent set, queue, and stack algorithms known to us in the literature, as well as some algorithms for implementing atomic memory read/write operations. To use the tool, the user needs to provide the code of the algorithm together with the controllers that specify linearization policies. To our knowledge, this is the first time all these examples are verified fully automatically in the same framework.

Related Work Much previous work has been devoted to the *manual* verification of linearizability for concurrent programs. Examples include [18,28]. In [24], O'Hearn *et al.* define a *hindsight lemma* that provides a non-constructive evidence for linearizability. The lemma is used to prove linearizability of an optimistic variant of the lazy set algorithm. Vafeiadis [29] uses forward and backward simulation relations together with history or prophecy variables to prove linearizability. These approaches are manual, and without tool implementations. *Mechanical* proofs of linearizability, using interactive theorem provers, have been reported in [6,7,26,25]. For instance, Colvin *et al.* [6] verify the lazy set algorithm in PVS, using a combination of forward and backward simulations.

There are several works on *automatic* verification of linearizability. In [31], Vafeiadis develops an automatic tool for proving linearizability that employs instrumentation to verify logically pure executions. However, this work can handle non-fixed LPs only for read-only methods, i.e, methods that do not modify the heap. This means that the method cannot handle algorithms like the *Elimination* queue [23], *HSY* stack [14], *CCAS* [12], *RDCSS* [12] and *HM* set [16] that we consider in this paper. In addition, their shape abstraction is not powerful enough to handle algorithms like *Harris* set [11] and *Michael* set [22] that are also handled by our method. Chakraborty *et al.* [15] describe an "aspect-oriented" method for modular verification of concurrent queues that they use to prove linearizability of the Herlihy/Wing queue. Bouajjani et al. [4] extended this work to show that verifying linearizability for certain fixed abstract data types, including queues and stacks, is reducible to control-state reachability. We can incorporate this technique into our framework by a suitable construction of observers. The method can not be applied to sets. The most recent work of Zhu *et al.* [36] describe a tool that is applied for specific set, queue, and stack algorithms. For queue algorithms, their technique can handle queues with helping mechanism except for *HW* queue [17] which is handled by our paper. For set algorithms, the authors can only handle those that perform an optimistic contains (or lookup) operation by applying the *hindsight lemma* from [24]. Hindsight-based proofs provide only *non-constructive* evidence of linearizability. Furthermore, some algorithms (e.g., the unordered list algorithm considered in Sec. 8 of this paper) do not contain the code patterns required by the hindsight method. Algorithms with non-optimistic contains (or lookup) operation like *HM* [16], *Harris* [11] and *Michael* [22] sets cannot be verified by their technique. Vechev *et al.* [33] check linearizability with user-specified non-fixed LPs, using a tool for finite-state verification. Their method assumes a bounded number of threads, and they report state space explosion when having more than two threads. Dragoi *et al.* [10] describe a method for proving linearizability that is applicable to algorithms with non-fixed LPs. However, their method needs to rewrite the implementation so that all operations have linearization points within the rewritten code. Černý *et al* [32] show decidability of a class of programs with a bounded number of threads operating on concurrent data structures. Finally, the works [1,3,30] all require fixed linearization points.

We have not found any report in the literature of a verification method that is sufficiently powerful to automatically verify the class of concurrent set implementations based on sorted and non-sorted singly-linked lists having non-optimistic contains (or lookup) operations we consider. For instance the lock-free sets of *HM* [16], *Harris* [11], or *Michael* [22], or unordered set of [35],

2 Data Structures, Observers, and Linearizability

Data Structure Semantics. A *data structure* $\mathtt{DS}$ is a pair $\langle \mathbb{D}, \mathbb{M} \rangle$ where $\mathbb{D}$ is the *data domain* and $\mathbb{M}$ is the alphabet of method names. An *operation op* is of the form $\mathtt{m}(d^{in}, d^{out})$ where $\mathtt{m} \in \mathbb{M}$ is a method name and $d^{in}, d^{out} \in \mathbb{D}$ are the *input* resp. *output* data values. A *trace* of $\mathtt{DS}$ is a sequence of operations. The behavior $[\![\mathtt{DS}]\!]$ of $\mathtt{DS}$ is the set of traces. We often identify $\mathtt{DS}$ with its behavior $[\![\mathtt{DS}]\!]$. For example, in the `Set` data structure, the set of method names is given by $\{\mathtt{add}, \mathtt{rmv}, \mathtt{ctn}\}$, and the data domain is the union $\mathbb{Z} \cup \mathbb{B}$ of the sets of integers and Booleans. Input and output data values are in $\mathbb{Z}$ and $\mathbb{B}$ respectively. For instance, the operation $\mathtt{add}(3, \mathtt{tt})$ successfully adds the value 3, while $\mathtt{ctn}(2, \mathtt{ff})$ is a failed search for 2 in the set.

Observers We specify traces of data structures by *observers*, as introduced in [1]. Observers are finite automata extended with a finite set of *registers* that assume values in $\mathbb{Z}$. At initialization, the registers are nondeterministically assigned arbitrary values, which never change during a run of the observer. Formally, an observer $\mathcal{O}$ is a tuple $\langle S^{\mathcal{O}}, s^{\mathcal{O}}_{\mathtt{init}}, \mathtt{X}^{\mathcal{O}}, \Delta^{\mathcal{O}}, s^{\mathcal{O}}_{\mathtt{acc}} \rangle$ where $S^{\mathcal{O}}$ is a finite set of *observer states* including the *initial state* $s^{\mathcal{O}}_{\mathtt{init}}$ and the *accepting state* $s^{\mathcal{O}}_{\mathtt{acc}}$, a finite set $\mathtt{X}^{\mathcal{O}}$ of *registers*, and $\Delta^{\mathcal{O}}$ is a finite set of *transitions*. Transitions are of the form $\langle s_1, \mathtt{m}(x^{in}, x^{out}), s_2 \rangle$ where x^{in} and x^{out} are either registers or constants, i.e., transitions are labeled by operations whose input or output data may be parameterized on registers. The observer accepts a trace if it can be processed in such a way that an accepting state is reached. Observers can be used to give *exact* specifications of the behaviors of data structures such as sets, queues, and stacks. The observer is defined in such a way that it accepts precisely those traces that do *not* belong to the behavior of the data structure. This is best illustrated by an example. Fig. 1 depicts an observer that accepts the sequences of operations that are *not* in $[\![\mathtt{Set}]\!]$.

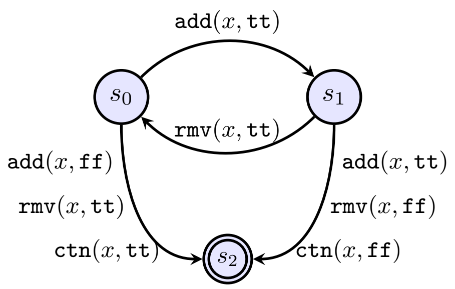

Fig. 1: Set observer.

Linearizability. An operation $\mathtt{m}(d^{in}, d^{out})$ gives rise to two *actions*, namely a *call action* $\left(\mathtt{eid}, \mathtt{m}, \mathtt{d}^{in}\right) \downarrow$ and a *return action* $(\mathtt{eid}, \mathtt{m}, \mathtt{d}^{out}) \uparrow$, where $\mathtt{eid} \in \mathbb{N}$ is an *action identifier*. A *history* h is a sequence of actions such that for each return action a_2 in h there is a unique *matching* call action a_1 where the action identifiers in a_1 and a_2 are identical and different from the action identifiers of all other actions in h and such that a_1 occurs before a_2 in h. A call action carries always the same method name as its matching return action. A call action which does not match any return action in h is said to be *pending*. A history without pending call actions is said to be *complete*. A *completed extension* of h is a complete history h' obtained from h by appending (at the end) zero or more return actions that are matched by pending call actions in h, and thereafter removing pending call actions. For action identifiers $\mathtt{eid}_1, \mathtt{eid}_2$, we write $\mathtt{eid}_1 \preceq_{\mathtt{h}} \mathtt{eid}_2$ to denote that the return action with identifier $\mathtt{eid}_1$ occurs before the call action with identifier $\mathtt{eid}_2$ in h. We say that a history is *sequential* if it is of the form $a_1 a'_1 a_2 a'_2 \cdots a_n a'_n$ where a'_i is the matching action of a_i for all $i : 1 \leq i \leq n$, i.e., each call action is immediately followed by the matching return action. We identify a sequential history of the above form with the corresponding trace $op_1\, op_2 \cdots op_n$ where $op_i = \mathtt{m}(d^{in}_i, d^{out}_i)$, $a_i = \left(\mathtt{eid}_\mathtt{i}, \mathtt{m}, \mathtt{d}^{in}_\mathtt{i}\right) \downarrow$, and $a_i = \left(\mathtt{eid}_\mathtt{i}, \mathtt{m}, \mathtt{d}^{out}_\mathtt{i}\right) \uparrow$, i.e., we merge each call action together with the matching return action into one operation. A complete history h' is a *linearization* of h if (i) h' is a permutation of h, (ii) h' is sequential, and (iii) $\mathtt{eid}_1 \preceq_{\mathtt{h'}} \mathtt{eid}_2$ if $\mathtt{eid}_1 \preceq_{\mathtt{h}} \mathtt{eid}_2$ for each pair of action identifiers $\mathtt{eid}_1$ and $\mathtt{eid}_2$. We say that a sequential history h' is *valid* wrt. $\mathtt{DS}$ if the corresponding trace is in $[\![\mathtt{DS}]\!]$. We say that h is *linearizable* wrt. $\mathtt{DS}$ if there is a completed extension of h, which has a linearization that is valid wrt. DS. A set H of histories is *linearizable* wrt. DS if all members of H are *linearizable* wrt. DS.

