


Polynomial-Time Verification and Testing of Implementations of the Snapshot Data Structure

Gal Amram ✉ 

Ben Gurion University of the Negev, Beer-Sheva, Israel
IBM Research, Haifa, Israel

Avi Hayoun ✉

Ben-Gurion University of the Negev, Beer-Sheva, Israel

Lior Mizrahi

Ben-Gurion University of the Negev, Beer-Sheva, Israel

Gera Weiss ✉

Ben-Gurion University of the Negev, Beer-Sheva, Israel

Abstract

We analyze correctness of implementations of the snapshot data structure in terms of linearizability. We show that such implementations can be verified in polynomial time. Additionally, we identify a set of representative executions for testing and show that the correctness of each of these executions can be validated in linear time. These results present a significant speedup considering that verifying linearizability of implementations of concurrent data structures, in general, is EXPSPACE-complete in the number of program-states, and testing linearizability is NP-complete in the length of the tested execution. The crux of our approach is identifying a class of executions, which we call *simple*, such that a snapshot implementation is linearizable if and only if all of its simple executions are linearizable. We then divide all possible non-linearizable simple executions into three categories and construct a small automaton that recognizes each category. We describe two implementations (one for verification and one for testing) of an automata-based approach that we develop based on this result and an evaluation that demonstrates significant improvements over existing tools. For verification, we show that restricting a state-of-the-art tool to analyzing only simple executions saves resources and allows the analysis of more complex cases. Specifically, restricting attention to simple executions finds bugs in 27 instances, whereas, without this restriction, we were only able to find 14 of the 30 bugs in the instances we examined. We also show that our technique accelerates testing performance significantly. Specifically, our implementation solves the complete set of 900 problems we generated, whereas the state-of-the-art linearizability testing tool solves only 554 problems.

2012 ACM Subject Classification Software and its engineering → Formal software verification; Theory of computation → Concurrent algorithms

Keywords and phrases Snapshot, Linearizability, Verification, Formal Methods

Digital Object Identifier 10.4230/LIPIcs.DISC.2022.5

Related Version *Full Version*: https://github.com/hayounav/Thesis_experiments/blob/main/snapshot_verification_and_testing/Polynomial_time_verification_of_snapshot_implementations_DISC_.pdf

Supplementary Material *Software (Source Code)*: https://github.com/hayounav/Thesis_experiments; archived at [swh:1:dir:10be8bad714e0da40fec5d0a1a6aa34c550ccc33](https://swh.io/1/dir/10be8bad714e0da40fec5d0a1a6aa34c550ccc33)

Funding This research was partially funded by grant no. 2714/19 from the Israel Science Foundation and by the Lynn and William Frankel Center for Computer Science at Ben-Gurion University.

1 Introduction

As concurrency is very effective for accelerating the performance of computer programs, there is much scientific research and practical attention on the design, implementation, and verification of data structures that allow parallel access. We focus on the well-known *snapshot*



© Gal Amram, Avi Hayoun, Lior Mizrahi, and Gera Weiss;
licensed under Creative Commons License CC-BY 4.0
36th International Symposium on Distributed Computing (DISC 2022).

Editor: Christian Scheideler; Article No. 5; pp. 5:1–5:20



Leibniz International Proceedings in Informatics

LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

data structure which is an essential building block of distributed arrays [4, 7, 9]. This data structure allows asynchronous processes to write values to a shared array of single-writer registers, by executing `update` operations, and to take instantaneous snapshots of the array values, by executing `scan` operations. It is useful for allowing processes to share information while maintaining a correct joint view of the data.

For proving the correctness of implementations of the snapshot data structure, we consider the standard *linearizability* [31] condition. Roughly speaking, linearizability is the requirement that for every execution of a given implementation, the procedure executions can be ordered linearly such that (a) the resulting linear order is consistent with the definition of the data structure; (b) it preserves the precedence of procedure executions in time. In the specific case of snapshot, the definition of the data structure is that a sequential (linear) execution is correct if every `scan` operation reports the value written by the last `update` operation of each of the processes. Linearizability is widely accepted as a correctness criterion since, effectively, it formulates the requirement that procedure executions are seen to a user as if they were executed one after the other (i.e., atomically).

Automatic verification of linearizability is known to be computationally expensive. The verification of finite-state implementations is EXPSpace-complete in the number of program states [5, 29] and undecidable for infinite-state implementations [14]. Testing linearizability, i.e., deciding whether a given execution is linearizable, is NP-complete [28]. These complexities do not stop the community from pursuing effective verification and testing techniques, because it is very difficult to provide correct implementations of concurrent data structures; bugs have been found in both academic and deployed implementations [18, 20, 42]. These made it clear that there is an acute necessity for reliable verification and testing techniques.

One commonly used technique is the *linearization points* based verification approach, which often does not work in the case of snapshot. A linearization point of a procedure execution is an action that represents the moment at which the procedure “actually occurs”. Once fixed linearization points are identified, verifying linearizability becomes PSPACE-complete [14]. Unfortunately, snapshot implementations do not usually admit fixed linearization points (e.g., all twelve published implementations listed in [35] do not admit such points). Researchers also suggested using linearization points as an optimization: ask the user to provide them (whether fixed or conditional) and use this information to accelerate verification [3, 6, 13, 50]. However, practice shows that it is difficult to find and specify the linearization points of snapshot implementations, even in a conditional manner. One difficulty is that the linearization points of `scan` operations often belong to other, parallel, procedure executions (see [4, 11, 46]).

In this paper, we propose an effective polynomial-time technique for verifying snapshot implementations, and an effective linear-time technique for testing snapshot executions. The crux of our techniques is an optimization approach that exponentially reduces the number of reachable program states. Specifically, we prove that if an algorithm is data-independent [54] then, in order to verify its correctness, it suffices to consider only a small fraction of its executions which we call *simple*.

The simple executions that we focus on are those in which:

1. All but two processes invoke only `update(v_0)` and `scan` operations, where v_0 is the initial value of the array segments. In other words, $n-2$ of the n processes are not allowed to change the initial value in their segments;
2. Each of the two remaining processes may only change their data value once, to a predetermined value: it executes only `update(v_0)` and `scan` operations up to an arbitrary point in the execution, after which it transitions to executing only `update(v_1)` and `scan` operations, where $v_1 \neq v_0$ are fixed data values.

The focus on simple executions reduces the number of reachable states significantly, as $n-2$ entries of the array are essentially constants (see Section 3).

After showing that it is enough to verify the correctness of simple executions, we continue and show that every non-linearizable simple execution falls into one of three categories that we identify. Moreover, we show that each of these three possible bug patterns can be recognized by an automaton with at most n states and n^2 transitions (see Section 4). This enables verifying linearizability of snapshot implementations via a reachability check applied to the graph product of the implementation and the automata where the target states of the reachability are the automata's accepting states. As there are $O(n^2)$ combinations to choose the two excluded processes, snapshot implementations can be verified with this method in time $O(mn^4)$ where m is the number of reachable states via simple executions (which is significantly smaller than the number of reachable states via all executions). Furthermore, by feeding a simple execution to these automata, an execution of length l can be tested in time $O(l)$. As it is sufficient to consider simple executions, this effectively means that snapshot executions can be tested in linear-time (see Section 5).

We have implemented and evaluated the proposed verification and testing techniques and ran them against state-of-the-art tools. For verification, we compared with the PAT [44] model checker. The results show that our approach allows deeper exploration of implementations from the literature. This allowed us to detect 27 of 30 inserted bugs, compared to 16 found by the baseline method. Furthermore, we managed to verify an algorithm by Bowman [17] for three and four processes, whereas the baseline method failed to do so. For testing, we compared with the linearizability testing tool proposed by Lowe [41]. The results show that our testing technique is robust and scalable and that it can cope with much longer histories than the baseline (see Section 7).

Due to lack of space, we give proof sketches and skip technical details. We provide a full version with complete proofs, and means to reproduce the experiments in the paper supporting materials [49].

