Poling: SMT Aided Linearizability Proofs

He Zhu, Gustavo Petri, and Suresh Jagannathan

Purdue University

Abstract. Proofs of linearizability of concurrent data structures generally rely on identifying linearization points to establish a simulation argument between the implementation and the specification. However, for many linearizable data structure operations, the linearization points may not correspond to their internal static code locations; for example, they might reside in the code of another concurrent operation. To overcome this limitation, we identify important program patterns that expose such instances, and describe a tool (Poling) that automatically verifies the linearizability of implementations that conform to these patterns.

1 Introduction

Linearizability [13] is the de facto correctness condition for the implementation of concurrent data structures. In a nutshell, linearizability establishes an observational equivalence between a multi-step fine-grained implementation, and an atomic coarse-grained specification of the data structure [7]. Thus, linearizability implies that each *operation* of a data structure implementation can be considered as executing atomically with respect to other concurrent operations.

While the definition of linearizability is intuitively simple, its proofs tend to be complex, and oftentimes depend on complicated simulation relations between the *abstraction* and the *implementation* (eg. [3,18]). A common strategy to define linearizability simulations is to identify program points in the implementation of the data structure – known as *linearization points* (LP) – upon whose execution an operation can be considered to have happened atomically [21]. Technically, a linearization point indicates in a weak simulation argument a unique and atomic step in the execution of the implementation at which the specification and the implementation match their behaviors.

In this paper, we present a lightweight technique which reduces this complex task into a property checking problem. In our approach, the abstract specification of a data structure operation is defined using *recursive functions* on heaplets (assertions describing a portion of the heap). We model these assertions as a set of memory locations following [17], enabling the use of SMT solvers to discharge our proof obligations. For example, at a linearization point (LP) of a *stack push* method, the *set of locations conforming the stack* after the execution of LP, should be equal to the set of locations conforming the stack before the push, plus an additional location containing the new value pushed. The validity of the formulae relating these two states of this linearizability argument (over *sets*) is decidable because they can be translated to quantifier-free formulas over integers and functions, and can be solved by using SMT solvers.

Tools like Cave [22] are succesful at automatically proving linearizability for data structures where the linearization points – static program locations – can be affixed to static program locations within each operation implementation. Unfortunately, a large class of lock-free linearizable data structures resist proofs by identifying such linearization points [3,11,16]. This should not be surprising since the definition of linearizability only requires the existence of a linearization for each invocation, which is intrinsically a *semantical* property of the implementation. Such a linearization needs not always correspond to the execution of predefined instructions within the code of the operation implementation. Our work extends previous efforts on the automation of linearizability proofs leveraging internal linearization points [22], by identifying data structure implementations whose linearization points may reside in code belonging to a concurrent operation.

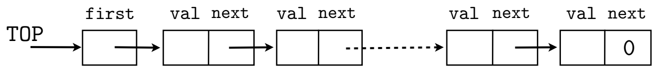
We do so by identifying two common *patterns*, occurring in fine-grained concurrent algorithms which cannot be verified using linearization points. Our ideas extend the state of the art in automatic linearizability verification by extending Cave [22] beyond linearization points. To the verification of linearizability by linearization points, Poling adds the following three notable features.

- Firstly, Poling converges faster than Cave in all the benchmarks that both tools can handle. This can be attributed to the fact that Poling reasons about separation logic (SL) verification conditions by interpreting them as *sets* of memory locations, following [17], with efficient SMT solvers.
- Secondly, Poling can verify linearizability for implementations that use *helping* [14], a mechanism that allows an operation performed by one thread to be completed by a concurrent operation of a different thread.
- Finally, Poling is sensitive to *hindsight* [16], a specific pattern in which the linearization of operations that do not update the state (called *pure*) can only be established *a posteriori* of their execution, and it might depend on other threads operations.

2 Overview

In the tradition of Cave, we focus on the linearizability of concurrent data structures implemented using linked lists. Some examples of such data structures are: stacks, queues and sets (see [12]). In our work we assume a garbage collected environment. Our approach is defined with respect to the *most general client* (MGC) program in which an unbounded number of threads interact though a single shared instance of the data structure [21], hence generalizing any possible client.

To introduce Poling, consider the implementation of a lock-free stack due to Treiber [19] shown in Figure 1.The stack is organized as a linked list of nodes containing a payload (`val`), and a `next` pointer to the following node in the list. The head of the list is saved in the field `first` of a shared global variable



```
int push (v) {                       int pop () {
  x = new Node(v);                     while(true) {
  while(true) {                          y = TOP->first;              // L0
     y = TOP->first;                     if (y==0) return EMPTY;
     x->next = y;                        else z = y->next;
     ⟨ if (TOP->first==y) {                   ⟨ if (TOP->first==y) {
          TOP->first = x;                          TOP->first = z;     // L
          break; } ⟩ }                             break; } ⟩ }
  return; }                            return y->val; }
```

Fig. 1. Treiber Stack.

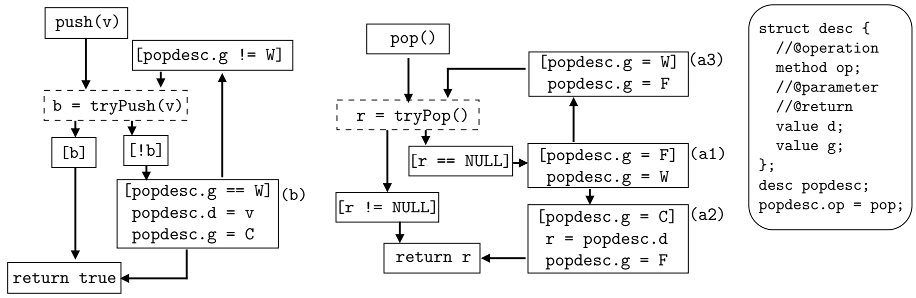
TOP, which serves as the synchronization point for competing threads. Thus, TOP->first points to the last-in node, when there is one, and it contains 0 if the stack is empty. The operations push and pop are presented. As customary, we group blocks that should be executed atomically with "⟨" and "⟩" (in this case they represent an inline of a cas instruction). The operation push creates a new node - it reads the head of the linked list, and then tries to atomically update it. If some other thread modifies TOP->first between the read and the write, the update is aborted and the process is reiterated. A similar argument applies to pop.