3 Programs

In this section, we introduce programs that consist of arbitrary numbers of concurrently executing threads that access a concurrent data structure. Each thread executes a method that performs an operation on the data structure. Each method declares local variables and a method body. Variables are either pointer variables (to heap cells), or data variables, assuming values from an infinite (ordered) domain, or from some finite set $\mathbb{F}$ that includes the Boolean values $\mathbb{B}$. We assume w.l.o.g. that the infinite set is given by the set $\mathbb{Z}$ of integers. The body is built in the standard way from atomic commands, using standard control flow constructs (sequential composition, selection, and loop constructs). Each statement is equipped with a unique label. We assume that the set of local variables include the parameter of the method in addition to the program counter `pc` whose value is the label of the next statement to be executed. Method execution is terminated by executing a `return` command, which may return a value. The global variables can be accessed by all threads, whereas local variables can be accessed only by the thread which is invoking the corresponding method. We assume that all global variables are pointer variables, and that they are initialized, together with the heap, by an initialization method, which is executed once at the beginning of program execution.

Heap cells have a fixed set $\mathcal{F}$ of fields, namely data fields that assume values in $\mathbb{Z}$ or $\mathbb{F}$, and lock fields. Furthermore, each cell has one pointer field, denoted `next`, and hence heap cells are organized into singly-linked lists.We use the term $\mathbb{Z}$-field for a data field that assumes values in $\mathbb{Z}$, and the terms $\mathbb{F}$-field and lock field with analogous meaning. Atomic commands include assignments between data variables, pointer variables, or fields of cells pointed to by a pointer variable. The command `new Node()` allocates a new structure of type `Node` on the heap, and returns a reference to it. The compare-and-swap command `CAS(&a,b,c)` atomically compares the values of `a` and `b`. If equal, it assigns the value of `c` to `a` and returns `true`, otherwise, it leaves `a` unchanged and returns `false`. We assume a memory management mechanism, which automatically collects garbage, and also ensures that a new cell is fresh, i.e., has not been used before by the program; this avoids the so-called ABA problem (e.g., [21]).

Fig. 2 depicts a program `Lazy Set` [13] that implements a concurrent set containing elements from $\mathbb{Z}$. The set is implemented as an ordered singly linked list. The program contains three methods, `add`, `rmv`, and `ctn`, corresponding to operations that add, remove, and check the existence of an element in the set, respectively. Each method takes an element as argument, and returns a value which indicates whether or not the operation has been successful. For instance, the operation add(e) returns the value `true` if `e` was not already a member of the set. In such a case a new cell with data value `e` is added to its appropriate position in the list. If `e` is already present, then the list is not changed and the value `false` is returned. The program also contains the subroutine `locate` that is called by `add` and `rmv` methods. A cell in the list has two data fields: mark : $\mathbb{F}$, where $\mathbb{F} = \{0, 1\}$, and val : $\mathbb{Z}$, one lock field lock, and one pointer field next. The `rmv` method first logically removes the node from the list by setting the `mark` field, before physically removing the node. The `ctn` method is wait-free and traverses the list ignoring the locks inside the cells.

```
struct Node {
        bool lock;
        int val;
        Node* next;
        bool mark;
   }
```

```
   ctn(e):
   local c
1  c := Head;
2  while (c.val < e)
3    c := c.next
4  b := c.mark ●
5  if (!b
        && c.val = e)
6    return true;
7  else
8    return false;
```

```
 locate(e):
 local p, c
 1  while (true)
 2    p := Head;
 3    c := p.next;
 4    while (c.val < e)
 5      p := c;
 6      c := c.next
 7    lock(p); lock(c);
 8    if (! p.mark &&
          ! c.mark&&
          p.next=c)
 9      return(p,c);
10    else
11      unlock(p);
12      unlock(c);
```

```
add(e):
local p, c, n, r
1  (p,c) := locate(e);
2  if (c.val <> e) ●
3    n :=
     new Node(
         0,e,c,false);
4     p.next := n; ●
5     r := true;
6  else r := false;
7  unlock(p);
8  unlock(c);
9  return r;
```

```
rmv(e):
local p, c, n, r
1  (p,c) := locate(e);
2  if (c.val = e) ●
3     c.mark := true; ●
4     n := c.next;
5     p.next := n;
6     r := true;
7  else r := false;
8  unlock(p);unlock(c);
9  return r;
```

```
initialize() :
  Head := new Node(
     0,-∞,null,false);
  Tail := new Node(
     0,+∞, null,false)
  Head.next := Tail;
```

Fig. 2: `Lazy Set.`

add

ρ_1 **when** ● **provided** `pc=4` **emit** `add(e,true)` **broadcast** `add(e)`

ρ_2 **when** ● **provided** `pc=2 && (c.val = e)` **emit** `add(e,false)`

rmv

ρ_3 **when** ● **provided** `pc=3` **emit** `rmv(e,true)`

ρ_4 **when** ● **provided** `pc=2 && c.val <> e` **emit** `rmv(e,false)`

ctn

ρ_5 **when** ● **provided** `pc=4 && !b && c.val = e` **emit** `ctn(e,true)`

ρ_6 **from** q_0 **when** ● **provided** `pc=4 && !(!b&&c.val=e)` **emit** `ctn(e,false)` **goto** q_0

ρ_7 **from** q_0 **when** `(add(e),before)` **provided** `1<=pc<=4` **emit** `ctn(e,false)` **goto** q_1

Fig. 3: Reaction Rules for Controllers of `Lazy Set`.

4 Linearization Policies

In this section, we introduce our formalism of *controllers* for expressing linearization policies. A linearization policy is expressed by defining for each method an associated controller, which is responsible for generating operations announcing the occurrence of LPs during each method invocation. The controller is occasionally activated, either by its thread or by another controller, and mediates the interaction of the thread with the observer as well as with other threads.

To add controllers, we first declare some statements in each method to be *triggering*: these are marked by the symbol ●, as in Figure 2. We specify the behavior of the controller, belonging to a method $\mathtt{m}$, by a set $R^{\mathtt{m}}$ of *reaction rules*. To define these rules, we first define different types of events that are used to specify their behaviors. Recall that an *operation* is of the form $\mathtt{m}(d^{in}, d^{out})$ where $\mathtt{m}$ is a method name and $d^{in}, d^{out} \in \mathbb{D}$ are data values. Operations are emitted by the controller to the observer to notify that the thread executing the method performs a linearization of the corresponding method with the given input and output values. Next, we fix a set Σ of *broadcast messages*, each with a fixed arity, which are used for synchronization between controllers. A message is formed by supplying data values as parameters. In reaction rules, these data values are denoted by expressions over the variables of the method, which are evaluated in the current state when the rule is invoked. In an operation, the first parameter, denoting input, must be either a constant or the parameter of the method call.

- A *triggered* rule, of form **when** ● **provided** cnd **emit** op **broadcast** se, specifies that whenever the method executes a triggering statement and the condition cnd evaluates to `true`, then the controller performs a *reaction* in which it emits the operation obtained by evaluating op to the observer, and broadcasts the message obtained by evaluating se to the controllers of other threads. The broadcast message se is optional.
- A *receiving* rule, of form **when** $\langle re, \mathtt{ord} \rangle$ **provided** cnd **emit** op, specifies that whenever the observer of some other thread broadcasts the message obtained by evaluating re, and cnd evaluates to `true`, then the controller performs a reaction where it emits the operation obtained by evaluating op to the observer. Note that no further broadcasting is performed. The interaction of the thread with the observer may occur either *before* or *after* the sender thread, according to the flag `ord`.

A controller may also use a finite set of states, which restrict the possible sequences of reactions by a controller in the standard way. Whenever such states are used, the rule includes source and target states using kewords **from** and **goto**. In Figure 3, the rule ρ_7 changes the state from q_0 to q_1, meaning that no further applications of rules ρ_6 or ρ_7 are possible, since they both start from state q_0. Rules that do not mention states can be applied regardless of the controller state and leave it unchanged.

Let us illustrate how the reaction rules for controllers in Figure 3 specify LPs for the algorithm in Figure 2. Here, a successful `rmv` method has its LP at line `3`, and an unsuccessful `rmv` has its LP at line `2` when the test `c.val = e` evaluates to `false`. Therefore, both these statements are marked as triggering. The controller has a reaction rule for each of these cases: in Figure 3: rule ρ_3 corresponds to a successful `rmv`, whereas rule ρ_4 corresponds to an unsuccessful `rmv`. Rule ρ_4 states that whenever the `rmv` method executes a triggering statement, from a state where `pc=2` and `c.val <> e`, then the operation `rmv(e,false)` will be emitted to the observer.

A successful `add` method has its LP at line `4`. Therefore, the controller for `add` has the triggered rule ρ_1 which emits the operation `add(e,true)` to the observer. In addition, the controller also broadcasts the message `add(e)`, which is received by any controller for a `ctn` method which has not yet passed line `4`, thereby linearizing an unsuccessful `ctn(e)` method by emitting `ctn(e,false)` to the observer. The keyword **before** denotes that the operation `ctn(e,false)` will be presented before `add(e,true)` to the observer. Since the reception of `add(e)` is performed in the same atomic step as the triggering statement at line `4` of the `add` method, this describes a linearization pattern, where a `ctn` method, which has not yet reached line `4`, linearizes an unsuccessful `ctn`-invocation just before some other thread linearizes a successful `add` of the same element.

To see why unsuccessful `ctn` method invocations may not have fixed LPs, note that the naive attempt of defining the LP at line `4` provided that the test at line `5` fails will not work. Namely, the `ctn` method may traverse the list and arrive at line `4` in a situation where the element `e` is not in the list (either `e` is not in any cell, or the cell containing `e` is marked). However, before executing the command at line `4`, another thread performs an `add` operation, inserting a new cell containing the element `e` into the list. The problem is now that the `ctn` method cannot "see" the new cell, since it is unreachable from the cell currently pointed to by the variable `b` of the `ctn` method. If the `ctn` method would now try to linearize an unsuccessful `ctn(e)`, this would violate the semantics of a set, since the `add` method just linearized a successful insertion of `e`.