2 Preliminaries

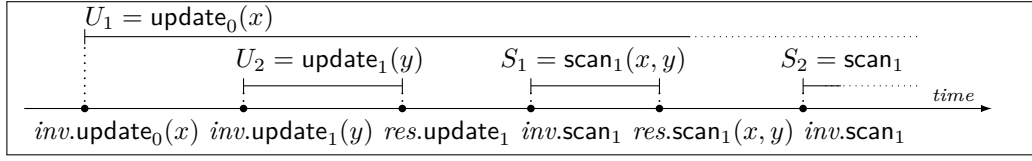
This section presents definitions and notations used throughout this paper. We provide further definitions, required for all complete proofs, and extended discussions in the supporting materials [49, Appendix A].

Let $Vals$ be an infinite set of abstract data values, and let $v_0 \in Vals$ be the distinguished value used to initialize the segments of a snapshot. For $n \in \mathbb{N}$, let p_0, \dots, p_{n-1} be processes. We model an execution of a snapshot algorithm by the processes as a sequence of actions. Among the actions the processes perform, we are interested in the invocations and responses of procedure executions. For process p_i and data values u, u_0, \dots, u_{n-1} , $inv.update_i(u)$, $res.update_i$, $inv.scan_i$, $res.scan_i(u_0, \dots, u_{n-1})$ are p_i -actions. Let Σ be the set of all such actions.

Throughout the paper, we refer to these actions using general terms such as: an update invocation, a scan response, a p_i -invocation etc., which are defined in a straightforward manner. For example, we may say that the action $res.scan_i(u_0, \dots, u_{n-1})$ is a scan response, or a p_i -action, etc.

A *history* is a word h over Σ that exhibits the following properties:

1. For every process p_i , the first p_i -action in h , if any, is a p_i -invocation.
2. For every p_i -update (respectively, scan) invocation in h , the following p_i -action in h , if any, is a p_i -update (respectively, scan) response.
3. For every p_i -response in h , the following p_i -action, if any, is a p_i -invocation.



■ **Figure 1** A linearizable history. U_2, S_1 are complete, while U_1, S_2 are pending ops.

An operation is an execution of an `update`/`scan` procedure. We identify operations in histories with their invocation and response actions. Operations that do not return are identified by their invocation alone.

A *complete operation* in a history $h = \alpha_0 \cdots \alpha_m$ is a pair of actions, (α_k, α_l) , where $k < l$, α_k is an `updatei` (respectively, `scani`) invocation, α_l is an `updatei` (respectively, `scani`) response, and there is no p_i -action in between. A *pending operation* in h is a single action, (α_k) , where α_k is a p_i -invocation, and there is no p_i -action that follows α_k in h .

Similarly to actions, we refer to operations using general terms: an operation O is, e.g., a p_i -operation, an `update` operation, a `scani(u_0, \dots, u_{n-1})` operation, etc. In a history h , for an `update(u)` operation U , we write $val_h(U) = u$, and for a `scan(u_0, \dots, u_{n-1})` operation S and $i < n$, we write $val_{h:i}(S) = u_i$ and $val_h(S) = (u_0, \dots, u_{n-1})$.

For a complete operation $A = (\alpha_k, \alpha_l)$ and an operation $B \in \{(\alpha_m, \alpha_t), (\alpha_m)\}$ in a history $h = \alpha_0 \alpha_1 \cdots$, we write $A <_h B$ if $l < m$. Clearly, $<_h$ is a partial order over the operations in h , in which pending operations are maximal elements. Figure 1 illustrates a two-process history with pending and complete operations.

Linearizability [31] is the standard correctness condition for concurrent data structures. Roughly speaking, a history h is linearizable if the partial ordering $<_h$ can be extended to a linear ordering that satisfies the sequential specification of the snapshot data structure. That is, each scan operation S returns in each entry i the value written by the maximal `updatei` operation that precedes it. The extension should include all complete operations, and each pending operation is either completed or omitted.

We now turn to define the linearizability condition formally:

► **Definition 1.** A history h is linearizable if it can be extended into a history h' by appending zero or more response events to h , such that there exists a linear ordering $\prec_{h'}$ of the complete operations in h' that satisfies the following conditions:

1. For $A, B \in h$, if $A <_h B$, then $A \prec_{h'} B$.
2. If $S \in h'$ is a scan operation and $U_i \in h'$ is the maximal `updatei` operation such that $U_i \prec_{h'} S$, then $val_h(U_i) = val_{h:i}(S)$. If no `updatei` operation precedes S in h' , then $val_{h:i}(S) = v_0$.

Any $\prec_{h'}$ that satisfies these conditions is said to be a linearization of h .

► **Example 2.** The history depicted in Figure 1 is linearizable by the order $U_1 \prec_h U_2 \prec_h S_1$. To obtain a linearization, we completed the pending operation U_1 , as its value is read by S_1 . However, we chose to omit the pending scan operation S_2 .

Our main goal is to analyze the linearizability of snapshot algorithms, defined as follows:

► **Definition 3 (Snapshot Linearizability).** A snapshot algorithm is linearizable if all of its histories are linearizable.

The data independence property, proposed by Wolper [54], roughly means that the behavior of an algorithm does not depend on the data values passed as arguments to the procedure executions. The formal definition employs the notion of a *renaming*: a function $f: Vals \rightarrow Vals$. An algorithm is data-independent if for every history h of the algorithm and a renaming f : (1) the history $f(h)$, obtained by replacing each data value u with $f(u)$, is also a history of the algorithm; and (2) if $h = f(h')$, then h' is a history of the algorithm. See full version [49] for more details.

Data independence is natural to assume, as snapshot implementations synchronize accesses to a shared resource and thus are expected to be value-agnostic. This is substantiated by all twelve different published implementations [4, 7–11, 26, 34–36, 46] listed in [35].

Finally, a history is *differentiated* if no two **update** operations in it were invoked with the same data value.¹ Abdulla et al. [1] showed that it is sufficient to consider *differentiated* histories to prove linearizability of data-independent algorithms.

3 Simple Histories

In this section, we identify a set of histories, which we name *simple*. We then prove that a data-independent snapshot implementation is linearizable if and only if all of its simple histories are linearizable. Therefore, this section shows that it is sufficient to consider only some histories to determine the linearizability of data-independent snapshot implementations.

In a simple history, the **update** operations are invoked with only two distinct values. The first is the initial value v_0 , and without loss of generality, we take some other $v_1 \in Vals$ as the second value. All but two processes invoke only **update**(v_0) and **scan** operations. The remaining two execute only **update**(v_0) and **scan** operations, and at some (possibly different) point, each of the two processes shifts to executing only **update**(v_1) and **scan** operations.

► **Definition 4** (Simple histories). *A history h of n processes is (i, j) -simple for $i < j < n$, if there are $r_i, r_j \in \mathbb{N}$ such that the following conditions hold:*

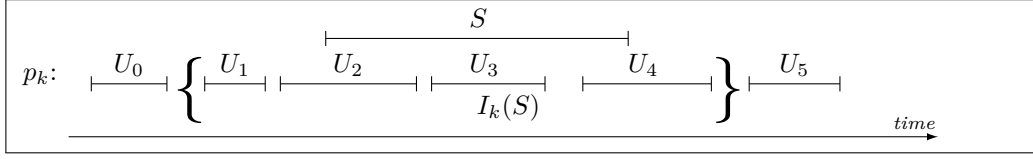
1. *Let U be the r -th update_i operation in h . If $r < r_i$, then U is an $\text{update}_i(v_0)$ operation, and if $r \geq r_i$, then U is an $\text{update}_i(v_1)$ operation.*
 2. *In the same way, let U be the r -th update_j operation in h . If $r < r_j$, then U is an $\text{update}_j(v_0)$ operation, and if $r \geq r_j$, then U is an $\text{update}_j(v_1)$ operation.*
 3. *Any update_k operation is an $\text{update}_k(v_0)$ operation, if $k \notin \{i, j\}$.*
- A history h is simple if it is (i, j) -simple for some $i < j < n$.*

We are ready to prove the sufficiency of focusing on simple histories. We provide a proof sketch below, and give a rigorous proof in the full version [49].