Since linearizability relates the implementation to a specification, the programmer must provide the specification. In the case of CAVE this is done by means of a simple language with primitives for the interpretation of lists and sets. The programmer can use these primitives to operationally specify the data structure. On the other hand, in POLING the specification is done declaratively using *recursive functions*. The *specification* of pop is defined declaratively as:

$$\begin{array}{lcl} \texttt{vals}(\sigma, \texttt{TOP->first}) = r :: \texttt{vals}(\sigma', \texttt{TOP->first}) & \Longleftrightarrow & \texttt{pop}() = r \\ \texttt{vals}(\sigma, \texttt{TOP->first}) = [\,] & \Longleftrightarrow & \texttt{pop}() = \texttt{Empty} \end{array}$$

In these equations, TOP and first are program variables. The keyword vals is a primitive used to abstractly refer to the contents of a data structure, where the argument σ represents the current state, and the argument TOP->first is a program expression pointing to the first element of the list. The state resulting from executing pop is represented by the symbol σ'. Finally, the return value of pop is denoted by r. Then, the specification of pop establishes the relation between the abstract state of the list before and after its execution. In this example, POLING interprets vals(σ, TOP->first) as a *recursive function* [17], defining the values stored in the locations reachable from TOP->first in σ. Recursive functions will be formalized in Section 3.

In Figure 1, the program point marked with L0 corresponds to the linearization point of pop when the list is empty. The specification for the non-empty case is satisfied at the program point L, because the set of values of locations reachable from TOP->first in a state σ (before the execution of the statement at L) equals to the set of values reachable in the resulting state (σ') plus the



```
struct desc {
  //@operation
  method op;
  //@parameter
  //@return
  value d;
  value g;
};
desc popdesc;
popdesc.op = pop;
```

Fig. 2. Elimination Based Stack. The formulae in square brackets are assumes and each node denotes an atomic block. Struct `desc` is an annotated descriptor.

element pointed by `y`, which is `pop`'s return value. Importantly no other point in `pop` changes the values of the reachable locations. POLING using the set of memory locations at `L` calls an SMT solver to automatically check whether the update to these locations, before and after `L`, respects the specification of `pop`. We present verification details in Section 4.

The example shown in Figure 2 (adapted from [6]) is a simplification of the HSY stack [10].The stack uses an underlying Treiber implementation, but improves its performance for high-contention scenarios through an elimination layer, an occurrence of the *helping mechanism*. The example allows pairs of overlapping instances of pop and push to exchange their values without accessing the list. Each operation attempts an iteration of the Treiber implementation, and if it fails it applies the helping mechanism. The process is iterated until the operation succeeds. Helping is implemented by storing a descriptor of a pending pop into the shared state variable `popdesc`. In the descriptor, if the value of `g` is W(aiting) in location $\mathtt{a}_1$, there is an invocation of pop ready to be helped. If the value of `g` is C(ollided), then a pair of push and pop invocations is in the process of exchanging the value through `g` in location `b`. The helping completes or gives up when `pop` sets the value of `g` back to F(ree), in locations $\mathtt{a}_2$ and $\mathtt{a}_3$, so that another instance of helping can happen. Importantly, the linearization of `pop` can happen in the code of a concurrent `push` in location `b` and, in location $\mathtt{a}_2$, the `pop` can witness that it has been helped because the transition through $\mathtt{a}_1$ to $\mathtt{a}_2$ can only be caused by an action that happened in location `b` of a `push` thread, where it was linearized.

We exploit this witnessing strategy in our linearization proofs. When an operation is linearized by another operation that helps it, we record the state at which the operation was linearized. Then, in the verification of the helped operation, we can use the recorded state as a witness to check that it was indeed helped by a concurrent operation. We observe that the witnessing is carried out through the descriptors (`popdesc`). To verify how a `push` can find and then help a pending `pop`, we allow programmers to use JML-style markers to annotate the descriptor data structure, indicating for an operation's name, the parameters and the return value (in Figure 2 denoted via @`operation`, @`parameter` and @`return` respectively). We cover the details of the verification in Section 4.

3 Formal Model

We define a data structure $\mathtt{D}$ as a pair $(\mathtt{D}_{\Sigma}, \mathtt{D}_{\mathtt{M}})$ consisting of $\mathtt{D}_{\Sigma}$, the domain of the data structure; and $\mathtt{D}_{\mathtt{M}}$ a set of method names, or *primitive operations* in the terminology of [13]. We call an *invocation* of the method $\mathtt{m}$ simply an *operation*, and use the metavariable $op_{\mathtt{m}}$ to range over the set $\mathcal{O}ps$ of all possible invocations. In turn, operations can be decomposed into a tuple $op_{\mathtt{m}} = (\mathtt{m}, t, v, r)$ containing a thread identifier $t \in \mathcal{T}id$, a vector with the arguments v, and if the operation is completed, a return value r.

Program Model. We omit a description of the program state and operational semantics of the programming language, assumed to be a standard first-order, shared-memory, concurrent, imperative language. We assume a set of states $\sigma \in \mathtt{D}_{\Sigma}$, and an operational semantics with execution steps between states labeled by events $ev \in Evs$ following the judgment: $\sigma \xrightarrow{t,ev} \sigma'$. Events capture data and control-flow actions (loads, stores, conditionals, etc.) with the addition of *invocation* and *response* events of operation $op_{\mathtt{m}}$, denoted $\mathsf{inv}(op_{\mathtt{m}})$ and $\mathsf{res}(op_{\mathtt{m}})$ respectively. These latter two kinds of events serve to delimit the "duration" of operation invocations. We assume the obvious extension of this step judgment to traces of events ranged by $\mathsf{tr} \in Evs^*$, denoted $\sigma \xrightarrow{\mathsf{tr}} \sigma'$, with the obvious inductive definition. Where unnecessary, we also omit the intermediate states.

Linearizability. Following [21], and without loss of generality, we assume a thread makes at most one invocation to an operation of the data structure. We then overload the notations for invocations and responses as $\mathsf{inv}(t)$ and $\mathsf{res}(t)$, for thread t. We define $\mathsf{history}(\mathsf{tr})$ to be the projection of invocation and response events in trace tr. Traces induce a partial order between operations. We say that a operation t precedes operation t' in tr ($t \prec_{\mathsf{tr}} t'$) as defined below[1]:

$$t \prec_{\mathsf{tr}} t' \iff \exists\ \mathsf{tr}_0\ \mathsf{tr}_1\ \mathsf{tr}_2,\ \mathsf{tr} = \mathsf{tr}_0 \cdot \mathsf{res}(t) \cdot \mathsf{tr}_1 \cdot \mathsf{inv}(t') \cdot \mathsf{tr}_2$$

We say that a history is *sequential* if each invocation event $\mathsf{inv}(t)$ is immediately followed by its corresponding response event $\mathsf{res}(t)$.

Definition 1 (Linearizable History). *A history h is linearizable w.r.t. the specification of a data structure $\mathtt{D}$ if, and only if, there exists a sequential trace h_s of $\mathtt{D}$ such that $\prec_h \subseteq \prec_{h_s}$, and the set of operations of h and h_s coincide.*

This notion is trivially lifted to implementations by requiring every implementation trace to have a linearizable history.