A solution to this problem, following [13], is to let an unsuccessful `ctn(e)` method linearize at line `4` *only if* no successful `add(e)` method linearized since its invocation. If some other thread linearizes a successful `add(e)` before the `ctn(e)` method executes line `4`, then the `ctn(e)` method should linearize immediately before the `add(e)` method. This solution is expressed by the rules in Figure 3. Note, however, that it is now possible for an invocation of a successful `ctn(e)` to emit several operations: It can first emit `ctn(e,false)` together with the linearization of a successful `add(e)`, and thereafter emit `ctn(e,true)` when it executes the statement at line 4, if it then finds the element `e` in the list (by rule ρ_5). In such a case, both operations are fine with the observer, and the last one is the event that causes the method to return `true`, i.e., conforms with the parameters and returns values.

Verifying Linearization Policies By using an observer to specify the sequential semantics of the data structure, and defining controllers that specify the linearization policy, the verification of linearizability is reduced to establishing four conditions: (i) each method invocation generates a non-empty sequence of operations, (ii) the last operation of a method conforms to its parameters and return value, (iii) only the last operation of a method may change the state of the observer, and (iv) the sequence of all operations cannot drive the observer to an accepting state. Our verification framework automatically reduces the establishment of these conditions to a problem of checking control state reachability. This is done by augmenting the observer by a *monitor*. The monitor is automatically generated. It keeps track of the state of the observer, and records the sequence of operations and call and return actions generated by the threads. For each thread, it keeps track of whether it has linearized, whether it has caused a state change in the observer, and the parameters used in its last linearization. Using this information, it goes to an error state whenever any of the above four conditions is violated. The correctness of this approach is established in the next section, as Theorem 1.

5 Semantics

In this section, we formalize the semantics of programs, controllers, observers, and monitors, as introduced in the preceding sections. We do this in several steps. First, we define the state of the heap and the transition relation induced by a single thread. From this we derive the semantics of the program and use it to define its history set. We then define the semantics of observers, and its augmentation by the monitor; the observer is embedded in the monitor. Finally, we define the product of the augmented program and the monitor. On this basis, we can formally state and prove the correctness of our approach, as Theorem 1.

For a function $f : A \mapsto B$ from a set A to a set B, we use $f[a_1 \leftarrow b_1, \ldots, a_n \leftarrow b_n]$ to denote the function f' such that $f'(a_i) = b_i$ and $f'(a) = f(a)$ if $a \notin \{a_1, \ldots, a_n\}$.

Below, we assume a program $\mathcal{P}$ with a set $\mathtt{X}^{\mathtt{gl}}$ of global variables, a data structure $\mathtt{DS}$ specified as an observer $\mathcal{O}$. We assume that each thread $\mathtt{th}$ executes one method denoted $\mathtt{Method}\,(\mathtt{th})$.

Heaps. A *heap (state)* is a tuple $\mathcal{H} = \left\langle \mathbb{C}, \mathtt{succ}, \mathtt{Val}^{\mathtt{gl}}, \mathtt{Val}^{\mathbb{C}} \right\rangle$, where (i) $\mathbb{C}$ is a finite set of cells, including the two special cells $\mathtt{null}$ and $\perp$ (dangling); we define $\mathbb{C}^- = \mathbb{C} - \{\mathtt{null}, \perp\}$, (ii) $\mathtt{succ} : \mathbb{C}^- \to \mathbb{C}$ is a total function that defines the next-pointer relation, i.e., $\mathtt{succ}\,(\mathbb{c}) = \mathbb{c}'$ means that cell $\mathbb{c}$ points to $\mathbb{c}'$, (iii) $\mathtt{Val}^{\mathtt{gl}} : \mathtt{X}^{\mathtt{gl}} \to \mathbb{C}$ maps the global (pointer) variables to their values, and (iv) $\mathtt{Val}^{\mathbb{C}} : \mathbb{C} \times \mathcal{F} \to \mathbb{F} \cup \mathbb{Z}$ maps data and lock fields of each cell to their values. We assume that the heap is initialized by an initialization thread, and let $\mathcal{H}_{\mathtt{init}}$ denote the initial heap.

Threads. A *local state* $\mathtt{loc}$ of a thread $\mathtt{th}$ wrt. a heap $\mathcal{H}$ defines the values of its local variables, including the program counter $\mathtt{pc}$ and the input parameter for the method executed by $\mathtt{th}$. In addition, there is the special initial state $\mathtt{idle}$, and terminated state $\mathtt{term}$. A *view* $\mathtt{view}$ is a pair $\langle \mathtt{loc}, \mathcal{H} \rangle$, where $\mathtt{loc}$ is a local state wrt. the heap $\mathcal{H}$. A thread $\mathtt{th}$ induces a transition relation $\to_{\mathtt{th}}$ on views. Initially, $\mathtt{th}$ is in the state $\mathtt{idle}$. It becomes active by executing a transition $\langle \mathtt{idle}, \mathcal{H} \rangle \xrightarrow{(\mathtt{th},\mathtt{m},d^{in})\downarrow}_{\mathtt{th}} \langle \mathtt{loc}^{\mathtt{th}}_{\mathtt{init}}, \mathcal{H} \rangle$, labeled by a call action with $\mathtt{th}$ as the event identifier and $\mathtt{m} = \mathtt{Method}\,(\mathtt{th})$. It moves to an initial local state $\mathtt{loc}^{\mathtt{th}}_{\mathtt{init}}$ where d^{in} is the value of its input parameter, the value of $\mathtt{pc}$ is the label of the first statement of the method, and the other local variables are undefined. Thereafter, $\mathtt{th}$ executes statements one after one. We write $\mathtt{view} \xrightarrow{\lambda}_{\mathtt{th}} \mathtt{view}'$ to denote that the statement labeled by λ can be executed from $\mathtt{view}$, yielding $\mathtt{view}'$. Note that the next move of $\mathtt{th}$ is uniquely determined by $\mathtt{view}$, since $\mathtt{th}$ cannot access the local states of other threads. Finally, $\mathtt{th}$ terminates when executing its $\mathtt{return}$ command giving rise to a transition $\langle \mathtt{loc}, \mathcal{H} \rangle \xrightarrow{(\mathtt{th},\mathtt{m},d^{out})\uparrow}_{\mathtt{th}} \langle \mathtt{term}, \mathcal{H} \rangle$, labeled by a return action with $\mathtt{th}$ as event identifier, $\mathtt{m} = \mathtt{Method}\,(\mathtt{th})$, and d^{out} as the returned value.

Programs. A *configuration* of a program $\mathcal{P}$ is a tuple $\langle \mathtt{T}, \mathtt{LOC}, \mathcal{H} \rangle$ where $\mathtt{T}$ is a set of threads, $\mathcal{H}$ is a heap, and $\mathtt{LOC}$ is a *local state mapping* over $\mathtt{T}$ wrt. $\mathcal{H}$ that maps each thread $\mathtt{th} \in \mathtt{T}$ to its local state $\mathtt{LOC}\,(\mathtt{th})$ wrt. $\mathcal{H}$. The initial configuration $c^{\mathcal{P}}_{\mathtt{init}}$ is the pair $\langle \mathtt{LOC}_{\mathtt{init}}, \mathcal{H}_{\mathtt{init}} \rangle$, where $\mathtt{LOC}_{\mathtt{init}}\,(\mathtt{th}) = \mathtt{idle}$ for each $\mathtt{th} \in \mathtt{T}$, i.e., $\mathcal{P}$ starts its execution from a configuration with an initial heap, and with each thread in its initial local state. A program $\mathcal{P}$ induces a transition relation $\to_{\mathcal{P}}$ where each step corresponds to one move of a single thread. This is captured by the rules P1, P2, and P3 of Fig. 4 (here ε denotes the empty word.) Note that the only visible transitions are those corresponding to call and return actions. A *history* of $\mathcal{P}$ is a sequence of form $a_1 a_2 \cdots a_n$, such that there is a sequence $c_0 \xrightarrow{a_1}_{\mathcal{P}} c_1 \xrightarrow{a_2}_{\mathcal{P}} \cdots c_{n-1} \xrightarrow{a_n}_{\mathcal{P}} c_n$ of transitions from the initial configuration $c_0 = c^{\mathcal{P}}_{\texttt{init}}$. We define $\mathcal{H}(\mathcal{P})$ to be the set of histories of $\mathcal{P}$. We say that $\mathcal{P}$ is linearizable wrt. DS if the set $\mathcal{H}(\mathcal{P})$ is linearizable wrt. DS.

Controllers. The controller of a thread th induces a transition relation $\rightarrow_{\texttt{cntrl(th)}}$, which depends on the current view view of th. For an expression *expr* denoting an operation, broadcast message, or condition, let view(*expr*) denote the result of evaluating *expr* in the state view. For a predicate *cnd*, we let view $\models$ *cnd* denote that view(*cnd*) is true. Transitions are defined by the set $R^{\texttt{Method(th)}}$ of rules ($R^{\texttt{th}}$ for short) associated with the method of th. Rule C1 describes transitions induced by a triggered rule, and states that if *cnd* is true under view, then the controller emits the operation obtained by instantiating *op* and broadcasts the message obtained by instantiating *se*, according to view. Rule C2 describes transitions induced by receiving a message e', and states that if *re* equals the value of e' in view, and *cnd* evaluates to true in view, then the operation view(*op*) is emitted. In both cases, the controller also emits to the observer the identity th of the thread. This identity will be used for bookkeeping by the monitor (see below.)