► **Theorem 5.** *A data-independent snapshot algorithm \mathcal{Snap} is linearizable if and only if all its simple histories are linearizable.*

Proof sketch. Anderson’s shrinking lemma identifies five properties that are equivalent to the linearizability of a snapshot history [8]. To prove the non-trivial direction of theorem 5 (‘if’), we assume that \mathcal{Snap} is not linearizable. Consider some non-linearizable differentiated history h . Since \mathcal{Snap} is not linearizable, h violates (at least) one of the shrinking lemma’s properties. Based on the violated property, we construct a renaming $f: Vals \rightarrow \{v_0, v_1\}$, and apply it to h to obtain a non-linearizable simple history. ◀

¹ For generating differentiated histories, a minor modification is required: we should allow different initial values for the array segments. See comment in full version [49, Appendix B]



■ **Figure 2** The k -th interval of S , $I_k(S)$.

► **Remark 6.** In the context of simple histories, scan operations return v_0 in all entries except for entries i and j . Thus, for the remainder of this paper, we use $res.scan_k(u_i, u_j)$ as shorthand for $res.scan_k(v_0, \dots, v_0, u_i, v_0, \dots, v_0, u_j, v_0, \dots, v_0)$.

► **Remark 7.** From this point on, for readability, we will use 0 and 1 instead of v_0 and v_1 , respectively.

4 A Simple Condition for the Linearizability of Simple Histories

In this section, we formulate three properties that are equivalent to the linearizability of an (i, j) -simple history. We then show that the negation of each property is regular, and present a construction of a matching automaton. Before providing our properties (in upcoming Theorem 10), we discuss each intuitively and explain why it is mandatory for linearizability.

Property 1: No Inversion. Assume that a scan operation S_1 returns 0 at the i th entry (for example), while S_2 returns 1. This indicates that S_2 read a more recent value from the i th segment. Hence, in any linearization, S_1 must precede S_2 . As the same reasoning goes for the j th entry, it is forbidden for S_1 to return 0 and 1 at the i th and j th entries, while S_2 returns the opposite values.

Property 2: Non-Decreasing. If a scan operation S_1 precedes a scan operation S_2 , then S_2 must obtain more recent values from all array segments. Therefore, it is forbidden for S_1 to return 1 in entry $k \in \{i, j\}$, while S_2 returns 0 in its k th entry.

Property 3: Appropriateness. We require that for each scan operation there are “appropriate” update operations, U_i by p_i and U_j by p_j , that we can linearize before S . “Appropriate” means that the next three conditions hold.

First condition. The timings of the update operations must not prevent them from being linearized before S . For example, they must not succeed S . Formally, we require that they belong to the *interval of S* , defined below and illustrated in Figure 2:

► **Definition 8.** Let S be a scan operation in a history h , and let $k < n$. The k th interval of S , denoted $I_k(S)$, is the set of update $_k$ operations $U \in h$ such that:

1. $\neg(S <_h U)$.
2. There is no update $_k$ operation U' such that $U <_h U' <_h S$.

Second condition. The values of the update operations U_i and U_j are the values returned by S in its corresponding entries.

Third condition. There is no, e.g., update $_i$ operation between U_i and U_j . This is because the existence of such an update $_i$ operation, say $U_i < U'_i < U_j$, would prevent us from linearizing both U_i and U_j before S .

We formalize the notion of appropriateness in the following definition:

► **Definition 9.** Let S be a complete scan operation in an (i, j) -simple history h . A pair (U_i, U_j) where U_i is an update_i operation and U_j is an update_j operation, is said to be S -appropriate, if:

1. $U_i \in I_i(S)$ and $U_j \in I_j(S)$.
2. $(\text{val}_h(U_i), \text{val}_h(U_j)) = (\text{val}_{h:i}(S), \text{val}_{h:j}(S))$.
3. There is no update_i operation U'_i such that $U_i < U'_i < U_j$, and there is no update_j operation U'_j such that $U_j < U'_j < U_i$.

So far, we have presented our properties and explained intuitively why they form a necessary condition for linearizability: i.e., why their negation prevents linearizability. The main theorem of this section asserts a much stronger claim: these properties also constitute a sufficient condition for linearizability. We provide a proof for Theorem 10 in the full version [49, Appendix D].

► **Theorem 10.** An (i, j) -simple history h is linearizable if and only if the following properties hold.

No Inversion. There are no complete scan operations S_1 and S_2 in h such that $(\text{val}_{h:i}(S_1), \text{val}_{h:j}(S_1)) = (0, 1)$ and $(\text{val}_{h:i}(S_2), \text{val}_{h:j}(S_2)) = (1, 0)$.

Non-Decreasing. If S_1 and S_2 are two complete scan operations in h such that $S_1 <_h S_2$, then $\text{val}_{h:i}(S_1) \leq \text{val}_{h:i}(S_2)$ and $\text{val}_{h:j}(S_1) \leq \text{val}_{h:j}(S_2)$.

Appropriateness. For each complete scan operation S in h , there exists an S -appropriate pair of update operations.

4.1 Detecting Incorrect Simple Histories

Finally, we show that the properties of Theorem 10 can be detected by an NFA. We provide here proof sketches for most claims, and full proofs for all claims in the full version [49, Appendix E].

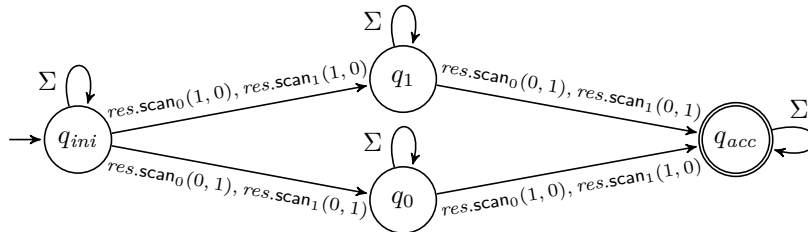
► **Theorem 11.** For $i < j < n$, there exists an automaton M with $O(n)$ states and $O(n^2)$ transitions, such that an (i, j) -simple history h is not linearizable if and only if $h \in L(M)$.

To prove Theorem 11, we construct automata that detect violations of the three properties presented in Theorem 10.

4.1.1 Detecting Violations of No-Inversion

► **Proposition 12.** There exists an automaton M_1 such that, for any (i, j) -simple history h , h violates No-Inversion if and only if $h \in L(M_1)$. Moreover, M_1 has $O(1)$ states and $O(n)$ transitions.

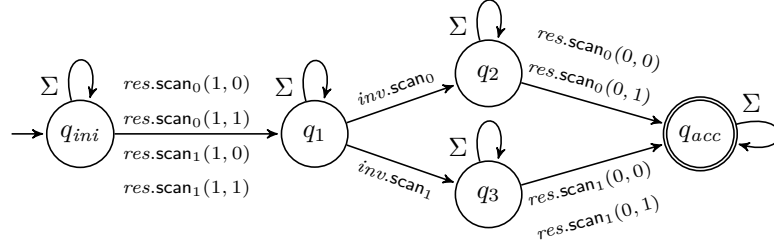
Proof sketch. We demonstrate the construction for the case that $n = 2$:



4.1.2 Detecting Violations of Non-Decreasing

► **Proposition 13.** *There exists an automaton M_2 such that, for any (i, j) -simple history h , h violates Non-Decreasing if and only if $h \in L(M_2)$. Moreover, M_2 has $O(n)$ states and $O(n^2)$ transitions.*

Proof sketch. As an illustrative demonstration, we present below a simpler automaton. It detects the existence of a violation of Non-Decreasing, for $n = 2$, and $S_1 < S_2$ where $val_{h:i}(S_1) = 1$.



4.1.3 Detecting Violations of Appropriateness

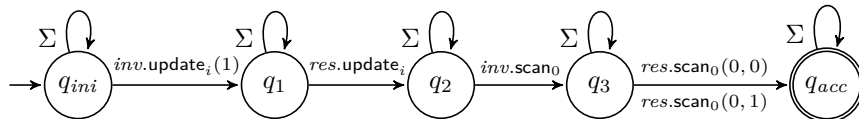
It remains to construct an automaton that accepts all (i, j) -simple histories that violate Appropriateness. To this end, we reformulate Appropriateness as a regular safety property; a word is rejected if and only if it has a “bad”-prefix.