Set Specification. As hinted in Section 2, we abstract the data structure through an abstraction function: $\mathtt{vals}(\sigma, \mathtt{x})$, which for a given state σ represents the set of values stored in the data structure pointed by the program variable $\mathtt{x}$. This allows us to express the abstract data structure specification as an equation relating the initial state σ and final state σ'. Then for *Set* data structures implemented as an ordered list, where we assume that the global variable $\mathsf{head}(Set)$

[1] We eschew treating uncompleted invocations [13], which is a simple extension.

points to the beginning of the list, we write:

$$\begin{array}{rll} \texttt{vals}(\sigma', \mathsf{head}(Set)) & = \{v\} \cup \texttt{vals}(\sigma, \mathsf{head}(Set)) & \Longleftrightarrow \texttt{add}(v) = \text{true} \\ \texttt{vals}(\sigma', \mathsf{head}(Set)) \cup \{v\} & = \texttt{vals}(\sigma, \mathsf{head}(Set)) & \Longleftrightarrow \texttt{contains}(v) = \text{true} \\ v \notin \texttt{vals}(\sigma, \mathsf{head}(Set)) & & \Longleftrightarrow \texttt{contains}(v) = \text{false} \end{array}$$

Order Preserving Specifications. Using the concatenation operator (::) instead of union ($\cup$) as above, we can capture the behavior of data structures whose temporal behavior imposes an ordering. For example a stack `pop` should always return the "last" value pushed. Assuming that $\mathsf{head}(\mathsf{D})$ is the global variable pointing to the first element of the list, we can specify `pop` and `dequeue` – where we omit all other methods – as:

$$\begin{array}{rl} \texttt{vals}(\sigma, \mathsf{head}(Stk)) = r :: \texttt{vals}(\sigma', \mathsf{head}(Stk)) & \Longleftrightarrow \texttt{pop}() = r \\ \texttt{vals}(\sigma, \mathsf{head}(Queue)) = \texttt{vals}(\sigma', \mathsf{head}(Queue)) :: r & \Longleftrightarrow \texttt{dequeue}() = r \end{array}$$

State Abstraction. Since we use RGSep [21], we abstract the program state σ using separation logic formulae, denoted by a metavariable ψ. The syntax of these formulae is given by the following grammars, where $\approx$ ranges over binary relations of expression.

$$\begin{array}{lll} \psi & ::= & P * \boxed{P'} \mid \psi \vee \psi' \\ P & ::= & \Pi \wedge \Gamma \end{array} \qquad \begin{array}{lll} \Pi & ::= & \texttt{true} \mid \texttt{false} \mid E \approx E' \mid (\Pi \wedge \Pi') \\ \Gamma & ::= & \texttt{emp} \mid E_{tl} \mapsto \rho \mid \Gamma * \Gamma' \mid \texttt{lseg}_{tl,\rho}(E, E) \end{array}$$

We briefly describe these assertions (details can be found in [21]): 1. The formulae are given in disjunctive normal form, representing the different possible states reachable through different paths, 2. Each of the disjuncts has two parts, the local state, P predicating over the state local to the thread, and a shared state $\boxed{P'}$ – demarcated by a box [21] – which predicates over the state accessible to all threads, 3. Further, each of these states can be separated into a *pure* part Π, only concerned with stack allocated variables, and a *spatial* part Γ, which describes the heap, 4. Finally, heap assertions include the standard separation logic operators, where $E_{tl} \mapsto \rho$ denotes that the location E contains a record ρ (a mapping from field names to values), where the special field tl points to the next node in a linked list, and $\texttt{lseg}_{tl,\rho}(E, E')$ denotes a linked list segment starting at location E and ending at E'. The same convention applies to tl. All the nodes in this list segment share a same field-value mapping described in ρ.
Data Structure Abstraction. We use the method of [17] to discharge proof obligations about the state using an SMT solver. In [17] SL assertions are encoded as predicates on sets of memory locations. To that end, we define in Figure 3 a data structure abstraction function that takes an RGSep assertion, and transforms it into a set of values. This is the function $(\!|\mathsf{vals}|\!)(\psi_\sigma, \mathtt{x})$, which is the symbolic counterpart to the function $\texttt{vals}(\sigma, \mathtt{x})$ used for specifications. This recursive abstraction definition represents the set of elements that inhabit the data structure. The function $\mathsf{Find}(\Gamma, \mathsf{E})$, simply finds the syntactic atomic occurrence of a node or a list starting from E in the spacial formula Γ. We omit its trivial recursive definition over RGSep formulae. Here, the capitalized expressions `VALS` are uninterpreted functions in the logic of the underlying theorem provers we use [17]. Finally, notice that if we substitute the concatenation operator (::) for all the oc-

$$(\!|\mathsf{vals}|\!)(P * \boxed{\Pi \wedge \Gamma}, \mathsf{E}) \;=\; \mathsf{val}_{\mathsf{aux}}(\Gamma, \mathsf{Find}(\Gamma, \mathsf{E}))$$

where

$$\begin{aligned}
\mathsf{val}_{\mathsf{aux}}(\Gamma, \mathsf{E}_{tl} \mapsto \rho) &= \mathtt{VAL}(E) \;\cup\; (\!|\mathsf{vals}|\!)(\boxed{\Gamma}, \rho(tl)) \\
\mathsf{val}_{\mathsf{aux}}(\Gamma, \mathsf{lseg}_{tl,\rho}(\mathsf{E}, \mathsf{E}')) &= \mathtt{VALS}(E, E') \;\cup\; (\!|\mathsf{vals}|\!)(\boxed{\Gamma}, \mathsf{E}')
\end{aligned}$$

Fig. 3. Recursive Abstraction Definition

currences of the union operator (∪) in Figure 3, we obtain a recursive definition for lists instead of sets. A refined definition of our data structure abstraction is given in [25] (that also considers reachable locations that are logically detached).

4 Verification

Our verification begins after a pass of the frontend of Cave [22], which given the implementation of a data structure D, uses symbolic execution and shape analysis to produce D's data structure invariant [4] and RGSep [21] rely-guarantee actions. To aid the verification of the helping mechanism, Poling requires the programmer to instrument descriptors (as exemplified in Figure 2). A simple analysis could be implemented to instrument the descriptors automatically, or an interface could be implemented to indicate the thread descriptors, but we omit this unrelated step to simplify our development.