Augmented Threads. An *augmented view* $\langle q, \texttt{loc}, \mathcal{H} \rangle$ extends a view $\langle \texttt{loc}, \mathcal{H} \rangle$ with a state q of the controller. We define a transition relation $\rightarrow_{\texttt{aug(th)}}$ over augmented views that describes the behavior of a thread th when augmented with its controller. For a label λ of a statement we write $\lambda^{\bullet}$ to denote that the corresponding statement is triggered. Rule TC1 states that a non-triggered statement by the thread does not affect the state of the controller, and does not emit or broadcast anything. Rule TC2 states a return action by the thread. Rule TC3 states that when the thread th performs a triggered statement, then the controller will emit an operation to the observer and broadcast messages to the other threads. The reception of broadcast messages will be described in Rule R.

Augmented Programs. An *augmented program* $\mathcal{Q}$ is obtained from a program $\mathcal{P}$ by augmenting each thread with its controller. A *configuration* c of $\mathcal{Q}$ is a tuple $\langle \texttt{T}, \texttt{Q}, \texttt{LOC}, \mathcal{H} \rangle$, which extends a configuration of $\mathcal{P}$ by a *controller state mapping*, Q, which maps each thread th $\in$ T to the state of its controller. We define the size $|c| = |\texttt{T}|$ and define $\texttt{ThreadsOf}(c) = \texttt{T}$. We use $\mathcal{C}^{\mathcal{Q}}$ to denote the set of configurations of $\mathcal{Q}$. Transitions of $\mathcal{Q}$ are performed by the threads in T. The rule Q1 describes return action or a non-triggered statement in which only its local state and the heap may change. Rule Q2 describes the case where a thread $\texttt{th}_s$ executes a triggered statement and hence also broadcasts a message to the other threads. Before describing this rule, we describe the effect of message reception. Consider a set T of threads, a heap $\mathcal{H}$, a controller state mapping Q over T, and a local state mapping LOC over T wrt. $\mathcal{H}$. We will define a relation that extracts the set of threads in T that can receive e from $\texttt{th}_s$. For $\texttt{ord} \in \{\texttt{before}, \texttt{after}\}$, define $\texttt{enabled}(\texttt{T}, \texttt{Q}, \texttt{LOC}, \mathcal{H}, e, \texttt{ord}, \texttt{th}_s)$ to be the set of threads th $\in$ T such that (i) $\texttt{th} \neq \texttt{th}_s$, i.e., a receiver thread should be different from the sender thread, and (ii) $\langle \texttt{LOC(th)}, \mathcal{H} \rangle : \texttt{Q(th)} \xrightarrow{\langle \texttt{th}, op \rangle | \langle e, \texttt{ord} \rangle ?}_{\texttt{cntrl(th)}} q'$ for some q' and op, i.e., the controller of th has an enabled receiving rule that can receive e, with the ordering flag given by

$$\frac{\langle \texttt{idle}, \mathcal{H}\rangle \xrightarrow{(\texttt{th},\texttt{m},d^{in})\downarrow}_{\texttt{th}} \langle \texttt{loc}^{\texttt{th}}_{\texttt{init}}, \mathcal{H}\rangle}{\langle \texttt{T}, \texttt{LOC}, \mathcal{H}\rangle \xrightarrow{(\texttt{th},\texttt{m},d^{in})\downarrow}_{\mathcal{P}} \langle \texttt{T}, \texttt{LOC}[\texttt{th} \leftarrow \texttt{loc}^{\texttt{th}}_{\texttt{init}}], \mathcal{H}\rangle}\;\texttt{P1} \qquad \frac{\langle \texttt{loc}, \mathcal{H}\rangle \xrightarrow{(\texttt{th},\texttt{m},d^{out})\uparrow}_{\texttt{th}} \langle \texttt{term}, \mathcal{H}\rangle}{\langle \texttt{T}, \texttt{LOC}, \mathcal{H}\rangle \xrightarrow{(\texttt{th},\texttt{m},d^{out})\uparrow}_{\mathcal{P}} \langle \texttt{T}, \texttt{LOC}[\texttt{th} \leftarrow \texttt{term}], \mathcal{H}\rangle}\;\texttt{P2}$$

$$\frac{\langle \texttt{loc}, \mathcal{H}\rangle \xrightarrow{\lambda}_{\texttt{th}} \langle \texttt{loc}', \mathcal{H}'\rangle}{\langle \texttt{T}, \texttt{LOC}, \mathcal{H}\rangle \xrightarrow{\varepsilon}_{\mathcal{P}} \langle \texttt{T}, \texttt{LOC}[\texttt{th} \leftarrow \texttt{loc}'], \mathcal{H}'\rangle}\;\texttt{P3}$$

$$\frac{\texttt{view} \models cnd\ , \texttt{m}(d^{in}, d^{out}) = \texttt{view}(op)\ , e = \texttt{view}(se)\ , (\textbf{from}\ q\ \textbf{when}\ \bullet\ \textbf{provided}\ cnd\ \textbf{emit}\ op\ \textbf{broadcast}\ se\ \textbf{goto}\ q') \in R^{\texttt{th}}}{\texttt{view} : q \xrightarrow{\langle \texttt{th},\texttt{m}(d^{in},d^{out})\rangle | e!}_{\texttt{cntrl(th)}} q'\ .}\;\texttt{C1}$$

$$\frac{\texttt{view} \models cnd\ , \texttt{m}(d^{in}, d^{out}) = \texttt{view}(op)\ , e' = \texttt{view}(re)\ , (\textbf{from}\ q\ \textbf{when}\ re\ \textbf{provided}\ cnd\ \textbf{emit}\ op\ \textbf{goto}\ q') \in R^{\texttt{th}}}{\texttt{view} : q \xrightarrow{\langle \texttt{th},\texttt{m}(d^{in},d^{out})\rangle | \langle e',\texttt{ord}\rangle ?}_{\texttt{cntrl(th)}} q'}\;\texttt{C2}$$

$$\frac{\langle \texttt{loc}, \mathcal{H}\rangle \xrightarrow{\lambda}_{\texttt{th}} \langle \texttt{loc}', \mathcal{H}'\rangle\ , \neg\lambda^{\bullet}}{\langle q, \texttt{loc}, \mathcal{H}\rangle \xrightarrow{\varepsilon}_{\texttt{aug(th)}} \langle q, \texttt{loc}', \mathcal{H}'\rangle}\;\texttt{TC1} \qquad \frac{\langle \texttt{loc}, \mathcal{H}\rangle \xrightarrow{(\texttt{th},\texttt{m},d^{out})\uparrow}_{\texttt{th}} \langle \texttt{loc}', \mathcal{H}'\rangle}{\langle q, \texttt{loc}, \mathcal{H}\rangle \xrightarrow{(\texttt{th},\texttt{m},d^{out})\uparrow}_{\texttt{aug(th)}} \langle q, \texttt{loc}', \mathcal{H}'\rangle}\;\texttt{TC2}$$

$$\frac{\langle \texttt{loc}, \mathcal{H}\rangle \xrightarrow{\lambda}_{\texttt{th}} \langle \texttt{loc}', \mathcal{H}'\rangle\ , \lambda^{\bullet}\ , \langle \texttt{loc}, \mathcal{H}\rangle : q \xrightarrow{\langle \texttt{th}, op\rangle | e'!}_{\texttt{cntrl(th)}} q'\ ,}{\langle q, \texttt{loc}, \mathcal{H}\rangle \xrightarrow{\langle \texttt{th}, op\rangle | e'!}_{\texttt{aug(th)}} \langle q', \texttt{loc}', \mathcal{H}'\rangle}\;\texttt{TC3}$$

$$\frac{\texttt{enabled}\,(\texttt{T}, \texttt{Q}, \texttt{LOC}, \mathcal{H}, e, \texttt{ord}, \texttt{th}_s) = \{\texttt{th}_1, \ldots, \texttt{th}_n\}\ , \forall i : 1 \le i \le n :\ \langle \texttt{LOC}(\texttt{th}_i), \mathcal{H}\rangle : \texttt{Q}(\texttt{th}_i) \xrightarrow{\langle \texttt{th}_i, op_i\rangle | \langle e,\texttt{ord}\rangle ?}_{\texttt{th}_i} q_i}{\langle \texttt{T}/\texttt{th}_s, \texttt{LOC}, \mathcal{H}\rangle : \texttt{Q} \overset{op_1 \cdots op_n | \langle e,\texttt{ord}\rangle ?}{\rightsquigarrow} \texttt{Q}[\texttt{th}_1 \leftarrow q_1, \ldots, \texttt{th}_n \leftarrow q_n]}\;\texttt{R}$$

$$\frac{\texttt{LOC}(\texttt{th}) = \texttt{loc}\ , \langle q, \texttt{loc}, \mathcal{H}\rangle \xrightarrow{\tau}_{\texttt{aug(th)}} \langle q, \texttt{loc}', \mathcal{H}'\rangle, \tau \in \{\varepsilon, (\texttt{th}, \texttt{m}, d^{out}) \uparrow\}}{\langle \texttt{T}, \texttt{Q}, \texttt{LOC}, \mathcal{H}\rangle \xrightarrow{\tau}_{\mathcal{Q}} \langle \texttt{T}, \texttt{Q}, \texttt{LOC}[\texttt{th} \leftarrow \texttt{loc}'], \mathcal{H}'\rangle}\;\texttt{Q1}$$

$$\frac{\langle q, \texttt{loc}, \mathcal{H}\rangle \xrightarrow{\langle \texttt{th}_s, op\rangle | e!}_{\texttt{aug(th}_s\texttt{)}} \langle q', \texttt{loc}', \mathcal{H}'\rangle\ , \langle \texttt{T}/\texttt{th}_s, \texttt{LOC}, \mathcal{H}\rangle : \texttt{Q} \overset{w_1 | \langle e,\texttt{before}\rangle ?}{\rightsquigarrow} \texttt{Q}'\ , \langle \texttt{T}/\texttt{th}_s, \texttt{LOC}, \mathcal{H}\rangle : \texttt{Q}' \overset{w_2 | \langle e,\texttt{after}\rangle ?}{\rightsquigarrow} \texttt{Q}''}{\langle \texttt{T}, \texttt{Q}, \texttt{LOC}, \mathcal{H}\rangle \xrightarrow{w_1 \bullet \langle \texttt{th}_s, op\rangle \bullet w_2}_{\mathcal{Q}} \langle \texttt{T}, \texttt{Q}''[\texttt{th}_s \leftarrow q'], \texttt{LOC}[\texttt{th}_s \leftarrow \texttt{loc}'], \mathcal{H}'\rangle}\;\texttt{Q2}$$

