Checking Linearizability Using Hitting Families

Burcu Kulahcioglu Ozkan
MPI-SWS
Kaiserslautern, Germany
burcu@mpi-sws.org

Rupak Majumdar
MPI-SWS
Kaiserslautern, Germany
rupak@mpi-sws.org

Filip Niksic
University of Pennsylvania
Philadelphia, PA, USA
fniksic@seas.upenn.edu

Abstract

Linearizability is a key correctness property for concurrent data types. Linearizability requires that the behavior of concurrently invoked operations of the data type be equivalent to the behavior in an execution where each operation takes effect at an instantaneous point of time between its invocation and return. Given an execution trace of operations, the problem of verifying its linearizability is NP-complete, and current exhaustive search tools scale poorly.

In this work, we empirically show that linearizability of an execution trace is often witnessed by a schedule that orders only a small number of operations (the "linearizability depth") in a specific way, independently of other operations. Accordingly, one can structure the search for linearizability witnesses by exploring schedules of low linearizability depth first. We provide such an algorithm. Key to our algorithm is a procedure to generate a *strong d-hitting family* of schedules, which is guaranteed to cover all linearizability witnesses of depth d. A strong d-hitting family of schedules of an execution trace consists of a set of schedules, such that for each tuple of d operations in the trace, there is a schedule in the family that (i) executes these operations in the order they appear in the tuple, and (ii) as late as possible in the execution.

We show that most linearizable execution traces from existing benchmarks can be witnessed by strongly d-hitting schedules for $d \leq 5$. Our result suggests a practical and automated method for showing linearizability of a trace based on a prioritization of schedules parameterized by the linearizability depth.

CCS Concepts • **Software and its engineering → Software testing and debugging**; • **Computing methodologies → Concurrent algorithms**; • **Mathematics of computing → Combinatorics**.

PPoPP '19, February 16–20, 2019, Washington, DC, USA

ACM ISBN 978-1-4503-6225-2/19/02…$15.00
https://doi.org/10.1145/3293883.3295726

Keywords linearizability, concurrent data structures, scheduling, hitting families

ACM Reference Format:
Burcu Kulahcioglu Ozkan, Rupak Majumdar, and Filip Niksic. 2019. Checking Linearizability Using Hitting Families. In *24th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '19), February 16–20, 2019, Washington, DC, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3293883.3295726

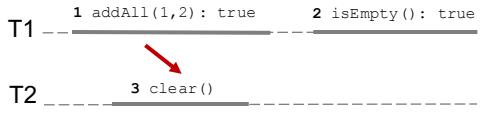
1 Introduction

Linearizability [15] is a standard correctness condition for concurrent data structures. It requires that each operation of a concurrent data structure executed concurrently by clients takes effect at an instantaneous point of time between its invocation and return, forming a total order on the effect of the operations consistent with a sequential execution. An execution is linearizable if there exists such a total order of operations where: (i) the total order respects the real time order of the non-concurrent operations and (ii) the behavior of the operations are consistent with the sequential specification of the data structure.

Consider, for example, the two execution histories of a concurrent list data structure in Figure 1. The execution history in Figure 1a is linearizable. The total order (linearization) of operations which orders the operation $addAll(1, 2)$ before $clear()$ produces the same outcomes as the outcomes recorded in the history. In this order $isEmpty()$ returns the value *true*, which matches its sequential specification. However, the execution in Figure 1b is not linearizable due to the interleavings between $addAll(1, 2)$ and $toString()$ methods. Neither of the sequential executions $[addAll(1, 2); toString()]$ and $[toString(); addAll(1, 2)]$ result in the outcome [1] for $toString()$ obtained in the execution history.

Given an execution trace, one can check if it is linearizable by finding an appropriate schedule of the operations and checking that the outcomes of the schedule conform to the sequential specification of the operations. Unfortunately, Gibbons and Korach [13] showed that checking a single execution history for linearizability is already NP-complete. Thus, linearizability checking tools (such as Burckhardt et al. [6], Horn and Kroening [16], Lowe [18], Vechev et al. [23]) based on exhaustively exploring all possible schedules of a history remain limited to small histories with few operations.