Central to our development are *LP (linearization point) functions.*

Definition 2 (Valid LP). *Assume a trace* $\mathsf{tr} : \sigma_i \xrightarrow{\mathsf{tr}} \sigma_f$ *of the data structure* D*, and a function* $\varrho : \mathcal{O}ps \to \mathtt{D}_\Sigma$*, mapping each operation op of* tr *to the state in* tr *exactly prior to the linearization of op. We say that* ϱ *is a* valid linearization point *function for* tr *with respect to an abstract specification* φ *if:*

1. *every operation* $op \in \mathsf{tr}$*, has an LP state (i.e.* $\varrho(op)$*) strictly between its invocation (*$\mathsf{inv}(op)$*) and its response (*$\mathsf{res}(op)$*). Formally:*
$$\exists\ \mathsf{tr}_{(1:4)},\ \sigma_i \xrightarrow{\mathsf{tr}} \sigma_f = \sigma_i \xrightarrow{\mathsf{tr}_1} \cdot \xrightarrow{\mathsf{inv}(op)} \cdot \xrightarrow{\mathsf{tr}_2} \varrho(op) \xrightarrow{\mathsf{tr}_3} \cdot \xrightarrow{\mathsf{res}(op)} \cdot \xrightarrow{\mathsf{tr}_4} \sigma_f$$
2. *only the states that linearize operations can affect the abstract data structure:*
$$\sigma_1 \notin \{\varrho(op) \mid op \in \mathsf{tr}\} \text{ and } \sigma_i \xrightarrow{\mathsf{tr}} \sigma_f = \sigma_i \xrightarrow{\mathsf{tr}_1} \sigma_1 \xrightarrow{ev} \sigma_2 \xrightarrow{\mathsf{tr}_2} \sigma_f \Rightarrow$$
$$\mathtt{vals}(\sigma_1, \mathsf{head}(\mathtt{D})) = \mathtt{vals}(\sigma_2, \mathsf{head}(\mathtt{D}))$$
3. *for each operation* $op \in \mathsf{tr}$*, we have that the LP state, and its subsequent state are related by the data structure specification* φ*. Formally, if the abstract specification of* $op = (\mathtt{m}, t, v, r)$ *is* φ*, for a trace of op:* $\sigma_i \xrightarrow{\mathsf{tr}} \sigma_f = \sigma_i \xrightarrow{\mathsf{tr}_1} \varrho(op) \xrightarrow{ev} \hat{\sigma} \xrightarrow{\mathsf{tr}_2} \cdot \xrightarrow{\mathsf{res}(op)} \sigma_r \xrightarrow{\mathsf{tr}_3} \sigma_f$ *where* $(\mathtt{m}(v) = \sigma_r(r))$*:*
(1) if op is the only operation linearized in $\varrho(op)$ *(i.e. there does not exist another* op'*, such that* $op' \neq op \wedge \varrho(op') = \varrho(op)$*) then*
$$[\varrho(op)/\sigma][\hat{\sigma}/\sigma']\varphi$$
(2) if there does exist one op' *whose abstract specification is* φ'*, such that* $op' \neq op \wedge \varrho(op') = \varrho(op)$ [2] *then, for any (ghost) state* σ_g*,*
$$([\varrho(op)/\sigma][\sigma_g/\sigma']\varphi \Rightarrow [\sigma_g/\sigma][\hat{\sigma}/\sigma']\varphi') \vee ([\varrho(op)/\sigma][\sigma_g/\sigma']\varphi' \Rightarrow [\sigma_g/\sigma][\hat{\sigma}/\sigma']\varphi)$$

[2] Definition 2 is defined for at most two operations linearized in one step. It can be extended to handle the case when finitely many operations are helped in one step.

In condition 3, to verify whether an operation op can be linearized when op is the single operation linearized in state $\varrho(op)$, we prove that the specification $[\varrho(op)/\sigma][\hat{\sigma}/\sigma']\varphi$ is respected by the execution step from the LP state $\varrho(op)$. In the substitution, the parametric pre state σ and post state σ' of the specification φ, are replaced with the LP state and its post state $\hat{\sigma}$. However, in the case of helping, many operations can be linearized in a single step (e.g. a `push` and a `pop` exemplified in Figure 2). We handle this case by introducing *ghost states*. For example, if an event $\varrho(op) \xrightarrow{ev} \hat{\sigma}$ linearizes two operations; say $\varrho(op) = \varrho(op')$, we check that there exists an intermediate state σ_g such that $\varrho(op)$ and σ_g satisfy the specification of op and σ_g and $\hat{\sigma}$ satisfy the specification of op', or viceversa. Intuitively this mediates and orders the linearization of the two operations.

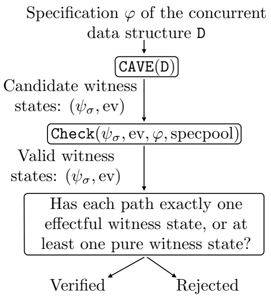
Theorem 1. *A data structure implementation* `D` *is linearizable with respect to an abstract specification* φ*, if for every trace* `tr` *of the implementation, there exists a* valid `D`-LP *function with respect to the specification* φ*.*[3]

Witness States. Definition 2 paired with Theorem 1 provides us with a way of checking linearizability by constructing valid LP functions. However, since LP functions map operations to states, it could be the case that the LP state of a certain operation may precede an event from a thread other than the one whose operation is linearized. The same argument applies when multiple operations are linearized in a single step. This is true for the helping mechanism of Figure 2. In this case, we define *witness states* as states from which one thread can make sure that it has been linearized, i.e. that a prior LP state ($\varrho(op)$) exists. For the simple case where the linearization point of an operation can be identified with its own program statement, the witness state is exactly the state before executing this statement. We prove linearizability by identifying *witness states*. In our approach, we distinguish effectful witness states where the abstract data structure is altered (like `L` in Figure 1), from pure (or effect-less) witness state that leave the abstract data structure intact (like `L0` in Figure 1).

Algorithm. We present our overall verification strategy in Figure 4. We first use heuristics derived from Cave to identify a set of states as candidate witness states (ψ_σ), paired with the event (ev) to be executed next; such events include all the memory reads or writes. The abstract states (ψ_σ) are obtained through the symbolic execution of Cave. The function `Check` verifies whether in a symbolic state ψ_σ the linearization of the operation can be witnessed w.r.t. its abstract specification φ. Programmers provide φ through the definition $\texttt{vals}(\sigma, \mathsf{head}(\texttt{D}))$ which is translated into the symbolic version $(\!|\mathsf{vals}|\!)(\psi_\sigma, \mathsf{head}(\texttt{D}))$.

Consider the pseudo-code of `Check` (given in Figure 4). In line 2 we symbolically execute the event ev from the state ψ_σ. To check if the abstract specification φ is fulfilled, in line 3, we replace the initial and final state with the ones obtained by symbolic execution and then unroll the definitions $(\!|\mathsf{vals}|\!)(\psi_\sigma, \texttt{head(D)})$ and $(\!|\mathsf{vals}|\!)(\psi_{\sigma'}, \texttt{head(D)})$ that are mentioned in the specification of the method, and encode them using first order logic (FOL) with set theories following [17]. We

[3] Theorems are proved in [25].