$$\frac{\langle s, \texttt{m}(x^{in}, x^{out}), s'\rangle \in \Delta^{\mathcal{O}}\ , \texttt{Val}\,(x^{in}) = d^{in}\ , \texttt{Val}\,(x^{out}) = d^{out}}{\langle s, \texttt{Val}\rangle \xrightarrow{\langle \texttt{th},\texttt{m}(d^{in},d^{out})\rangle}_{\mathcal{O}} \langle s', \texttt{Val}\rangle}\;\texttt{O1}$$

$$\frac{\forall x^{in}.x^{out}.s'.\ \neg\left(\langle s, \texttt{m}(x^{in}, x^{out}), s'\rangle \in \Delta^{\mathcal{O}}\ \wedge\ \texttt{Val}\,(x^{in}) = d^{in}\ \wedge\ \texttt{Val}\,(x^{out}) = d^{out}\right)}{\langle s, \texttt{Val}\rangle \xrightarrow{\langle \texttt{th},\texttt{m}(d^{in},d^{out})\rangle}_{\mathcal{O}} \langle s, \texttt{Val}\rangle}\;\texttt{O2}$$

$$\frac{\langle s, \texttt{Val}\rangle \xrightarrow{\texttt{m}(d^{in},d^{out})}_{\mathcal{O}} \langle s, \texttt{Val}\rangle\ , \texttt{Lin}\,(\texttt{th}) \in \{0,1\}\ , \texttt{Method}\,(\texttt{th}) = \texttt{m}}{\langle \texttt{T}, \langle s, \texttt{Val}\rangle\ , \texttt{Lin}, \texttt{RV}\rangle \xrightarrow{\langle \texttt{th},\texttt{m}(d^{in},d^{out})\rangle}_{\mathcal{M}} \langle \texttt{T}, \langle s, \texttt{Val}\rangle\ , \texttt{Lin}[\texttt{th} \leftarrow 1], \texttt{RV}[\texttt{th} \leftarrow d^{out}]\rangle}\;\texttt{M1}$$

$$\frac{\langle s, \texttt{Val}\rangle \xrightarrow{\langle \texttt{th},\texttt{m}(d^{in},d^{out})\rangle}_{\mathcal{O}} \langle s', \texttt{Val}\rangle\ , s' \neq s\ , \texttt{Lin}\,(\texttt{th}) \in \{0,1\}\ , \texttt{Method}\,(\texttt{th}) = \texttt{m}}{\langle \texttt{T}, \langle s, \texttt{Val}\rangle\ , \texttt{Lin}, \texttt{RV}\rangle \xrightarrow{\langle \texttt{th},\texttt{m}(d^{in},d^{out})\rangle}_{\mathcal{M}} \langle \texttt{T}, \langle s', \texttt{Val}\rangle\ , \texttt{Lin}[\texttt{th} \leftarrow 2], \texttt{RV}[\texttt{th} \leftarrow d^{out}]\rangle}\;\texttt{M2}$$

$$\frac{c \xrightarrow{\langle \texttt{th},\texttt{m}(d^{in},d^{out})\rangle}_{\mathcal{O}} \langle s^{\mathcal{O}}_{\texttt{acc}}, \texttt{Val}\rangle}{\langle \texttt{T}, c, \texttt{Lin}, \texttt{RV}\rangle \xrightarrow{\langle \texttt{th},\texttt{m}(d^{in},d^{out})\rangle}_{\mathcal{M}} \textbf{error}}\;\texttt{M3} \qquad \frac{\texttt{Lin}\,(\texttt{th}) = 0}{\langle \texttt{T}, c, \texttt{Lin}, \texttt{RV}\rangle \xrightarrow{(\texttt{th},\texttt{m},d^{out})\uparrow}_{\mathcal{M}} \textbf{error}}\;\texttt{M4}$$

$$\frac{\texttt{Lin}\,(\texttt{th}) = 2}{\langle \texttt{T}, c, \texttt{Lin}, \texttt{RV}\rangle \xrightarrow{\langle \texttt{th},\texttt{m}(d^{in},d^{out})\rangle}_{\mathcal{M}} \textbf{error}}\;\texttt{M5} \qquad \frac{\texttt{RV}\,(\texttt{th}) \neq d^{out}}{\langle \texttt{T}, c, \texttt{Lin}, \texttt{RV}\rangle \xrightarrow{(\texttt{th},\texttt{m},d^{out})\uparrow}_{\mathcal{M}} \textbf{error}}\;\texttt{M6} \qquad \frac{}{\textbf{error} \xrightarrow{e}_{\mathcal{M}} \textbf{error}}\;\texttt{M7}$$

$$\frac{\forall i : 0 \le i < n : c_i \xrightarrow{e_{i+1}}_{\mathcal{M}} c_{i+1}}{c_0 \xrightarrow{e_1 \cdots e_n}_{\mathcal{M}} c_n}\;\texttt{M8} \qquad \frac{c_1 \xrightarrow{w}_{\mathcal{Q}} c_1'\ , c_2 \xrightarrow{w}_{\mathcal{M}} c_2'}{\langle c_1, c_2\rangle \rightarrow_{\mathcal{S}} \langle c_1', c_2'\rangle}\;\texttt{QM}$$

Fig. 4: Inference rules of the semantics.

$\texttt{ord}$. The rule $\texttt{R}$ describes the effect of receiving a message that has been broadcast by a sender thread $\texttt{th}_s$. Each thread that can receive the message will do so, and the thread potentially will all change their controller state and emit an operation. Notice that the receiver threads are collected in a set and therefore the rule allows any ordering of the receiving threads. We are now ready to explain the rule $\texttt{Q2}$. The sender thread $\texttt{th}_s$ broadcast a message e_2 that will be received by other threads according to rule $\texttt{R}$. The sender thread and the receiver threads, all emit operations (e_1 and w respectively). Depending on the order specified in the specification of the controller, a receiving thread may linearize *before* or *after* $\texttt{th}_s$. Notice that the receiver threads only change the local states of their controllers and the state of the observer. An *initial configuration* of $\mathcal{Q} \in \mathcal{C}^{\mathcal{Q}}$ is of the form $\langle \texttt{T}, \texttt{Q}_{\texttt{init}}, \texttt{LOC}_{\texttt{init}}, \mathcal{H}_{\texttt{init}} \rangle$ where, for each $\texttt{th} \in \texttt{T}$, $\texttt{LOC}_{\texttt{init}}(\texttt{th})$ is the initial state of the controller of $\texttt{th}$, and $\texttt{LOC}_{\texttt{init}}(\texttt{th}) = \texttt{idle}$.

Observers. Let $\mathcal{O} = \left\langle S^{\mathcal{O}}, s^{\mathcal{O}}_{\texttt{init}}, \texttt{X}^{\mathcal{O}}, \Delta^{\mathcal{O}}, s^{\mathcal{O}}_{\texttt{acc}} \right\rangle$. A configuration c of $\mathcal{O}$ is a pair $\langle s, \texttt{Val} \rangle$ where $s \in S^{\mathcal{O}}$ is an observer state and $\texttt{Val}$ assigns a value to each register in $\mathcal{O}$. We use $\mathcal{C}^{\mathcal{O}}$ to denote the set of configurations of $\mathcal{O}$. The transition relation $\rightarrow_{\mathcal{O}}$ of the observer is described by the rules $\texttt{O1}$ and $\texttt{O2}$ of Fig. 4. Rule $\texttt{O1}$ states that if there is an enabled transition that is labeled by a given action then such a transition may be performed. Rule $\texttt{O2}$ states that if there is no such a transition, the observer remains in its current state. Notice that the rules imply that the register values never change during a run of the observer. An initial configuration of $\mathcal{O} \in \mathcal{C}^{\mathcal{O}}$ is of the form $\left\langle s^{\mathcal{O}}_{\texttt{init}}, \texttt{Val} \right\rangle$.

Monitors. The monitor $\mathcal{M}$ augments the observer. It keeps track of the observer state, and the sequence of operations and call and return actions generated by the threads of the augmented program. It reports an error if one of the following happens: (i) a thread terminates without having linearized, (ii) the parameters of call and return actions are different from those of the corresponding emitted operation, (iii) a thread linearizes although it has previously changed the state of the observer, or (iv) $\mathcal{Q}$ generates a trace which violates the specification of the data structure $\texttt{DS}$. The monitor keeps track of these conditions as follows. A *configuration* of $\mathcal{M}$ is either (i) a tuple $\langle \texttt{T}, c, \texttt{Lin}, \texttt{RV} \rangle$ where $\texttt{T}$ is a set of threads, c is an observer configuration, $\texttt{Lin} : \texttt{T} \mapsto \{0, 1, 2\}$, and $\texttt{RV} : \texttt{T} \mapsto \mathbb{Z} \cup \mathbb{F}$; or (ii) the special state **error**. For a thread $\texttt{th} \in \texttt{T}$, the values $0, 1, 2$ of $\texttt{Lin}(\texttt{th})$ have the following interpretations: 0 means that $\texttt{th}$ has not linearized yet, 1 means that $\texttt{th}$ has linearized but not changed the state of the observer, and 2 means that $\texttt{th}$ has both linearized and changed the state of the observer. Furthermore, $\texttt{RV}(\texttt{th})$ stores the value returned by $\texttt{th}$ the latest time it performed a linearization. In case c is of the first form, we define $\texttt{ThreadsOf}(c) = \texttt{T}$. We use $\mathcal{C}^{\mathcal{M}}$ to denote the set of configurations of $\mathcal{M}$. The rules $\texttt{M1}$ through $\texttt{M8}$ describe the transition relation $\rightarrow_{\mathcal{M}}$ induced by $\mathcal{M}$. Rule $\texttt{M1}$ describes the scenario when the monitor detects an operation $\texttt{m}(d^{in}, d^{out})$ performed by a thread $\texttt{th}$ such that $\mathcal{O}$ does not change its state. In such a case, the flag $\texttt{Lin}$ for $\texttt{th}$ is updated to 1 (the thread has linearized but not changed the state of the observer), and the latest value d^{out} returned by $\texttt{th}$ is stored $\texttt{RV}$. Rule $\texttt{M2}$ is similar except that the observer changes state and hence the flag $\texttt{Lin}$ is updated to 2 for $\texttt{th}$. Notice that in both rules, a premise is that $\texttt{th}$ has not changed the observer state previously (the flag $\texttt{Lin}$ for $\texttt{th}$ is different from 2.) Rules $\texttt{M3}$, $\texttt{M4}$, $\texttt{M5}$, and $\texttt{M6}$ describe the conditions (i), (ii), (iii), and (iv) respectively described above that lead to the error state. The rule $\texttt{M7}$ describes the fact that the error state is a sink, i.e., once