► **Lemma 14.** *Let h be an (i, j) -simple history, and let S be a complete scan operation in h . For $l \in \{i, j\}$, let F_l be the first $update_l(1)$ operation in h , if exists. Then, there is no S -appropriate pair in h if and only if any of the following holds:*

1. *For $l \in \{i, j\}$, F_l exists, $val_{h:l}(S) = 0$, and $F_l <_h S$.*
2. *For $l \in \{i, j\}$, $val_{h:l}(S) = 1$, and either $S <_h F_l$ or F_l doesn't exist.*
3. *$(val_{h:i}(S), val_{h:j}(S)) = (0, 1)$ and $F_i <_h F_j$.*
4. *$(val_{h:i}(S), val_{h:j}(S)) = (1, 0)$ and $F_j <_h F_i$.*

► **Proposition 15.** *There exists an automaton M_3 such that, for any (i, j) -simple history h , h violates Appropriateness if and only if $h \in L(M_3)$. Moreover, M_3 has $O(n)$ states and $O(n^2)$ transitions.*

Proof sketch. The automaton is a “union” of four automaton, such that the k th automaton checks whether there exists a complete scan operation S for which the k th case of Lemma 14 holds. Below, we provide an automaton that identifies the first case where $l = i$ and S is a $scan_0$ operation (for $n = 2$). Hence, in fact, the first case is a union of $2n$ automaton. We leave it for the reader to verify that (rather simple) automaton exist for all other cases.



► **Corollary 16.** *Theorem 11 is trivially correct by propositions 12, 13, and 15.*

5 Verifying and Testing Linearizability

The results of the previous section allow us to both verify data-independent snapshot implementations, and test the linearizability of simple snapshot histories, in polynomial time.

For verifying a data-independent snapshot implementation, by Theorem 5, it is sufficient to verify that all of its simple histories are linearizable. Theorem 11 allows one to apply model checking of regular properties [12, Chapter 4.2], i.e., one can check whether the implementation admits an (i, j) -simple history accepted by the automaton (and thus not linearizable). As there are $O(n^2)$ valuations for i and j , our first key result follows:

► **Theorem 17** (Polynomial-time verification). *Let $\mathcal{L}_{\text{snap}}$ be a data-independent snapshot algorithm such that only finitely many states of $\mathcal{L}_{\text{snap}}$ are reachable by its simple histories. Determining the linearizability of $\mathcal{L}_{\text{snap}}$ is decidable in $O(mn^4)$ time, where m is the size of an automaton that accepts all simple histories of $\mathcal{L}_{\text{snap}}$.*

Moreover, Theorem 11 enables the testing of simple histories efficiently, by feeding the automaton of Theorem 11 (or its determinization) with the (i, j) -simple histories to be tested. The automaton will report acceptance once it identifies a non-linearizable prefix of the input history h (which testifies that h is not linearizable). Hence, our second key result follows.

► **Theorem 18** (Linear-time testing). *For $i < j < n$, (i, j) -simple histories can be tested in linear-time, i.e., in time $O(|h|)$.*

6 Optimization: Omitting Redundant Commands

The focus on simple histories yields an optimization that can significantly reduce the state space of an examined snapshot algorithm, speeding up its verification. The optimization relies on the observation that in the executions of (i, j) -simple histories, some commands are vacuous. To elaborate, assume that register R stores the data value of process p_k , $k \neq i, j$. In all executions of (i, j) -simple histories, R will only ever store the value 0. Thus, read and write commands from and to R can be ignored, reducing the possible values of the program counters. In some cases, we can even ignore R altogether, reducing the number of registers.

We use Bowman's obstruction-free snapshot algorithm [17] (Algorithm 1) to demonstrate the optimization. During an `update` operation, process p_k writes its new value to register $A[k]$ (line 3). During a `scan` operation, the values stored in $A[0], \dots, A[n-1]$ are read into the local variables $a[0], \dots, a[n-1]$ (lines 7-8). Let $i < j$ be two process ids, and consider executions of (i, j) -simple histories of Algorithm 1. In such executions, every write command to register $A[k]$, $k \notin \{i, j\}$, writes 0. Hence, we may ignore and omit all registers $A[k]$, $k \notin \{i, j\}$. This yields a simplified version of the algorithm, as shown in Algorithm 2, which has a substantially smaller state space than Algorithm 1, as it employs fewer registers. Since we omitted only vacuous commands (i.e. commands that always write and read 0) Algorithm 2 is linearizable if and only if all of Algorithm 1's (i, j) -simple histories are linearizable.

7 Implementation and Evaluation

In this section, we describe implementations of the procedures described in Section 5, and the experiments we performed to evaluate their efficiency. We provide the means to reproduce all experiments in the paper's supporting materials [49].

■ **Algorithm 1** Unoptimized algorithm.

```

1: procedure updatek(v)
2:   Active ← ⊥
3:   A[k] ← v

4: procedure scank
5:   repeat
6:     Active ← k
7:     for ℓ = 0, …, n-1 do
8:       a[ℓ] ← A[ℓ]
9:   until Active = k
10:  return (a[0], …, a[n-1])

```

■ **Algorithm 2** Optimized for (i, j)-simple executions.

```

1: procedure updatek                                ▷ k ∉ {i, j}
2:   Active ← ⊥

3: procedure updater(v)                                ▷ r ∈ {i, j}
4:   Active ← ⊥
5:   A[r] ← v

6: procedure scanq                                    ▷ q < n
7:   repeat
8:     Active ← q
9:     a[i] ← A[i]
10:    a[j] ← A[j]
11:  until Active = q
12:  return (0, …, 0, a[i], 0, …, 0, a[j], 0, …, 0)

```

■ **Figure 3** Illustration of the redundant command omission optimization with Bowman’s algorithm.

7.1 Implementation of our Verification Procedures

We used the model checker PAT [44] as the basis for our two verification approaches. PAT contains a system for checking the linearizability of a given concurrent algorithm against an abstract specification, via refinement [38, 40]. We made use of this system in our first verification approach: we encoded known snapshot algorithms from the literature (listed in subsection 7.4), modified to admit only simple histories. We provided a matching abstract simple-history snapshot specification.

For our second approach, we encoded the automaton from Theorem 11 in PAT. As that automaton is a union of several automatons, we treated each one as a separate process, and encoded the union as the parallel composition of these processes. We exploited PAT’s reachability checker to encode the accepting states of the automaton. We then asked PAT to check whether the algorithms listed in subsection 7.4 admit any simple histories that are accepted by the automaton.

We note three sources of possible errors in our implementations: (1) We could have encoded the snapshot algorithms incorrectly. (2) We could have encoded the automatons or the abstract specification incorrectly. (3) PAT itself may have bugs. To mitigate the first two threats, we used PAT’s linearizability system to ensure that the algorithms we encoded are linearizable, that we manage to find several artificially-inserted bugs, and that the reachability approach agrees with PAT’s standard refinement approach. We did not take steps to mitigate the third threat, but as PAT is a widely used model checker which has itself been partially model-checked [48], our confidence in its correctness is high., our confidence in its correctness is high.

7.2 Implementation of our Testing Procedure

The testing procedure we implemented receives an (i, j)-simple history, and runs it through an implementation of the automaton described in Theorem 11. The tool announces whether the automaton accepts the history, indicating it is not linearizable, or it rejects the history, indicating it is linearizable.

To validate that our implementation has no bugs, we generated hundreds of random simple histories, both linearizable and non-linearizable, and ensured our implementation classified them correctly.

7.3 Research Questions

We start with research questions related to our verification technique. As we propose a model checking approach, although polynomial, it still suffers from the state explosion problem [23]. This holds since the algorithms we check admit an enormous number of states, even when we restrict ourselves to simple histories. Model-checking approaches are mainly evaluated based on their feasibility; their ability to verify correctness/find bugs, perhaps only up to a reasonable depth, measured in the number of operations each process executes, with real-world resources: realistic time and space and limitations. Hence, we formulate the following research questions:

RQ1 Does the focus on simple histories help to prove/disprove correctness, in terms of feasibility/depth to be processed?

RQ2 Is our polynomial-time technique efficient for proving/disproving correctness, in terms of feasibility/depth to be processed?

We use the following research question to evaluate our testing technique:

RQ3 Is our testing technique efficient, in terms of feasibility, and time and space consumption?

7.4 Corpus

To address RQ1 and RQ2, we constructed a corpus for our experiments that includes several snapshot algorithms from the literature: An obstruction-free [30] algorithm by Bowman [17], denoted BOWMAN; A snapshot algorithm by Jayanti [35], denoted JAYANTI; The bounded and unbounded versions of Afek et al. [4], denoted AFEK1 and AFEK2, respectively; and A snapshot algorithm by Riany et al [46], denoted RIANY.