Specification φ of the concurrent data structure D
CAVE(D)
Candidate witness states: $(\psi_\sigma, \mathrm{ev})$
$\texttt{Check}(\psi_\sigma, \mathrm{ev}, \varphi, \mathrm{specpool})$
Valid witness states: $(\psi_\sigma, \mathrm{ev})$
Has each path exactly one effectful witness state, or at least one pure witness state?
Verified
Rejected

```
1: Check($\psi_\sigma$, $ev$, $\varphi$, specpool) =
2:   let $\psi_{\sigma'}$ = symb-exec($\psi_\sigma$, $ev$) in
3:     if (smt-check($\varphi[\sigma \mapsto \psi_\sigma, \sigma' \mapsto \psi_{\sigma'}]$)) then ✓
4:     else if (helps($\psi_\sigma$, $\psi_{\sigma'}$, specpool)) then ✓
5:     else if (helped($\psi_\sigma$, $\psi_{\sigma'}$, specpool)) then ✓
6:     else if (hindsight($\psi_{\sigma'}$)) then ✓
7:     else if ($(\!|\mathsf{vals}|\!)(\psi_\sigma, \mathtt{head(D)}) = (\!|\mathsf{vals}|\!)(\psi_{\sigma'}, \mathtt{head(D)})$)
8:       then ✗
9:       else (abort "not linearizable")
```

Fig. 4. General Framework. (The full pseudo code is provided in [25].)

feed the unrolled formulae to an SMT solver (the arguments v and return values r in the specification are also replaced with proper program variables, not shown here). Notice that satisfiability of quantifier-free formulas over sets/multisets with set union ($\cup$) is decidable. Concatenation ($::$) is considered as uninterpreted. If the formula is provable we have identified a witness state.

This strategy only applies to the case when linearization can be syntactically associated to instructions of the operation's own code, ie. LPs. Lines 4-6 deal with the cases when the linearization point might reside in operations of concurrent threads, which will be covered in the subsequent sections. If we are not able to prove φ in ψ_σ after these checks, at line 7, before reporting this state is not a candidate witness, we check that the abstract data structure did not change. We recall that we assume that only linearization events can modify the abstract state of the data structure. Hence, if the state did change, we abort the process in line 9, and report that the implementation could not be proved linearizable. After all the witness states are validated, following the strategy in [22], we use a simple data-flow analysis to verify each program path has either exactly one witness state or at least one pure witness.

Example 1. Consider the pop method of the stack implementation of Figure 1. With the method delineated above we obtain a symbolic state before the program point L (we only show the shared state):

$$\mathsf{TOP} \mapsto (\mathtt{first} : \mathsf{y}) \ * \ \mathsf{y} \mapsto \mathsf{z} \ * \ \mathsf{lseg}_{\mathtt{next}}(\mathsf{z}, 0)$$

This state is rendered from the successful test in the pop ($\mathtt{TOP} \mapsto \mathtt{first} = \mathtt{y}$). We will consider this state to be the witness state of pop (i.e. ψ_σ). The assignment of z to TOP->first would then be performed. To verify whether this implementation is faithful to the Stack specification, we first symbolically execute the instruction at L to render the post state after L (i.e. $\psi_{\sigma'}$):

$$\mathsf{TOP} \mapsto (\mathtt{first} : \mathsf{z}) \ * \ \mathsf{lseg}_{\mathtt{next}}(\mathsf{z}, 0) \ * \ \mathsf{y} \mapsto \mathsf{z}$$

According to the abstract specification of pop (Definition 3), we have to prove:

$$(\!|\mathsf{vals}|\!)(\psi_\sigma, \mathsf{TOP} \mapsto (\mathtt{first})) = r :: (\!|\mathsf{vals}|\!)(\psi_{\sigma'}, \mathsf{TOP} \mapsto (\mathtt{first}))$$

After unfolding, and substituting the special return symbol r with the actual return value, we obtain:

$$\mathtt{VAL}(\mathsf{y}) :: \mathtt{VALS}(\mathsf{z}, 0) = \mathtt{VAL}(\mathsf{y}) :: \mathtt{VALS}(\mathsf{z}, 0)$$

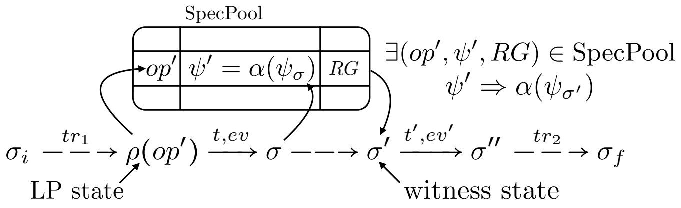

SpecPool
op'
$\psi' = \alpha(\psi_\sigma)$
RG
$\exists(op', \psi', RG) \in \text{SpecPool}$
$\psi' \Rightarrow \alpha(\psi_{\sigma'})$
$\sigma_i \overset{tr_1}{\dashrightarrow} \varrho(op') \xrightarrow{t,ev} \sigma \dashrightarrow \sigma' \xrightarrow{t',ev'} \sigma'' \overset{tr_2}{\dashrightarrow} \sigma_f$
LP state
witness state

Fig. 5. Spec Pool.

which is clearly provable. Moreover,

$$(\!|\mathsf{vals}|\!)(\psi_\sigma, \mathtt{TOP} \mapsto (\mathtt{first})) = (\!|\mathsf{vals}|\!)(\psi_{\sigma'}, \mathtt{TOP} \mapsto (\mathtt{first}))$$

holds for all the other states of `pop`. After all these verification steps, our method concludes that the program state before `L` is a valid effectful witness state.

Helping Verification. Consider a trace $\sigma_i \xrightarrow{\mathsf{tr}_1} \varrho(op') \xrightarrow{t,ev} \sigma \xrightarrow{\mathsf{tr}_2} \sigma_f$ corresponding to an execution of a data structure where op' is specified as $(\mathtt{m}, t', v, r)$. This trace is typical of algorithms implementing the helping mechanism. Here, the event (ev) that linearizes thread t' (the thread executing op') is taken by a concurrent thread t. A key ingredient of this pattern are the *descriptors*, which are used to keep information about ongoing invocations performed by different threads (c.f. `popdesc` in Figure 2). A thread can acknowledge its concurrent threads through its descriptors which are used by the concurrent threads to complete helping. In our proof, a thread under verification can retrieve the specifications of the other concurrent operations through their descriptors.

To exploit such descriptors, we add to the symbolic state, a set that represents helped operations. We call this set the *Spec Pool*, and use it to keep track of the synchronization entailed through the descriptors. Operations that perform helping are assumed to affect the Spec Pool. In the Spec Pool, each helped operation is equipped with (1) the condition that must hold upon its linearization and (2) the rely actions (used by the helper thread) that linearize it.