$\mathcal{M}$ enters that state then it will never leave it. Finally, rule M8 describes the reflexive transitive closure of the relation $\rightarrow_{\mathcal{M}}$. An initial configuration of $\mathcal{M} \in \mathcal{C}^{\mathcal{M}}$ is of the form $\langle \mathtt{T}, c_{\mathtt{init}}, \mathtt{Lin}_{\mathtt{init}}, \mathtt{RV}_{\mathtt{init}} \rangle$ where $c_{\mathtt{init}} \in c^{\mathcal{O}}_{\mathtt{init}}$ is an initial configuration of $\mathcal{O}$, $\mathtt{Lin}_{\mathtt{init}}(\mathtt{th}) = 0$ and $\mathtt{RV}_{\mathtt{init}}(\mathtt{th})$ is undefined for every thread $\mathtt{th} \in \mathtt{T}$.

Product. We use $\mathcal{S} = \mathcal{Q} \otimes \mathcal{M}$ to denote the system obtained by running $\mathcal{Q}$ and $\mathcal{M}$ together. A (initial) configuration of $\mathcal{S}$ is of the form $\langle c_1, c_2 \rangle$ where c_1 and c_2 are (initial) configurations of $\mathcal{Q}$ and $\mathcal{M}$ respectively such that $\mathtt{ThreadsOf}(c_1) = \mathtt{ThreadsOf}(c_2)$. We use $\mathcal{C}^{\mathcal{S}}$ to denote the set of configurations of $\mathcal{S}$. The induced transition relation $\rightarrow_{\mathcal{S}}$ is described by rule QM of Fig. 4. Intuitively, in the composed systems, the augmented program and the monitor synchronize through the actions they produce. For a configuration $c \in \mathcal{C}^{\mathcal{S}}$, we define $\mathtt{Post}(c) = \{c' \mid c \rightarrow_{\mathcal{S}} c'\}$. For a set C of system configurations, we define $\mathtt{Post}(C) = \bigcup_{c \in C} \mathtt{Post}(c)$. We say that that **error** is reachable in $\mathcal{S}$ if $c \rightarrow_{\mathcal{S}}{}^{*} \langle c', \mathbf{error} \rangle$ for an initial configuration c of $\mathcal{S}$ and configuration c' of $\mathcal{Q}$.

Verifying Linearizability. The correctness of this approach is established in the following theorem.

Theorem 1. *If* **error** *is not reachable in $\mathcal{S}$ then $\mathcal{P}$ is linearizable wrt.* DS.

Proof: To prove $\mathcal{P}$ is linearizable wrt. DS. We must establish that for any history h of $\mathcal{P}$, there exists a history h' such that h' is linearization of h and h' is valid wrt. DS. So, consider a history h of $\mathcal{P}$. From Condition (i) in the paragraph about monitors in section 5, h is a complete history. Then from Conditions (ii) and (iii), each action in h has a LP and whose call and return parameters are consistent with the corresponding emitted operation by the controller. Therefore, there exists a complete sequential history h' which is a permutation of h whose actions are ordered according to the order of their LPs in h. Therefore h' is linearization of h. From Condition (iv), h' is $valid$ wrt. DS. Therefore, h is $linearizable$ wrt. DS. □

6 Thread Abstraction

By Theorem 1, linearizability can be verified by establishing a reachability property for the product $\mathcal{S} = \mathcal{Q} \otimes \mathcal{M}$. This verification must handle the challenges of an unbounded number of threads, an unbounded heap, and unbounded data domain. In this section, we describe how we handle an unbounded number of threads by adapting the thread-modular approach [3], and extending it to handle global transitions where several threads synchronize, due to interaction between controllers.

Let a heap cell $\mathbb{c}$ be *accessible* from a thread th if it is reachable (directly or via sequence of next-pointers) from a global variable or local variable of $\mathtt{th}_1$. A *thread abstracted* configuration of $\mathcal{S}$ is a pair $\langle c^{ta}, \mathtt{private} \rangle$, where $c^{ta} \in \mathcal{C}^{\mathcal{S}}$ with $\mathtt{ThreadsOf}(c^{ta}) = \{\mathtt{th}_1\}$, such that each cell in the heap of c^{ta} is accessible from $\mathtt{th}_1$ (i.e., the heap has no garbage), and where private is a predicate over the cells in the heap of c^{ta}. For a system configuration $c^{\mathcal{S}}$, let $\alpha_{\mathtt{thread}}\left(c^{\mathcal{S}}\right)$ be the set of thread abstracted configurations $\langle c^{ta}, \mathtt{private} \rangle$, such that (i) c^{ta} can be obtained by removing all threads from $c^{\mathcal{S}}$ except one, renaming that thread to $\mathtt{th}_1$, and thereafter removing all heap cells that are not accessible from $\mathtt{th}_1$, (ii) $\mathtt{private}(\mathbb{c})$ is true iff in $c^{\mathcal{S}}$, the heap cell $\mathbb{c}$ is accessible only from the thread that is not removed. For a set C of system configurations, define $\alpha_{\texttt{thread}}(C) = \bigcup_{c^{\mathcal{S}} \in C} \alpha_{\texttt{thread}}\left(c^{\mathcal{S}}\right)$. Conversely, for a set C of thread abstracted configurations, define $\gamma_{\texttt{thread}}(C) = \left\{c^{\mathcal{S}} \mid \alpha_{\texttt{thread}}(\{C\}) \subseteq C\right\}$. Define the postcondition operation $\texttt{Post}^{ta}$ on sets of thread abstracted configurations in the standard way by $\texttt{Post}^{ta}(C) = \alpha_{\texttt{thread}}(\texttt{Post}(\gamma_{\texttt{thread}}(C)))$. By standard arguments, $\texttt{Post}^{ta}$ then soundly overapproximates $\texttt{Post}$.

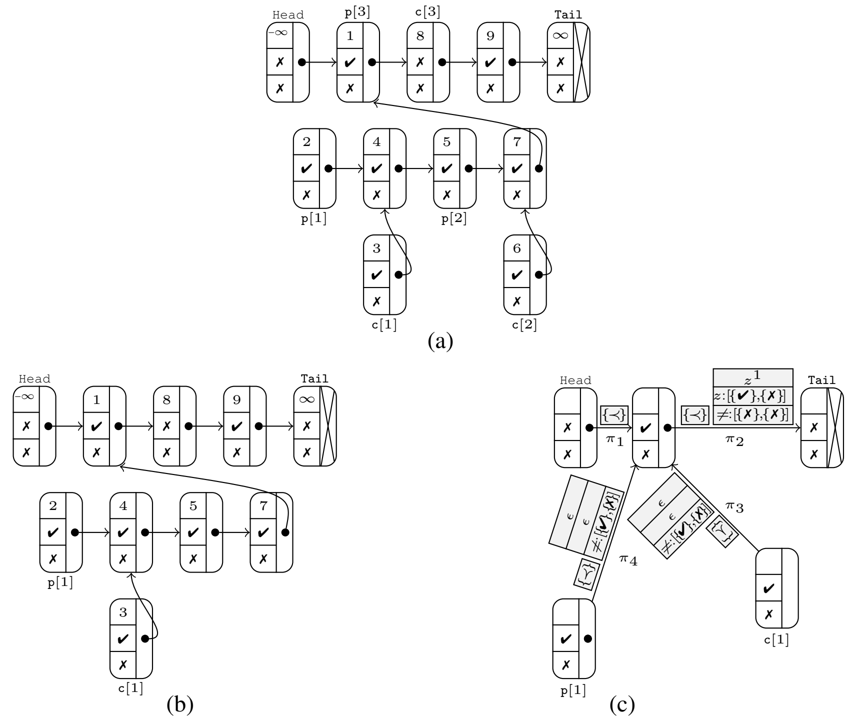

Fig. 5: Example of (a) heap state of Lazy Set algorithm, (b) its thread-abstracted version, and (c) symbolic version. The observer register x has value 9.

In Figure 5 (a) is a possible heap state of the Lazy Set algorithm of Figure 2. Each cell contains the values of `val`, `mark`, and `lock` from top to bottom, where ✔ denotes `true`, and ✗ denotes `false` (or *free* for `lock`). There are three threads, `1`, `2`, and `3` with pointer variable `p[i]` of thread `i` labeling the cell that it points to. The observer register x has value 9. Figure 5 (b) shows its thread abstraction onto thread `1`.