For each algorithm and $n \in \{3, 4, 5, 6\}$ processes, we encoded the original version (denoted “full”), as well as a modified version which generates only $(0, 1)$ -simple histories, with the optimization detailed in Section 6 (denoted “simple-only”). Then, for $n \in \{3, 4, 5, 6, 8, 10\}$, we also encoded buggy versions thereof (denoted, “buggy-full” and “buggy-simple-only”, respectively). Overall, we created 100 configurations of pairs of algorithm encoding with n processes.

To address RQ3, we began by generating 25 linearizable histories of length $l \in \{200, 500, 1000\}$ with $n \in \{5, 8, 11, 14, 17, 20\}$ processes, by randomly executing an atomic snapshot implementation, and recording its actions. We then generated 25 non-linearizable histories of length $l \in \{50, 100, 200\}$ with $n \in \{3, 4, 5, 6, 8, 10\}$ processes as follows: we generated a random linearizable history, and changed its 20-length suffix by randomly changing the values of the scan responses. We repeated this process until we obtained 25 non-linearizable histories. In the context of RQ3, we refer to a choice of l , n , and “linearizable/non-linearizable” as a configuration. This resulted in 900 histories, divided into 18 linearizable and 18 non-linearizable configurations, added to our corpus.

7.5 Experiments and Results

In this section, we detail the experiments we performed to tackle our research questions, and report our results. All experiments were performed on a rather ordinary laptop with an Intel Core i7-6820HK CPU and 32GB of DDR4 RAM, running Windows 10 21H1 and the WSL2 Ubuntu 20.04.2 image from Microsoft.

■ **Table 1** Results of bug detection in non-linearizable implementations. b: max bound on #operation per process, t: time used (sec.), and s: memory used (GB).

test	normal			simple			polynomial			normal			simple			polynomial		
	b	t	s	b	t	s	b	t	s	b	t	s	b	t	s	b	t	s
algorithm	3 processes									4 processes								
BOWMAN	∞	11	0.64	∞	2	0.1	∞	2	0.3	3	168	12.0	∞	60	0.5	∞	54	0.6
JAYANTI	14	272	22.8	23	393	25.1	3	72	0.7	3	149	11.9	6	218	16.0	3	98	0.7
AFEK1	-	-	-	2	242	1.9	2	417	1.8	-	-	-	2	305	2.0	2	506	2.3
AFEK2	2	50	0.5	4	422	4.3	3	215	1.4	2	163	2.0	4	522	7.0	3	255	2.1
RIANY	6	285	4.1	9	445	5.4	27	595	2.0	3	172	12.1	6	275	17	24	512	1.9
algorithm	5 processes									6 processes								
BOWMAN	1	9	0.4	3	290	18.4	40	575	4.4	1	66	3.8	1	7	0.5	35	507	3.3
JAYANTI	1	141	0.9	3	253	18.4	3	118	0.7	1	302	4.6	2	590	28.1	3	142	0.7
AFEK1	-	-	-	2	417	4.0	-	-	-	-	-	-	-	-	-	-	-	-
AFEK2	1	21	0.4	3	441	18.4	3	298	2.6	1	95	4.2	1	7	0.5	3	326	2.8
RIANY	-	-	-	-	-	-	23	579	1.8	-	-	-	-	-	-	21	508	2.2
algorithm	8 processes									10 processes								
BOWMAN	-	-	-	1	288	15.6	33	554	2.7	-	-	-	-	-	-	30	558	2.9
JAYANTI	-	-	-	1	386	15.6	3	197	0.7	-	-	-	-	-	-	3	267	1.1
AFEK1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
AFEK2	-	-	-	1	256	15.7	3	402	3.2	-	-	-	-	-	-	3	491	3.6
RIANY	-	-	-	-	-	-	19	502	2.2	-	-	-	-	-	-	18	586	3.0

7.5.1 Verification Experiments

To address RQ1 and RQ2, we used PAT to verify the correctness of our configurations. We ran three different types of linearizability experiments:

normal. Using PAT’s standard linearizability checker with full and buggy-full configurations.
 simple. Using PAT’s standard linearizability checker with simple-only and buggy-simple-only configurations.

polynomial. Using PAT’s reachability checker with simple-only and buggy-simple-only configurations, in parallel to the automaton threads that detect bugs.

Furthermore, for each configuration and matching experiment type, we limited the number of operations that each process was allowed to perform. As some algorithms employ infinite data types (e.g. integers), at least in those cases, the bound is mandatory for PAT to terminate. We set a timeout of 10 min. for buggy implementations, and 1 hr. for correct implementations. We repeated each experiment with various bounds until we found the maximal bound for which each experiment terminated in the allotted time.

► **Remark 19.** For simple configurations, we checked only $(0, 1)$ -simple histories. For full verification, it is required to test all (i, j) -histories. Nevertheless, this observation does not affect the feasibility of the approach, since the tests for (i_0, j_0) and (i_1, j_1) simple histories are independent, and can even run on separate machines. Furthermore, symmetry arguments may increase confidence even when checking only $(0, 1)$ -simple histories.

► **Remark 20.** We also tried to use Cave [21, 50] and its extension Poling [45, 55], static analysis-based linearizability verifiers. Unfortunately, despite our best efforts, we could not make either tool work for the algorithms we tried to encode. Even for toy correct and

■ **Table 2** Results of verification of linearizable implementations. b: max bound on #operation per process, t: time used (sec.), and s: memory used (GB).

test	normal			simple			polynomial			normal			simple			polynomial		
	b	t	s	b	t	s	b	t	s	b	t	s	b	t	s	b	t	s
algorithm	3 processes									4 processes								
BOWMAN	2	47	0.7	∞	3	0.1	∞	11	0.1	1	29	0.6	∞	134	1.8	∞	460	0.7
JAYANTI	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
AFEK1	1	113	1.0	1	15	0.2	1	31	0.3	-	-	-	-	-	-	-	-	-
AFEK2	1	12	0.2	2	2627	19.5	2	2675	7.5	-	-	-	1	154	1.9	1	269	1.5
RIANY	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
algorithm	5 processes									6 processes								
BOWMAN	-	-	-	1	158	1.4	1	121	0.5	-	-	-	-	-	-	-	-	-
JAYANTI	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
AFEK1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
AFEK2	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
RIANY	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

incorrect algorithms that operate atomically, Cave reported errors, and Poling returned unexpected responses. Perhaps the snapshot object deviates from the types of data structures that these tools aim to handle. To the best of our knowledge, PAT, Cave, and Poling are the only available tools that verify linearizability automatically, without requiring additional input.

We present the bug-detection results in Table 1, and the verification results in Table 2. For each configuration and linearizability experiment type, we report the maximal bound on the number of operations per process, for which the experiment terminated before the timeout. If the experiment terminated without imposing a bound, we report the value ∞ . Furthermore, for the max bound we found, we report time and space consumption by the corresponding linearizability experiment. As an example, for RIANY buggy-simple-only with 4 processes, when we ran the polynomial linearizability experiment, we found the bug while limiting each process to 24 operations. The execution took 512 sec. and consumed 1.9 GB. Accordingly, in the upper part of Table 1, the cells on the row titled “RIANY” and the columns titled “polynomial; 4 processes” read: b:24, t:512, and s:1.9.

7.5.2 Testing Experiments

To address RQ3, we tested all linearizable and non-linearizable generated histories, applying two methods: our implemented method, and a tool by Lowe [37, 41], with a 10 min. timeout. For each configuration and each tool, we report the percentage of tests that successfully terminated within the allotted time. Furthermore, for the terminated executions, we report the median running time and space consumption. Table 3 presents results for linearizable configurations, and Table 4 for non-linearizable configurations. As an example, when we applied our method to linearizable histories of length 500 with 20 processes, 100% of the tests were successful, the median running time was 0.15sec, and the median space consumption was 150MB. Hence, in Table 3, the cells on the rows titled “500:terminated”, “500:median time”, and “500:median space” with the column titled “20;This paper” read 100%, 0.15, and 150, respectively.

■ **Table 3** Linearizable simple history testing results. Terminated tests (%), median time used (sec.), and median memory used (MB).