We provide a pictorial description of the process in Figure 5. Here we consider an event $\varrho(op') \xrightarrow{t,ev} \sigma$ from thread t which helps a concurrent operation op'. This event modifies the Spec Pool by inserting a tuple (op', $\alpha(\psi_\sigma)$, RG) indicating that op' has been helped at a state σ, and RG is the rely-guarantee actions extracted from this step [23], where α is a function that encodes a SL formula ψ_σ into a FOL formula. A verification step from op' (i.e. t') can observe the effects of t at state σ' (by checking a first order logical implication between $\alpha(\psi_\sigma)$ and $\alpha(\psi_{\sigma'})$). When verifying op' we also need to check that σ' can only be reached with the help of the rely action RG, absent of which σ' would be unreachable for t'. If the check is successful, σ' is considered as the witness state for op'. Figure 6 presents the definition of $\alpha(\psi)$. Simply stated, we keep the pure part of the SL formula and forget about the list segments. We encode the field-value mapping of a memory location ($\mathsf{E}_{\mathsf{tls}} \mapsto \rho$) into a conjunction of equations; each equation

$$\alpha(P * \boxed{\mathit{\Pi} \wedge \mathit{\Gamma}}) = \mathit{\Pi} \wedge \alpha_{\mathit{\Gamma}}(\mathit{\Gamma})$$

$$\alpha_{\mathit{\Gamma}}(\mathit{\Gamma}' * \mathit{\Gamma}'') = \alpha_{\mathit{\Gamma}}(\mathit{\Gamma}'') \wedge \begin{cases} \bigwedge_{\mathsf{pf} \in \mathsf{dom}(\rho)} \mathsf{pf}(\mathsf{E}) = \rho(\mathsf{pf}) & \text{if } \mathit{\Gamma}' = \mathsf{E}_{\mathsf{tls}} \mapsto \rho \ \& \ \mathtt{Isdesc}(E) \\ \mathsf{true} & \text{otherwise} \end{cases}$$

Fig. 6. Data Abstraction from SL formula to FOL formula.

encodes the value $\rho(\mathsf{pf})$ of a field pf on the location E. We encode E only if it is marked as a descriptor (local variables are implicitly existentially quantified).

Our approach hence reduces the problem of verifying linearizability to the following proof obligations: (a) we must check *how* an operation can be helped at the valid LP state in Figure 5 (i.e. linearized by another thread); this corresponds to the `helps` function in Figure 4, (b) for the thread that is helped, we must check the code that detects *whether* the operation has been helped at the valid witness state in Figure 5; this corresponds to the `helped` function in Figure 4, and (c) we must check that the helped operation is linearized exactly once.

For (a) we prove whether a given execution step in thread t can linearize another thread t' (with $t' \neq t$), directly following Definition 2. Let us consider how this proof works for `push` and `pop` of Figure 2. At the statement `b` of `push` we detect the descriptor `popdesc`, representing a concurrent `pop` thread. Assuming $\mathtt{head}(\mathit{Stk}) = \mathtt{TOP}$, we check:

$$\forall \psi_{gst},\ \Big(\ \llparenthesis \mathsf{vals} \rrparenthesis(\psi_{gst}, \mathtt{TOP}) = \mathtt{v} :: \llparenthesis \mathsf{vals} \rrparenthesis(\psi_{\sigma}, \mathtt{TOP}) \Rightarrow \\ \llparenthesis \mathsf{vals} \rrparenthesis(\psi_{gst}, \mathtt{TOP}) = \mathtt{popdesc.d} :: \llparenthesis \mathsf{vals} \rrparenthesis(\psi_{\sigma'}, \mathtt{TOP})\ \Big)$$

According to Definition 2, ψ_{gst} is a necessary intermediate state in the abstract data structure between the push and pop operations (it does not exist in the actual execution). The precedent is obtained from the `push`'s operation specification, with the argument substituted with the formal parameter `v`. The consequent is the specification of `pop`'s operation, substituting the return value for `popdesc.d`, known from the descriptor. Since the stack is not updated by the instruction `b`, after unrolling we can prove the above formula. Both operations are linearized in this step. After verifying that `pop` is helped, we create a Spec Pool item (op', ψ', RG), representing the result of helping at statement `b`:

$$\big(\mathtt{pop},\ \exists \mathtt{v}.\mathtt{popdes.g} = \mathtt{C} \wedge \mathtt{popdes.d} = \mathtt{v},\ \mathtt{popdes} \mapsto (\mathtt{g} : \mathtt{W}) \rightsquigarrow \mathtt{popdes} \mapsto (\mathtt{g} : \mathtt{C})\big)$$

As stated, ψ' is the data abstraction of state σ' while RG here only shows the key rely-guarantee action ([21]), i.e., `g` is changed from W(aiting) to C(ollided).

We prove (b) by showing that if a thread t' is linearized by another thread t, this fact is manifest through the Spec Pool. To prove that t' has been helped in a state $\psi_{\sigma'}$, we need to find a pool element (op', ψ', RG) such that the operation of t' is with the same method name to op', and prove with an SMT solver that:

$$(\psi' \Rightarrow \alpha(\psi_{\sigma'})) \wedge \neg(\psi' \wedge \alpha(\psi_{\sigma' \setminus RG}))$$

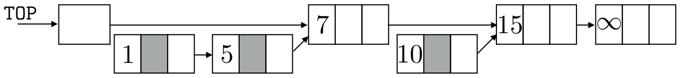
The first conjunct implies that op' may have been linearized by another thread, and the second one ensures that this could only result from other threads' interference RG. To check that this linearization could not have been possible without the interference from another thread's helps, we compute the state $\psi_{\sigma' \setminus RG}$

by symbolically executing the method (using CAVE) to the code location of σ' dropping the rely action RG. Then the conditions recorded in the Spec Pool (the conditions hold upon helping) must contradict $\psi_{\sigma' \setminus RG}$. Consider the path reaching the statement `a2` in `pop` in Figure 2. The conditions in the Spec Pool for `pop` (ψ') entail the data abstraction of state at `a2` (abstracted as `popdesc.g` = `C`). The only possible way to satisfy this assertion is by the rely RG, since originally we had `popdesc.g` = `W` at `a1`. We conclude that the `pop` was linearized by RG, the action made by a concurrent `push`. We also need to ensure a program path that witnesses helping must return the value of the return-field instrumented in the descriptor (e.g. return `popdesc.d` at `a2`).

We prove (c) by checking that an operation can only be helped once (e.g. helping for thread t' should be prohibited from state σ'' in the trace in Figure 5). We leave the details in [25].

Our verification procedure maintains the Spec Pool as part of the abstract state, and calls function `Check` of Figure 4 twice. In the first pass, we construct the Spec Pool by identifying helping scenarios; in the second pass, we exploit the Spec Pool to identify helped operations. Specifically, in Figure 4, when a candidate state fails to fulfill the specification at line 3, we attempt to prove (a), calling function `helps` at line 4, which identifies a set of descriptors in the state that enable helping. If successful, the corresponding pool items are created (in the first pass). Otherwise, at line 5, by calling function `helped`, we attempt proof obligation b to check if the operation has been helped (in the second pass).

Verification with Hindsight. This pattern is based on the Hindsight Lemma of [16]. In the interest of space we shall avoid presenting a full example like `Lazy_set` [9] (see [25]), which implements a set with an optimistic lock free *contains* operations using a linked list. As in the picture below, each node contains three fields: a value, a mark bit representing whether the item has been removed from the list (marked with grey), and a link (denoted as n) to the following node.