We observe that the concretization of a set of abstract configurations is a set of configurations with arbitrary sizes. However, as we argue below, it is sufficient to only consider such sizes up to $3 \cdot (\#\mathcal{O}) + 2$. The reason is that the thread abstraction encodes, for an arbitrary thread `th`, its view, i.e., its local state, the state of the heap, and the observer. In order to preserve soundness, we need not only to consider the transitions of `th`, but also how the transitions of other threads may change the states of the heap and the observer. Suppose that a different thread $\texttt{th}'$ performs a transition. We consider two cases depending on whether $\texttt{th}'$ sends a broadcast message or not. If not, it is sufficient for $\texttt{th}$ to see how $\texttt{th}'$ alone changes the heap and the observer. If $\texttt{th}'$ sends a message an arbitrary number of threads may receive it. Now note that only $\texttt{th}'$ may change the heap, while the receivers only change the state of the observer (Sec. 5). Also note that the values of the observer registers never change. The total effect is that $\texttt{th}'$ may change the state of the heap, while the other threads only change the state of the observer. Any such effect can be accomplished by only considering at most $3 \cdot (\#\mathcal{O}) + 2$ threads. The diameter $\#\mathcal{O}$ of $\mathcal{O}$ is the length of the longest simple path in $\mathcal{O}$. Formally, define $\gamma^k_{\texttt{thread}}(C) = \{c \mid (c \in \gamma_{\texttt{thread}}(C)) \cap |c| \leq k\}$, i.e., it is the set of configurations of in the concretization of C with at most k threads.

Lemma 1. *For a set of abstract configurations C, we have* $\texttt{Post}^{ta}(C) = \alpha_{\texttt{thread}}\left(\texttt{Post}\left(\gamma^k_{\texttt{thread}}(C)\right)\right)$ *where* $k = 3 \cdot (\#\mathcal{O}) + 2$.

As we mentioned above, when we perform helping transitions, the only effect of receiver threads is to change the state of the observer. When the receivers do not change the observer to a non-accepting state, we say that the transition is *stuttering* wrt. the observer. The system $\mathcal{S}$ is *stuttering* if all helping transitions of $\mathcal{S}$ are *stuttering*. This can be verified by a simple syntactic check of the observer and controller. In all the examples we have considered, the system turns out to be stuttering. For instance, the receiver threads in the `Lazy Set` algorithm are all performing stuttering `ctn` operations. Hence, the concretization needs only consider the sender thread, together with the thread whose view we are considering, reducing the number of threads to two.

Lemma 2. *For a set abstract configurations C of $\mathcal{S}$, if $\mathcal{S}$ is stuttering, then* $\texttt{Post}^{ta}(C) = \alpha_{\texttt{thread}}\left(\texttt{Post}\left(\gamma^2_{\texttt{thread}}(C)\right)\right)$.

7 Symbolic Shape and Data Abstraction

This section describes how we handle an unbounded heap and data domain in thread-abstracted configurations. We assume that each heap cell contains exactly one $\mathbb{Z}$-field.

The abstraction of thread-abstracted configurations we use is a variation of a well-known abstraction of singly-linked lists [19]. We explicitly represent only a finite subset of the heap cells, called the *relevant* cells. The remaining cells are summarized into linked-list segments of arbitrary length. Thus, we intuitively abstract the heap into a graph-like structure, whose nodes are relevant cells, and whose edges are the connecting segments. More precisely, a cell $\mathbb{c}$ (in a thread-abstracted configuration) is *relevant* if it is either (i) pointed to by a (global or local) pointer variable, or (ii) the value of $\mathbb{c}_1.\texttt{next}$ and $\mathbb{c}_2.\texttt{next}$ for two different cells $\mathbb{c}_1$, $\mathbb{c}_2 \in \mathbb{C}$, one of which is reachable from some global variable, or (iii) the value of $\mathbb{c}'.\texttt{next}$ for some cell $\mathbb{c}' \in \mathbb{C}$ such that $\texttt{private}(\mathbb{c}')$ but $\neg\texttt{private}(\mathbb{c})$. Each relevant cell is connected, by a linked-list segment, to a uniquely defined next relevant cell. For instance, the thread abstracted heap in Figure 5(b) has 5 relevant cells: 4 cells that are pointed to by variables (`Head`, `Tail`, `p[1]`, and `c[1]`) and the cell where the lower list segment meets the upper one (the second cell in the top row). The corresponding symbolic representation, shown in Figure 5(c), contains only the relevant cells, and connects each relevant cell to its successor relevant cell. Note that the cell containing `4` is not relevant, since it is not globally reachable. Consequently, we do not represent the precise branching structure among cells that are not globally reachable. This is an optimization to limit the size of our symbolic representation.

Our symbolic representation must now represent (A) data variables and data fields of relevant cells, and (B) data fields of list segments. For (A) we do as follows.

1. Each data and lock variable, and each data and lock field of each relevant cell, is mapped to a domain which depends on the set of values it can assume: (i) variables and fields that assume values in $\mathbb{F}$ are mapped to $\mathbb{F}$, (ii) lock variables and fields are mapped to $\{\mathtt{th}_1, \textit{other}, \textit{free}\}$, representing whether the lock was last acquired by $\mathtt{th}_1$, by another thread, or is free, (iii) variables and fields that assume values in $\mathbb{Z}$ are mapped to $[\mathtt{X}^{\mathcal{O}} \mapsto \{=, \neq\}]$, representing for each observer register $x \in \mathtt{X}^{\mathcal{O}}$, whether or not the value of the variable or field equals the value of x.
2. each pair of variables or fields that assume values in $\mathbb{Z}$ is mapped to a subset of $\{<, =, >\}$, representing the set of possible relations between them.=

For (B), we define a domain of *path expressions*, which capture the following information about a segment.

- The possible relations between adjacent $\mathbb{Z}$-fields in the heap segment, represented as a subset of $\{<, =, >\}$, e.g., to represent that the segment is sorted.
- The set of observer registers whose value appears in some $\mathbb{Z}$-field of the segment. For these registers, the path expression provides (i) the order in which the first occurrence of their values appear in the segment; this allows, e.g., to check ordering properties between data values for stacks and queues, and (ii) whether there is exactly one, or more than one occurrence of their value.
- For each data and lock field, the set of possible values of that field in the heap segment, represented as a subset of the domain defined in case 1 above. This subset is provided both for the cells whose $\mathbb{Z}$-field is not equal to any observer register, and for the cells whose $\mathbb{Z}$-field is equal to each observer register.

To illustrate, in Figure 5(c), each heap segment is labeled by a representation of its path expression. The first component of each path expression is $\{<\}$, i.e., the segment is sorted. The path expression at the top right has as its second component the sequence x^1, expressing that there is exactly one cell whose $\mathbb{Z}$-field has the same value as the observer register x. The row x : $[\{✔\}, \{✗\}]$ expresses that the cells whose $\mathbb{Z}$-value is equal to that of x, have their `mark` field `true`, and their `lock` field `free`. The row $\neq$: $[\{✗\}, \{✗\}]$ expresses that the other cells have their `mark` field `false`, and their `lock` field `free`. The two lower path expressions express that no cell has a value equal to that of x, and summarize the possible values of fields.

The symbolic representation combines the above described shape and data representations, representing thread-abstracted configurations by *symbolic configurations*. These are tuples of form $\Phi = \langle \mathtt{I}, \mathtt{Val}^{\mathtt{v}}, \mathtt{nextrel}, \pi, \mathtt{Val}^{\mathtt{d}}, \mathtt{Val}^{\mathtt{r}}, \mathtt{private} \rangle$, where

- $\mathtt{I}$ is a finite set of *indices*, denoting the relevant cells of the heap,
- $\mathtt{Val}^{\mathtt{v}}$ maps each (global or local) pointer variable to an index in $\mathtt{I}$,

– $\mathtt{nextrel}$ maps each index i to $\mathtt{I} \cup \{\mathtt{null}, \bot\}$,
– π maps each index i to a path expression; intuitively, if the index i represents the relevant cell $\mathbb{c}$, then $\mathtt{nextrel}(i)$ represents the next relevant cell, and $\pi(i)$ summarizes the heap segment between $\mathbb{c}$ and the cell represented by $\mathtt{nextrel}(i)$,
– $\mathtt{Val}^{\mathtt{d}}$ maps data variables and fields of relevant cells to appropriate domains, and $\mathtt{Val}^{\mathtt{r}}$ maps pair of those that assume values in $\mathbb{Z}$ to a subset of $\{<, =, >\}$.
– $\mathtt{private}$ is a predicate on indices.

We define a satisfaction relation between thread-abstracted and symbolic configurations. A *symbolic representation* Ψ is a set of symbolic configurations. A set of thread-abstracted configurations then satisfies a symbolic representation Ψ if each of its thread-abstracted configuration satisfies some symbolic configuration in Ψ.

Symbolic Postcondition Computation It remains to define a symbolic post operation $\mathtt{Post}^{symb}$ on symbolic representations that reflects $\mathtt{Post}^{ta}$ on sets of thread-abstracted configurations. Given a set Ψ of symbolic configurations representing possible views of single threads, it first generates all ways of merging them to symbolic representations of combined views of k threads; thereafter computing their postconditions wrp. to the next statement in each thread; and finally projecting the results onto each of the k participating threads. As follows from Lemmas 1 and 2, it is sufficient to consider only bounded values of k, and indeed $k = 2$ is sufficient for all examples that we considered.