#processes		5		8		11		14		17		20	
mth. len.		This paper	Lowe	This paper	Lowe	This paper	Lowe	This paper	Lowe	This paper	Lowe	This paper	Lowe
		200	term.	100%	100%	100%	100%	100%	100%	100%	84%	100%	52%
	time	0.02	0.20	0.02	0.22	0.02	0.28	0.02	1.34	0.02	1.11	0.03	2.08
	space	130	336	131	352	131	360	131	456	131	444	132	1228
500	term.	100%	100%	100%	100%	100%	100%	100%	80%	100%	48%	100%	16%
	time	0.04	0.20	0.06	0.22	0.09	0.30	0.12	1.45	0.16	12.57	0.15	0.21
	space	132	216	136	336	140	352	144	492	150	2414	150	346
1000	term.	100%	100%	100%	100%	100%	100%	100%	72%	100%	44%	100%	16%
	time	0.07	0.21	0.14	0.21	0.21	0.33	0.33	1.36	0.40	3.29	0.56	0.85
	space	136	344	146	336	156	352	171	482	182	1620	203	398

■ **Table 4** Non-linearizable simple history testing results. Terminated tests (%), median time used (sec.), and median memory used (MB).

#processes		3		4		5		6		8		10	
mth. len.		This paper	Lowe	This paper	Lowe	This paper	Lowe	This paper	Lowe	This paper	Lowe	This paper	Lowe
		50	term.	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%
	time	0.06	0.28	0.06	0.32	0.06	0.34	0.06	0.35	0.06	0.65	0.06	1.19
	space	129	336	129	348	129	336	129	344	129	368	129	440
100	term.	100%	100%	100%	100%	100%	84%	100%	52%	100%	32%	100%	8%
	time	0.03	0.48	0.03	1.45	0.03	3.76	0.03	24.61	0.03	61.93	0.03	179.80
	space	129	340	129	508	129	2784	129	17896	129	12996	129	12192
200	term.	100%	36%	100%	4%	100%	0%	100%	0%	100%	0%	100%	0%
	time	0.03	30.78	0.03	5.11	0.03	-	0.03	-	0.03	-	0.03	-
	space	129	20392	129	3216	129	-	129	-	129	-	129	-

7.6 Analysis of the Results

Focusing on simple histories is beneficial, as both **simple** and **polynomial** outperform the **normal** linearizability method of PAT for finding bugs. In addition, overall, **polynomial** performs better than **simple** (see Table 1). **normal** found the bug in 14/30 cases, with up to 6 processes. **simple** succeeded in 3 additional cases with up to 8 processes, with **polynomial** succeeding in 26/30 cases with up to 10 processes. Importantly, **simple** allows for larger bounds than **normal** in 16/17 cases, and the same bound in the remaining case. **polynomial** allows for larger bounds than **simple** in 16 cases, and smaller bounds in 5 cases. This indicates that **polynomial** enables deeper exploration than **simple**, and thus we conclude that it is more efficient. Both **polynomial** and **simple** manage to explore implementations significantly deeper than **normal**. We also observe that **polynomial** consumes less space, which is the main bottleneck of model checking, than **simple**. The peak memory consumption we recorded for **polynomial** was 4.4GB, whereas the peak we recorded for **simple** was 25.1GB, and 9/20 executions with more than 10GB. While the peak we recorded for **normal** was 22.8GB, it scaled much worse than **simple** and failed to cope with the more challenging configurations.

In Table 2, we see that **simple** and **polynomial** enable the verification of BOWMAN with 3 and 4 processes. To the best of our knowledge, this is the first time that this algorithm has been verified to some extent. Model-checking techniques are complete and mainly efficient for

bug detection. Verifying a concurrent algorithm for 4 processes is noteworthy (compare, e.g., to the results of [39]). Yet, excluding these results, although both methods perform better than `normal`, we did not manage to verify other implementations. We mention that, in some old evaluations we performed, we used a prototype tool we wrote that uses simple histories (but does not employ the automata of Section 4.1), and managed to verify `JAYANTI` with 3 processes within 21 sec. (reference hidden for double-blind review). As this deviates from what Table 2 illustrates, we believe that further investigation is required.

Tables 3 and 4 show that our testing technique outperforms [41] by several orders of magnitude, mainly and most importantly, in terms of feasibility. Our tool easily handled all 900 histories, while the competitor failed to cope with challenging configurations, successfully handling only 554/900 histories. We also observe that our technique is scalable. The differences in time and space consumption between extremum values are negligible.

Moreover, we note that our technique is insensitive to the correctness of the tested history. In contrast, our competitor quickly fails over non-linearizable histories. To gain more confidence in this observation, we further generated 25 non-linearizable histories of length 1000 for 20 processes, with a linearizable prefix of length at least 980. Our tool handled all with a median running time of 0.55sec. Note that our competitor failed almost entirely over non-linearizable histories of length 200, with 3-10 processes.

8 Related Work

Alur et al. proposed an EXPSPACE-technique for verifying linearizability [5], and Hamza proved EXPSPACE-completeness [29]. Bouajjani et al. proved the undecidability of linearizability of infinite-state systems, and the PSPACE-completeness of linearizability with fixed linearization points [14].

Due to the high complexity of the problem, sound and complete model-checking techniques manage to perform limited verification with up to 3 processes [19, 39, 52]. [39] also verifies a stack implementation for 4 processes, but only by limiting the stack size to two data values. Hsu et al. [33] proposed a bounded model checking technique for hyper-LTL, and used it to rediscover known bugs (see [25]) in the “Snark” dequeue implementation [24].

Static analysis efforts are incomplete, but can work for infinite-state implementations. However, most ask for additional information from the user. The works [3, 6, 13, 50] ask for linearization points, some in a conditional manner. The work [2] asks for linearization policies. The work [47] asks for the specification of sub-operations and relations between them. Cave [21, 51] and Poling [55] work without further information. However, as we report in Section 7, we did not manage to work with these tools. Perhaps the snapshot object deviates from the types of data structures that these tools aim to handle.

The way we employ the data independence property resembles Abdulla et al. [1]. They ran automata in parallel to queue and stack implementations to detect bugs. Their approach is incomplete, but works for infinite-state implementations. However, their automata detect incorrect sequential histories, in contrast to concurrent histories as we do, and thus their approach requires specifying linearization points. It is rather simple to construct an automaton that detects incorrect sequential snapshot histories, hence their approach can be applied to the snapshot object straightforwardly. But, as linearization points of snapshot implementations are evasive, the benefit of doing so is questionable.

Other works also focused on specific data structures. Bouajjani et al. [15] prove that verification of data-independent queue, stack, register, and mutex implementations is PSPACE-complete for a fixed number of processes, and EXPSPACE-complete for infinitely many

processes. In [16], Bouajjani et al. extend the latter result to data-independent and projection-closed priority queues. To the best of our knowledge, those techniques have not been implemented or evaluated. Chakraborty et al. [22] identified conditions that are equivalent to the linearizability of data-independent queue implementations, and use them to automatically verify Herlihy and Wing’s queue [31]. Abdulla et al. [3] used those conditions and the results of [22] to extend their static analysis technique [2] to verify stack and queue implementations without linearization points.

Wing and Gong considered the problem of testing linearizability, and gave an exponential-time algorithm [53]. Gibbons and Korach proved NP-completeness [28], and further showed that register-histories with k processes can be tested in time $O(n2^{O(k)}+n \log n)$. Lowe [41] suggested optimizations for the algorithm of [53]. Horn and Kroening suggested an optimization that applies to set implementations [32]. Emmi and Enea [27] identified a class of data structures for which a polynomial-time testing algorithm exists. This class includes queue, stack, set, and map, but does not include snapshot.

9 Conclusion

We proved that a data-independent snapshot algorithm is linearizable if and only if all of its simple histories are linearizable. This gives rise to an optimization for proving/disproving the correctness of snapshot implementations, i.e, examining only simple histories. This optimization can exponentially reduce the number of reachable states to inspect. Moreover, we proved that non-linearizable simple histories are identified by a polynomial-sized automaton. This enables a polynomial-time technique for verifying the linearizability of snapshot implementations, and a linear-time technique for testing the linearizability of snapshot histories. We implemented our techniques, and reported on evaluations that support the efficiency of our methods over existing techniques.