The fundamental invariants for this algorithm are: 1. the elements in the set are *ordered* for fast lookups through the lock-free *contains* method, 2. the elements *in* the list are all reachable from the `TOP` pointer, and are not marked, 3. removed elements are marked before being unlinked, and 4. the next pointer of a removed node never changes, hence it might still point to a node in the data structure, until this node is in turn removed. In the figure, the set contains the elements 7 and 15, but from the removed nodes we know that it contained the elements 1, 5 and 10 at some point in the past. A concurrent *contains* operation, which started before the elements were removed, may assume 1, 5 and 10 are still *contained.* Following [16], we shall call nodes that are reachable from `TOP` (including those that are marked) *backbone nodes* (e.g. 7 and 15). Conversely, nodes that cannot be reached from `TOP` are called *exterior nodes* (e.g. 1, 5 and 10).

Lemma 1. (Hindsight [16]). *Let* tr *be an execution of the set data structure presented above satisfying:*

1. *An exterior node never becomes a backbone node.*
2. *The successor of an exterior node will never change.*
3. *If the successor of a backbone node changes, the node remains a backbone node in the immediate following state.*

Then, for any states $\sigma_i = \mathsf{tr}(i)$, $\sigma_k = \mathsf{tr}(k)$ *such that* $0 \leq i \leq k < |\mathsf{tr}|$ *and for any nodes* u, v, w *such that* $u.n \mapsto v$ *is a backbone link in* σ_i, *and* $v.n \mapsto w$ *is a link (not necessarily in the backbone) in* σ_k, *there exists a state* $\sigma_j = \mathsf{tr}(j)$ *such that* $i \leq j \leq k$ *and* $v.n \mapsto w$ *is a backbone link in* σ_j.

Lemma 1 allows us to use exterior nodes and links in the current state to infer that there existed a past state in which the exterior nodes were in the backbone. Using this information we attempt to linearize the *contains* method, even if the found node is, in the current state, an exterior node. However, Lemma 1 cannot be used directly because, although an exterior link $v.n \mapsto w$ might be found in the current state, its premise, that a link $u.n \mapsto v$ was present in the backbone in a previous state, cannot be immediately established by looking at the current state in the symbolic execution. To resolve this problem we propose Theorem 2 which we exploit to automate the application of Lemma 1 in POLING.

Theorem 2. *If there is an exterior link* $v.n \mapsto w$ *in* σ, *a past state of* σ *in which the link is a backbone link exists provided the following conditions:*

1. *The premises of Lemma 1 hold, and*
2. $\mathtt{Reach}(head(\mathtt{D}), v)$ *can be proved in the sequential state* σ^{seq}.[4]

The sequential counterpart σ^{seq} of a state σ in a trace of an operation *op*, $\sigma_i \xrightarrow{\mathsf{inv}(op)} \cdot \xrightarrow{\mathsf{tr}_2} \sigma \xrightarrow{\mathsf{tr}_3} \cdot \xrightarrow{\mathsf{res}(op)} \sigma_f$, is obtained by execution from σ_i to σ dropping all the steps from *op*'s concurrent operations (executing *op* sequentially).

Note that the second condition ensures a temporal traversal to v (see [16]) and hence guarantees that $v.n \mapsto w$ was once a backbone link. The verification of this pattern (e.g. *contains* operation) is implemented in the `hindsight` function in line 6 in the `Check` function (Figure 4). In this function, exploiting Theorem 2, if an exterior link $v.n \mapsto w$ is found in a candidate state σ and $\mathtt{Reach}(\mathtt{head}(\mathtt{D}), v)$ holds in σ^{seq} (we compute $\psi_{\sigma^{seq}}$ by utilizing symbolic execution with an empty set of rely-guarantee actions in the implementation), we construct a past state σ_p and substitute it for σ when verifying the method's specification. If the verification succeeds, σ is a pure witness state for the verifying thread's linearization, that is, we can deduce the existence of LP state (σ_p) from witness state σ. We also customize the symbolic execution engine to verify all the three premises in Lemma 1: for each execution step $\sigma \xrightarrow{t,ev} \sigma'$, we collect exterior nodes (symbolically) in σ (ψ_σ), and verify that the step *ev* does not change their successors and they do not become reachable from `head(D)`; we also collect backbone nodes and check, if their successors are changed by *ev*, then they remain reachable from `head(D)` in σ' (ψ'_σ). If any of these checks fails, the `hindsight` function (Figure 4) returns false.

[4] `Reach` is the obvious reachability predicate over SL formulas.

$$\frac{\psi^{\mathsf{D}}_{Inv} : \bigvee_i \mathit{\Pi}_i \wedge \mathit{\Gamma}_i \qquad v.n \mapsto w \in \sigma}{\sigma_p \equiv \bigvee_i (\bigvee_{S \in \mathit{\Gamma}_i} \mathit{\Pi}_i \wedge \mathsf{exp}(v, w, S) * (\mathit{\Gamma}_i \backslash S))}$$

$$\begin{aligned} \mathsf{exp}(v, w, \mathsf{true}) &= v.n \mapsto w * \mathsf{true} \\ \mathsf{exp}(v, w, z \mapsto z') &= v = z \wedge v \mapsto z' \\ \mathsf{exp}(v, w, lseg(z, z')) &= \begin{array}{r} lseg(z, v) * v.n \mapsto w \\ * \, lseg(w, z') \end{array} \end{aligned}$$

Fig. 7. Hindsight Application Rule.

To reconstruct a past state σ_p as above, we introduce the Hindsight Lemma application rule in Figure 7. The rule is an adaptation of May-subtraction [23]. Intuitively, $\mathsf{May\text{-}Subtract}(P, Q)$ considers the ways in which an RGSep assertion Q can be removed from another assertion P. Our application rule works as $\mathsf{May\text{-}Subtract}(\psi^{\mathsf{D}}_{Inv}, v.n \mapsto w)$ to subtract an exterior link out of data structure invariant, and return the remaining state with the link added back. The auxiliary function `exp` (expose) considers all the ways in which v can be matched to a `node` or `linked list` assertion. Notice that since the only thing that is assumed in the rule (the hypotheses) is the data structure shape invariant ψ^{D}_{Inv} (derived from CAVE), the resulting symbolic state is an abstraction of an actual past state. The correctness of this rule is guaranteed by the proof of Theorem 2.

Limitations. Although POLING can automatically handle concurrent data structures with non-internal linearization points, we acknowledge that it cannot verify a class of concurrent data structure whose linearization points depend on future behaviors [11]. We expect to extend POLING to support this class of programs in the future.

5 Experimental Results

We evaluated POLING[5] on 11 examples, divided into 3 categories shown in the tables of Figure 8. In the first table we present algorithms provable using internal linearization points. We compare the times that CAVE (version 2.1) and POLING take to verify the algorithms and notice that for all these programs POLING outperforms CAVE. This can be attributed to our usage of SMT solvers following [17] to efficiently discharge linearizability proof obligations.