8 Experimental Results

Algorithms	Time (s)	Algorithms	Time (s)
MS two-lock queue [21]	0.32	Treiber stack [27]	0.18
MS lock-free queue [21]	21.07	HSY stack [14] ✓	83.89
Elimination queue [23] ✓	105.56	DGLM queue [9]	16.99
Vechev-CAS set [34] ✓	24.01	Optimistic set [16]	60.43
Vechev-DCAS set [34]	16.02	Lazy set [13] ✓	289
Harris lock-free set [11]	1512	O′Hearn set [24] ✓	12.01
Michael lock-free set [22]	110	HM lock-free set [16] ✓	462.01
Pessimistic set [16]	1.51	CCAS [12] ✓	0.04
Unordered set [35] ✓	2301	RDCSS [12] ✓	0.26

(a)

Algorithms	Time (s)	Algorithms	Time (s)
MS two-lock queue [21]	1.80	Treiber stack [27]	140.03
MS lock-free queue [21]	287	DGLM queue [9]	500
		HW queue [17]	6.99

(b)

Fig. 6: Experimental Results for verifying concurrent programs

Based on our framework, we have implemented a tool in OCaml, and used it for verifying 19 concurrent algorithms (both lock-based and lock-free) including two stacks, five queues, nine ordered sets, one unordered set and two CAS algorithms. The user needs to provide the program code and the set of controllers. The experiments were performed on a desktop 2.8 GHz processor with 8GB memory. The results are presented in Fig. 6, where running times are given in seconds. Fig. 6(a) provides the verification results for all algorithms except for HW queue with provided linearization policies, using the technique of controllers introduced in the paper. Fig. 6(b) provides the verification results for queues and stack without LP placement, using the technique adapted from [4,15]. All experiments start from the initial heap, and end either when the analysis reaches the fixed point or when a violation of linearizability is detected.

Helping. The algorithms marked by ✓ use helping. We run some of the algorithms with two different helping patterns (such as the ones described in Sec. 3), and report the execution time for each.

Arrays. Our tool does not currently support arrays, and hence we have transformed arrays to singly-linked lists in the algorithms that use the former.

Safety Properties. Our tool is also capable of verifying memory related safety properties such as the absence of null pointer dereferencing, dangling pointers, double-freeing, cycles, and dereferencing of freed nodes, as well as sortedness. In fact, for each algorithm, the time reported in Fig. 6 is the sum of the times taken to show linearizability and all the properties mentioned above.

Running Times. As can be seen from the table, the running times vary in the different examples. This is due to the types of shapes that are produced during the analysis. For instance, the `CCAS`, `RDCSS`, stack and queue algorithms without an elimination produce simple shape patterns and hence they have shorter running times. Several features may make shapes more complex, such as insertion/removal of elements in the middle of a list in the ordered set algorithms, and having two linked lists (instead of one) in the elimination queue and stack algorithms. Also, the unordered set algorithm generates complex shapes since it allows physically removed elements to re-appear in the set.

Error Detection In addition to establishing correctness of the original versions of the benchmark algorithms, we tested our tool with intentionally inserted bugs. For example, we emitted broadcast messages in the controllers or inserted bugs into the codes of algorithms. In all cases, the tool, as expected, successfully detected and reported the bug. In the `Lazy Set` algorithm, when we emitted rule ρ_7 in Fig. 3, the tool reported an error. As another example, when we removed the statement in line 4 of the add method in the `Lazy Set` algorithm, the tool also reported an error.

9 Conclusions

We have presented a uniform framework for automatically verifying linearizability of singly-linked list-based concurrent implementations of sets, stacks, and queues, annotated with linearization policies. Our contributions include a novel formalism for specifying linearization policies with non-fixed LPs, an extension of thread-modular reasoning, and an extension of existing symbolic representations of unbounded singly-linked-list structures containing data from an unbounded domain. We have verified all linearizable singly-linked list-based implementations known to us in the literature. In the future, we intend to extend the framework to more complex data structures.

Acknowledgments We thank the reviewers for helpful comments. This work was supported in part by the Swedish Foundation for Strategic Research within the ProFuN project, and by the Swedish Research Council within the UPMARC centre of excellence.

References

1. P. A. Abdulla, F. Haziza, L. Holík, B. Jonsson, and A. Rezine. An integrated specification and verification technique for highly concurrent data structures. In *TACAS*, volume 7795 of *LNCS*, pages 324–338, 2013.
2. D. Amit, N. Rinetzky, T. Reps, M. Sagiv, and E. Yahav. Comparison under abstraction for verifying linearizability. In *CAV'07*, volume 4590 of *LNCS*, pages 477–490, 2007.
3. J. Berdine, T. Lev-Ami, R. Manevich, G. Ramalingam, and S. Sagiv. Thread quantification for concurrent shape analysis. In *CAV'08*, volume 5123 of *LNCS*, pages 399–413, 2008.
4. A. Bouajjani, M. Emmi, C. Enea, and J. Hamza. On reducing linearizability to state reachability. In *ICALP*, volume 9135 of *LNCS*, pages 95–107, 2015.
5. A. Bouajjani, M. Emmi, C. Enea, and J. Hamza. Verifying concurrent programs against sequential specifications. In *ESOP*, volume 7792 of *LNCS*, pages 290–309, 2015.
6. R. Colvin, L. Groves, V. Luchangco, and M. Moir. Formal verification of a lazy concurrent list-based set algorithm. In *CAV*, volume 4144 of *LNCS*, pages 475–488, 2006.
7. J. Derrick, B. Dongol, G. Schellhorn, B. Tofan, O. Travkin, and H. Wehrheim. Quiescent consistency: Defining and verifying relaxed linearizability. In *FM*, volume 8442 of *LNCS*, pages 200–214, 2014.
8. S. Doherty, D. Detlefs, L. Groves, C. Flood, V. Luchangco, P. Martin, M. Moir, N. Shavit, and G. Steele Jr. DCAS is not a silver bullet for nonblocking algorithm design. In *SPAA'04*, pages 216–224. ACM, 2004.
9. S. Doherty, L. Groves, V. Luchangco, and M. Moir. Formal verification of a practical lock-free queue algorithm. In *FORTE'04*, volume 3235 of *LNCS*, pages 97–114, 2004.
10. C. Dragoi, A. Gupta, and T. A. Henzinger. Automatic linearizability proofs of concurrent objects with cooperating updates. In *CAV*, volume 8044 of *LNCS*, pages 174–190, 2013.
11. T. L. Harris. A pragmatic implementation of non-blocking linked-lists. In *DISC*, pages 300–314, 2001.
12. T. L. Harris, K. Fraser, and I. A. Pratt. A practical multi-word compare-and-swap operation. In *DISC,*, pages 265–279, 2002.
13. S. Heller, M. Herlihy, V. Luchangco, M. Moir, W. N. S. III, and N. Shavit. A lazy concurrent list-based set algorithm. In *OPODIS*, pages 3–16, 2005.
14. D. Hendler, N. Shavit, and L. Yerushalmi. A scalable lock-free stack algorithm. *J. Parallel Distrib. Comput.*, 70(1):1–12, 2010.
15. T. Henzinger, A. Sezgin, and V. Vafeiadis. Aspect-oriented linearizability proofs. In *CONCUR*, volume 8052 of *LNCS*, pages 242–256, 2013.
16. M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.
17. M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

18. H. Liang and X. Feng. Modular verification of linearizability with non-fixed linearization points. In *PLDI*, pages 459–470. ACM, 2013.
19. R. Manevich, E. Yahav, G. Ramalingam, and S. Sagiv. Predicate abstraction and canonical abstraction for singly-linked lists. In *VMCAI*, LNCS, pages 181–198, 2005.
20. M. Michael and M. Scott. Correction of a memory management method for lock-free data structures. Technical Report TR599, University of Rochester, Rochester, NY, USA, 1995.
21. M. Michael and M. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *PODC*, pages 267–275, 1996.
22. M. M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *SPAA*, pages 73–82, 2002.
23. M. Moir, D. Nussbaum, O. Shalev, and N. Shavit. Using elimination to implement scalable and lock-free FIFO queues. In *SPAA*, pages 253–262, 2005.
24. P. W. O'Hearn, N. Rinetzky, M. T. Vechev, E. Yahav, and G. Yorsh. Verifying linearizability with hindsight. In *PODC*, pages 85–94, 2010.
25. G. Schellhorn, J. Derrick, and H. Wehrheim. A sound and complete proof technique for linearizability of concurrent data structures. *ACM Trans. Comput. Log.*, 15(4):31:1–31:37, 2014.
26. G. Schellhorn, H. Wehrheim, and J. Derrick. How to prove algorithms linearisable. In *CAV*, volume 7358 of *LNCS*, pages 243–259, 2012.
27. R. Treiber. Systems programming: Coping with parallelism. Technical Report RJ5118, IBM Almaden Res. Ctr., 1986.
28. A. J. Turon, J. Thamsborg, A. Ahmed, L. Birkedal, and D. Dreyer. Logical relations for fine-grained concurrency. In *POPL '13*, pages 343–356, 2013.
29. V. Vafeiadis. *Modular fine-grained concurrency verification*. PhD thesis, University of Cambridge, 2008.
30. V. Vafeiadis. Shape-value abstraction for verifying linearizability. In *VMCAI*, volume 5403 of *LNCS*, pages 335–348, 2009.
31. V. Vafeiadis. Automatically proving linearizability. In *CAV*, volume 6174 of *LNCS*, pages 450–464, 2010.
32. P. Černý, A. Radhakrishna, D. Zufferey, S. Chaudhuri, and R. Alur. Model checking of linearizability of concurrent list implementations. In *CAV*, volume 6174 of *LNCS*, pages 465–479, 2010.
33. M. Vechev, E. Yahav, and G. Yorsh. Experience with model checking linearizability. In *SPIN*, volume 5578 of *LNCS*, pages 261–278, 2009.
34. M. T. Vechev and E. Yahav. Deriving linearizable fine-grained concurrent objects. In *PLDI*, pages 125–135, 2008.
35. K. Zhang, Y. Zhao, Y. Yang, Y. Liu, and M. F. Spear. Practical non-blocking unordered lists. In *DISC*, pages 239–253, 2013.
36. H. Zhu, G. Petri, and S. Jagannathan. Poling: SMT aided linearizability proofs. In *CAV*, volume 9207 of *LNCS*, pages 3–19, 2015.