Future Work

We wonder if the notion of simple histories can be replicated to other data structures. In particular, it would be interesting to investigate whether such an adaptation would admit automata-based verification/testing techniques similar to those we presented for the snapshot object. The automata presented in [14] seem like a good place to begin in order to define simple histories geared at queues and stacks. Another future direction is to extend our results to multi-writer snapshots, and to implementations that are strongly linearizable [43].

References

- 1 Parosh Aziz Abdulla, Frédéric Haziza, Lukás Holík, Bengt Jonsson, and Ahmed Rezzine. An integrated specification and verification technique for highly concurrent data structures for highly concurrent data structures. *Int. J. Softw. Tools Technol. Transf.*, 19(5):549–563, 2017. doi:10.1007/s10009-016-0415-4.
- 2 Parosh Aziz Abdulla, Bengt Jonsson, and Cong Quy Trinh. Automated verification of linearization policies. In Xavier Rival, editor, *Static Analysis - 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8-10, 2016, Proceedings*, volume 9837 of *Lecture Notes in Computer Science*, pages 61–83. Springer, 2016. doi:10.1007/978-3-662-53413-7_4.

- 3 Parosh Aziz Abdulla, Bengt Jonsson, and Cong Quy Trinh. Fragment abstraction for concurrent shape analysis. In Amal Ahmed, editor, *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*, volume 10801 of *Lecture Notes in Computer Science*, pages 442–471. Springer, 2018. doi:10.1007/978-3-319-89884-1_16.
- 4 Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic snapshots of shared memory. *J. ACM*, 40(4):873–890, 1993. doi:10.1145/153724.153741.
- 5 Rajeev Alur, Kenneth L. McMillan, and Doron A. Peled. Model-checking of correctness conditions for concurrent objects. *Inf. Comput.*, 160(1-2):167–188, 2000. doi:10.1006/inco.1999.2847.
- 6 Daphna Amit, Noam Rinetzky, Thomas W. Reps, Mooly Sagiv, and Eran Yahav. Comparison under abstraction for verifying linearizability. In Werner Damm and Holger Hermanns, editors, *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*, volume 4590 of *Lecture Notes in Computer Science*, pages 477–490. Springer, 2007. doi:10.1007/978-3-540-73368-3_49.
- 7 James H. Anderson. Composite registers. *Distributed Comput.*, 6(3):141–154, 1993. doi:10.1007/BF02242703.
- 8 James H. Anderson. Multi-writer composite registers. *Distributed Comput.*, 7(4):175–195, 1994. doi:10.1007/BF02280833.
- 9 James Aspnes and Maurice Herlihy. Wait-free data structures in the asynchronous PRAM model. In Frank Thomson Leighton, editor, *Proceedings of the 2nd Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA '90, Island of Crete, Greece, July 2-6, 1990*, pages 340–349. ACM, 1990. doi:10.1145/97444.97701.
- 10 Hagit Attiya, Maurice Herlihy, and Ophir Rachman. Efficient atomic snapshots using lattice agreement (extended abstract). In Adrian Segall and Shmuel Zaks, editors, *Distributed Algorithms, 6th International Workshop, WDAG '92, Haifa, Israel, November 2-4, 1992, Proceedings*, volume 647 of *Lecture Notes in Computer Science*, pages 35–53. Springer, 1992. doi:10.1007/3-540-56188-9_3.
- 11 Hagit Attiya and Ophir Rachman. Atomic snapshots in $o(n \log n)$ operations. *SIAM J. Comput.*, 27(2):319–340, 1998. doi:10.1137/S0097539795279463.
- 12 Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, 2008.
- 13 Josh Berdine, Tal Lev-Ami, Roman Manevich, G. Ramalingam, and Shmuel Sagiv. Thread quantification for concurrent shape analysis. In Aarti Gupta and Sharad Malik, editors, *Computer Aided Verification, 20th International Conference, CAV 2008, Princeton, NJ, USA, July 7-14, 2008, Proceedings*, volume 5123 of *Lecture Notes in Computer Science*, pages 399–413. Springer, 2008. doi:10.1007/978-3-540-70545-1_37.
- 14 Ahmed Bouajjani, Michael Emmi, Constantin Enea, and Jad Hamza. Verifying concurrent programs against sequential specifications. In Matthias Felleisen and Philippa Gardner, editors, *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, volume 7792 of *Lecture Notes in Computer Science*, pages 290–309. Springer, 2013. doi:10.1007/978-3-642-37036-6_17.
- 15 Ahmed Bouajjani, Michael Emmi, Constantin Enea, and Jad Hamza. On reducing linearizability to state reachability. *Inf. Comput.*, 261:383–400, 2018. doi:10.1016/j.ic.2018.02.014.
- 16 Ahmed Bouajjani, Constantin Enea, and Chao Wang. Checking linearizability of concurrent priority queues. In Roland Meyer and Uwe Nestmann, editors, *28th International Conference on Concurrency Theory, CONCUR 2017, September 5-8, 2017, Berlin, Germany*, volume 85 of *LIPICs*, pages 16:1–16:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPICs.CONCUR.2017.16.

- 17 Jack R Bowman. Obstruction-free snapshot, obstruction-free consensus, and fetch-and-add modulo k . Technical report, Technical Report TR2011-681, Dartmouth College, Computer Science, Hanover, NH, 2011.
- 18 Sebastian Burckhardt, Rajeev Alur, and Milo M. K. Martin. Checkfence: checking consistency of concurrent data types on relaxed memory models. In Jeanne Ferrante and Kathryn S. McKinley, editors, *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, pages 12–21. ACM, 2007. doi:10.1145/1250734.1250737.
- 19 Sebastian Burckhardt, Chris Dern, Madanlal Musuvathi, and Roy Tan. Line-up: a complete and automatic linearizability checker. In Benjamin G. Zorn and Alexander Aiken, editors, *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010*, pages 330–340. ACM, 2010. doi:10.1145/1806596.1806634.
- 20 Jacob Burnim, George C. Necula, and Koushik Sen. Specifying and checking semantic atomicity for multithreaded programs. In Rajiv Gupta and Todd C. Mowry, editors, *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2011, Newport Beach, CA, USA, March 5-11, 2011*, pages 79–90. ACM, 2011. doi:10.1145/1950365.1950377.
- 21 CAVE Website. <https://people.mpi-sws.org/~viktor/cave/>. Last Accessed: Aug. 31, 2021.
- 22 Soham Chakraborty, Thomas A. Henzinger, Ali Sezgin, and Viktor Vafeiadis. Aspect-oriented linearizability proofs. *Log. Methods Comput. Sci.*, 11(1), 2015. doi:10.2168/LMCS-11(1:20)2015.
- 23 Edmund M. Clarke, William Klieber, Milos Nováček, and Paolo Zuliani. Model checking and the state explosion problem. In Bertrand Meyer and Martin Nordio, editors, *Tools for Practical Software Verification, LASER, International Summer School 2011, Elba Island, Italy, Revised Tutorial Lectures*, volume 7682 of *Lecture Notes in Computer Science*, pages 1–30. Springer, 2011. doi:10.1007/978-3-642-35746-6_1.
- 24 David Detlefs, Christine H. Flood, Alex Garthwaite, Paul Alan Martin, Nir Shavit, and Guy L. Steele Jr. Even better dcas-based concurrent dequeues. In Maurice Herlihy, editor, *Distributed Computing, 14th International Conference, DISC 2000, Toledo, Spain, October 4-6, 2000, Proceedings*, volume 1914 of *Lecture Notes in Computer Science*, pages 59–73. Springer, 2000. doi:10.1007/3-540-40026-5_4.
- 25 Simon Doherty, David Detlefs, Lindsay Groves, Christine H. Flood, Victor Luchangco, Paul Alan Martin, Mark Moir, Nir Shavit, and Guy L. Steele Jr. DCAS is not a silver bullet for nonblocking algorithm design. In Phillip B. Gibbons and Micah Adler, editors, *SPAA 2004: Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures, June 27-30, 2004, Barcelona, Spain*, pages 216–224. ACM, 2004. doi:10.1145/1007912.1007945.
- 26 Cynthia Dwork, Maurice Herlihy, Serge A. Plotkin, and Orli Waarts. Time-lapse snapshots. *SIAM J. Comput.*, 28(5):1848–1874, 1999. doi:10.1137/S0097539793243685.
- 27 Michael Emmi and Constantin Enea. Sound, complete, and tractable linearizability monitoring for concurrent collections. *Proc. ACM Program. Lang.*, 2(POPL):25:1–25:27, 2018. doi:10.1145/3158113.
- 28 Phillip B. Gibbons and Ephraim Korach. Testing shared memories. *SIAM J. Comput.*, 26(4):1208–1244, 1997. doi:10.1137/S0097539794279614.
- 29 Jad Hamza. On the complexity of linearizability. *Comput.*, 101(9):1227–1240, 2019. doi:10.1007/s00607-018-0596-7.
- 30 Maurice Herlihy, Victor Luchangco, and Mark Moir. Obstruction-free synchronization: Double-ended queues as an example. In *23rd International Conference on Distributed Computing Systems (ICDCS 2003), 19-22 May 2003, Providence, RI, USA*, pages 522–529. IEEE Computer Society, 2003. doi:10.1109/ICDCS.2003.1203503.