The second table presents algorithms falling under the hindsight pattern. We considered set implementation algorithms that perform an optimistic `contains` (or `lookup`) operation. Optimistic_set [16] traverses the list optimistically (without acquiring any locks, or synchronizing with other threads) to find a node. In contrast, Lazy_set [9], and its variant Vechev_CAS_set [24] use a bit for marking nodes before deletion.

The last table includes programs that implement the helping mechanism. Conditional compare-and-swap (CCAS) [20] is a simplified version of the well known RDCSS algorithm [8]. If a CCAS method finds a (thread) descriptor in its targeting shared memory location, they attempt to help complete the operation in that descriptor before performing its own. Finally, HSY_stack is the full HSY stack implementation [10]. Our running time in this complex example is comparable to a rewriting technique illustrated in [6]. As expected, CAVE cannot prove all the programs in the second and third categories.

[5] Project page: `https://www.cs.purdue.edu/homes/zhu103/poling/index.html`

Linearization Points

Program	CAVE	Poling
LockCoupling_set [12]	13.28s	4.01s
Vechev_DCAS_set [24]	73.90s	3.15s
2lock_queue [15]	2.91s	2.51s
Treiber [19]	0.28s	0.06s
MSqueue [15]	7.66s	1.12s
DGLMqueue [5]	9.40s	1.47s

Hindsight

Program	Poling
Vechev_CAS_set [24]	868.44s
Optimistic_set [16]	27.51s
Lazy_set [9]	321.78s

Helping

Program	Poling
CCAS [20]	0.82s
HSY_stack [10]	5.98s

Fig. 8. Experimental Results.

6 Related Work and Conclusion

Related Work. Most techniques on linearizability verification (e.g., [1,2]) are based on forward simulation arguments, and typically only work for methods with internal linearization points local to their own code locations. To deal with external linearization points, [3] proposed a technique limited to the case where only read-only operations may have external linearization points.

Complete backward simulation strategies have been proposed in [18]. However, they are often difficult to automate. Other methods combine both forward and backward simulations, using history and/or prophecy variables [21], instrumenting the program with auxiliary state [14], or using logical relations to construct relational proofs [20]. A general and modular proof strategy is proposed in [14], that, along with lightweight instrumentation, leverages rely-guarantee reasoning to manually verify algorithms with external linearization points. In contrast, our method exploits witness states to infer a proof automatically. The helping mechanism is also considered in [6] (which does not deal with the hindsight pattern) by rewriting the implementation so that all operations have their linearization points within their rewritten code. Our technique does not rely on rewritings because the relevant witness is found within the Spec Pool.

Our technique can be considered an adaptation of [17] which verifies sequential data structures using recursive definitions on heaplets. Similar to Poling, the automata based approach [1] is also a property checking algorithm which formalizes linearizability specifications as automata, and checks the cross-product of a symbolic encoding of the program with the specification automata for safety. The main difference between Poling and [1] resides in the verification of implementations with external linearization points.

Conclusion. We describe a procedure and a tool Poling that automatically checks the linearizability of fine-grained concurrent data structures. Poling abstracts concurrent data structure into sets of locations following [17] and considers linearizability verification as a property checking technique, which are efficiently solved with an SMT solver. Poling extends prior art by incorporating important concurrent programming patterns: algorithms using *helping*, and algorithms that can be proved using the *hindsight* lemma [16]. Our experimental results provide evidence of the effectiveness of our tool.

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*, 2013.
2. D. Amit, N. Rinetzky, T. Reps, M. Sagiv, and E. Yahav. Comparison under abstraction for verifying linearizability. In *CAV*, 2007.
3. J. Derrick, G. Schellhorn, and H. Wehriheim. Verifying linearizability with potential linearisation points. In *FM*, 2011.
4. D. Distefano, P. W. O'Hearn, and H. Yang. A local shape analysis based on separation logic. In *TACAS*, 2006.
5. S. Doherty, L. Groves, V. Luchangco, and M. Moir. Formal verification of a practical lock-free queue algorithm. In *FORTE*, 2004.
6. C. Dragoi, A. Gupta, and T. A. Henzinger. Automatic linearizability proofs of concurrent objects with cooperating updates. In *CAV*, 2013.
7. I. Filipovic, P. W. O'Hearn, N. Rinetzky, and H. Yang. Abstraction for concurrent objects. *Theor. Comput. Sci., 411*, 2010.
8. T. Harris, K. Fraser, and I. Pratt. A practical multi-word compare-and-swap operation. In *DISC*, 2001.
9. S. Heller, M. Herlihy, V. Luchangco, W. Moir, M an Scherer, and N. Shavit. A lazy concurrent list-based set algorithm. In *OPODIS*, 2005.
10. D. Hendler, N. Shavit, and L. Yerushalmi. A scalable lock-free stack algorithm. In *SPAA*, 2004.
11. T. A. Henzinger, A. Sezgin, and V. Vafeiadis. Aspect-oriented linearizability proofs. In *CONCUR*, 2013.
12. M. Herlihy and N. Shavit. *The Art of Multiporcessor Programming.* MorganKaufmann, San Francisco, 2008.
13. M. Herlihy and J. Wing. Linearizability: a correctness condition for concurrent objects. In *ACM TOPLAS, 12(3)*, 1990.
14. H. Liang and X. Feng. Modular verification of linearizability with non-fixed linearization points. In *PLDI*, 2013.
15. M. Michael and M. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *PODC*, 1996.
16. P. O'Hearn, N. Rinetzky, M. Vechev, E. Yahav, and G. Yorsh. Verifying linearizability with hindsight. In *PODC*, 2010.
17. X. Qiu, P. Garg, A. Stefanescu, and P. Madhusudan. Natural proofs for structure, data, and separation. In *PLDI*, 2013.
18. G. Schellhorn, H. Wehriheim, and J. Derrick. How to prove algorithms linearisable. In *CAV*, 2012.
19. P. Treiber. System programming: coping with parallelism. In *Technique Report RJ 5118, IBM Almaden Research Center*, 1986.
20. A. Turon, J. Thamsborg, A. Ahmed, L. Birkedal, and D. Dreyer. Logical relations for fine-grained concurrency. In *POPL*, 2013.
21. V. Vafeiadis. *Modular fine-grained concurrency verification.* PhD thesis, University of Cambridge, 2008.
22. V. Vafeiadis. Automatically proving linearizability. In *CAV*, 2010.
23. V. Vafeiadis. Rgsep action inference. In *VMCAI*, 2010.
24. M. Vechev and E. Yahav. Deriving linerizable fine-grained concurrent objects. In *PLDI*, 2008.

25. H. Zhu, G. Petri, and S. Jagannathan. POLING: Smt aided linearizability proofs. Technical report, Purdue Univsersity, 2015. `https://www.cs.purdue.edu/homes/zhu103/poling/tech.pdf`.