- 31 Maurice Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990. doi:10.1145/78969.78972.
- 32 Alex Horn and Daniel Kroening. Faster linearizability checking via p-compositionality. In Susanne Graf and Mahesh Viswanathan, editors, *Formal Techniques for Distributed Objects, Components, and Systems - 35th IFIP WG 6.1 International Conference, FORTE 2015, Held as Part of the 10th International Federated Conference on Distributed Computing Techniques, DisCoTec 2015, Grenoble, France, June 2-4, 2015, Proceedings*, volume 9039 of *Lecture Notes in Computer Science*, pages 50–65. Springer, 2015. doi:10.1007/978-3-319-19195-9_4.
- 33 Tzu-Han Hsu, César Sánchez, and Borzoo Bonakdarpour. Bounded model checking for hyperproperties. In Jan Friso Groote and Kim Guldstrand Larsen, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings, Part I*, volume 12651 of *Lecture Notes in Computer Science*, pages 94–112. Springer, 2021. doi:10.1007/978-3-030-72016-2_6.
- 34 Prasad Jayanti. *f*-arrays: implementation and applications. In Aleta Ricciardi, editor, *Proceedings of the Twenty-First Annual ACM Symposium on Principles of Distributed Computing, PODC 2002, Monterey, California, USA, July 21-24, 2002*, pages 270–279. ACM, 2002. doi:10.1145/571825.571875.
- 35 Prasad Jayanti. An optimal multi-writer snapshot algorithm. In Harold N. Gabow and Ronald Fagin, editors, *Proceedings of the 37th Annual ACM Symposium on Theory of Computing, Baltimore, MD, USA, May 22-24, 2005*, pages 723–732. ACM, 2005. doi:10.1145/1060590.1060697.
- 36 Lefteris M. Kirousis, Paul G. Spirakis, and Philippos Tsigas. Reading many variables in one atomic operation: Solutions with linear or sublinear complexity. *IEEE Trans. Parallel Distrib. Syst.*, 5(7):688–696, 1994. doi:10.1109/71.296315.
- 37 Linearizability Tester Website. <http://www.cs.ox.ac.uk/people/gavin.lowe/LinearizabilityTesting/>. Last Accessed: Aug. 31, 2021.
- 38 Yang Liu, Wei Chen, Yanhong A. Liu, and Jun Sun. Model checking linearizability via refinement. In Ana Cavalcanti and Dennis Dams, editors, *Proceedings of the Second World Congress on Formal Methods (FM'09)*, volume 5850 of *Lecture Notes in Computer Science*, pages 321–337. Springer, 2009.
- 39 Yang Liu, Wei Chen, Yanhong A. Liu, and Jun Sun. Model checking linearizability via refinement. In Ana Cavalcanti and Dennis Dams, editors, *FM 2009: Formal Methods, Second World Congress, Eindhoven, The Netherlands, November 2-6, 2009. Proceedings*, volume 5850 of *Lecture Notes in Computer Science*, pages 321–337. Springer, 2009. doi:10.1007/978-3-642-05089-3_21.
- 40 Yang Liu, Wei Chen, Yanhong A. Liu, Jun Sun, Shao Jie Zhang, and Jin Song Dong. Verifying linearizability via optimized refinement checking. *IEEE Trans. Software Eng.*, 39(7):1018–1039, 2013. doi:10.1109/TSE.2012.82.
- 41 Gavin Lowe. Testing for linearizability. *Concurr. Comput. Pract. Exp.*, 29(4), 2017. doi:10.1002/cpe.3928.
- 42 Maged M Michael and Michael L Scott. Correction of a memory management method for lock-free data structures. Technical report, University of Rochester, Computer Science, 1995.
- 43 Sean Owens and Philipp Woelfel. Strongly linearizable implementations of snapshots and other types. In Peter Robinson and Faith Ellen, editors, *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, PODC 2019, Toronto, ON, Canada, July 29 - August 2, 2019*, pages 197–206. ACM, 2019. doi:10.1145/3293611.3331632.
- 44 PAT Website. <https://pat.comp.nus.edu.sg/>. Last Accessed: Aug. 31, 2021.
- 45 Poling Website. <https://github.com/rowangithub/Poling/>. Last Accessed: Aug. 31, 2021.
- 46 Yaron Riany, Nir Shavit, and Dan Touitou. Towards a practical snapshot algorithm. *Theor. Comput. Sci.*, 269(1-2):163–201, 2001. doi:10.1016/S0304-3975(00)00412-6.

- 47 Vineet Singh, Iulian Neamtiu, and Rajiv Gupta. Proving concurrent data structures linearizable. In *27th IEEE International Symposium on Software Reliability Engineering, ISSRE 2016, Ottawa, ON, Canada, October 23-27, 2016*, pages 230–240. IEEE Computer Society, 2016. doi:10.1109/ISSRE.2016.31.
- 48 Jun Sun, Yang Liu, and Bin Cheng. Model checking a model checker: A code contract combined approach. In Jin Song Dong and Huibiao Zhu, editors, *Formal Methods and Software Engineering - 12th International Conference on Formal Engineering Methods, ICFEM 2010, Shanghai, China, November 17-19, 2010. Proceedings*, volume 6447 of *Lecture Notes in Computer Science*, pages 518–533. Springer, 2010. doi:10.1007/978-3-642-16901-4_34.
- 49 Supporting materials. https://github.com/hayounav/Thesis_experiments/tree/main/snapshot%20verification%20and%20testing.
- 50 Viktor Vafeiadis. Shape-value abstraction for verifying linearizability. In Neil D. Jones and Markus Müller-Olm, editors, *Verification, Model Checking, and Abstract Interpretation, 10th International Conference, VMCAI 2009, Savannah, GA, USA, January 18-20, 2009. Proceedings*, volume 5403 of *Lecture Notes in Computer Science*, pages 335–348. Springer, 2009. doi:10.1007/978-3-540-93900-9_27.
- 51 Viktor Vafeiadis. Automatically proving linearizability. In Tayssir Touili, Byron Cook, and Paul B. Jackson, editors, *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*, volume 6174 of *Lecture Notes in Computer Science*, pages 450–464. Springer, 2010. doi:10.1007/978-3-642-14295-6_40.
- 52 Martin T. Vechev and Eran Yahav. Deriving linearizable fine-grained concurrent objects. In Rajiv Gupta and Saman P. Amarasinghe, editors, *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, pages 125–135. ACM, 2008. doi:10.1145/1375581.1375598.
- 53 Jeannette M. Wing and Chun Gong. Testing and verifying concurrent objects. *J. Parallel Distributed Comput.*, 17(1-2):164–182, 1993. doi:10.1006/jpdc.1993.1015.
- 54 Pierre Wolper. Expressing interesting properties of programs in propositional temporal logic. In *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages, St. Petersburg Beach, Florida, USA, January 1986*, pages 184–193. ACM Press, 1986. doi:10.1145/512644.512661.
- 55 He Zhu, Gustavo Petri, and Suresh Jagannathan. Poling: SMT aided linearizability proofs. In Daniel Kroening and Corina S. Pasareanu, editors, *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II*, volume 9207 of *Lecture Notes in Computer Science*, pages 3–19. Springer, 2015. doi:10.1007/978-3-319-21668-3_1.