In this paper, we propose a prioritization of the search space of schedules using the notion of *linearizability depth*. Empirically, many execution histories can be shown to be



(a) A linearizable execution with a total order [1, 3, 2]

(b) A non-linearizable execution

Figure 1. Example executions of a concurrent list data structure operations

linearizable by carefully scheduling a small number of operations relative to each other, no matter how the other operations are scheduled. We capture this observation by defining the twin notions of *strong hitting schedules* and *linearizability depth*. Informally, a schedule *strongly hits* a sequence of d operations $(o_1, \ldots, o_d)$ if it orders these operations according to this sequence and, moreover, schedules each operation o_j as late as possible in the schedule. The linearizability depth of an execution history is d if there exist d operations $(o_1, \ldots, o_d)$ such that any schedule strongly hitting these operations is a witness for linearizability.

A set of schedules forms a *strong d-hitting family* if for every d tuple of operations, there is some schedule in the family that is strongly d-hitting for this tuple. Clearly, a strong d-hitting family is sufficient to check linearizability of any execution history of linearizability depth at most d. Our main contribution is an algorithm to construct a strong d-hitting family of schedules for a given execution history and a given d. In particular, we show that the size of such a family can be bounded by $O(mn^{d-1})$ for execution histories with m threads and n operations.

The notion of strongly hitting schedules was recently introduced in Ozkan et al. [20] (see also Chistikov et al. [7]), in the context of random testing of asynchronous distributed programs. We adapt the notion to the setting of linearizability in two ways. First, unlike Ozkan et al. [20] which provides a probabilistic guarantee for random testing, we provide a strong d-hitting family guaranteed to prove linearizability (up to depth d). Our construction uses the combinatorial insights in the proofs of the probabilistic guarantee, such as indexing schedules using chain partitions of the underlying partial ordering of events. Second, unlike Ozkan et al. [20] which constructs an unknown partial order dynamically and hence computes chain partitions online, we can use the natural chain partition induced by the underlying threads in an execution trace. This simplifies some combinatorial aspects of Ozkan et al. [20] and also allows a more precise upper bound.

Overall, our construction gives a technique that prioritizes searching for linearizability witnesses in "small linearizability depth first" order. To show the effectiveness of this strategy, we have evaluated our linearizability checker on a benchmark of execution histories of Java's `java.util.concurrent` package, collected by test harnesses of the Violat framework [9] for testing linearizability. For these benchmarks, we show that exploring a strong d-hitting family with $d \leq 5$ suffices for showing linearizability for 99.9% of the linearizable traces (8660 out of 8673, or 99.9%). For 99.5% of linearizable traces, the value of $d \leq 4$ suffices, and for 93.3% of linearizable traces the value as low as $d \leq 2$ suffices! Thus, most traces could be proved to be linearizable using strong 5-hitting families. In our experiments, the size of strong d-hitting families were smaller than the theoretical upper bound and much smaller than the total possible number of schedules.

2 Motivating Example

We motivate linearizability depth through an example. Consider the execution history in Figure 2 obtained from a test that invokes random operations on `java.util.concurrent.ConcurrentLinkedQueue`, the Java implementation of an unbounded thread-safe queue based on linked nodes. Each line in the figure represents the execution on a thread, with the passage of time from left to right. The thick segments show the duration between the invocation and return of an operation. The operations with overlapping durations run concurrently to each other. The operations in the history are provided with the arguments written in parentheses and the results written after a colon. The descriptions of the operations are given in Figure 3.

In order to demonstrate linearizability of an execution history, one needs to find a *witnessing schedule* (a total order of operations) that is consistent with the history: (1) if the first operation returns before the second one is invoked, the first one should appear before the second one in the schedule, and (2) the return values should be consistent with the sequential specification of a queue. For the execution history in Figure 2, this means the witnessing schedule needs to order the operations 0–*toString*(), 1–*poll*(), 2–*size*(), 3–*isEmpty*(), 6–*removeAll*(1, 0), 8–*retainAll*(0, 0), 9–*poll*(), 10–*toArray*(), and 12–*containsAll*(1, 1) before the operation 5–*addAll*(1, 2), since their return values are consistent with their execution on an empty queue, and 5–*addAll*(1, 2) populates the queue with elements. These dependencies are shown with red arrows in Figure 2. A naive enumeration of schedules would need to iterate over 1,004,640 total schedules in search for one of 134,400 witnessing schedules.

While it might seem that a witnessing schedule must consider many pairwise interactions between concurrent calls, the linearizability depth of this execution history is just 1

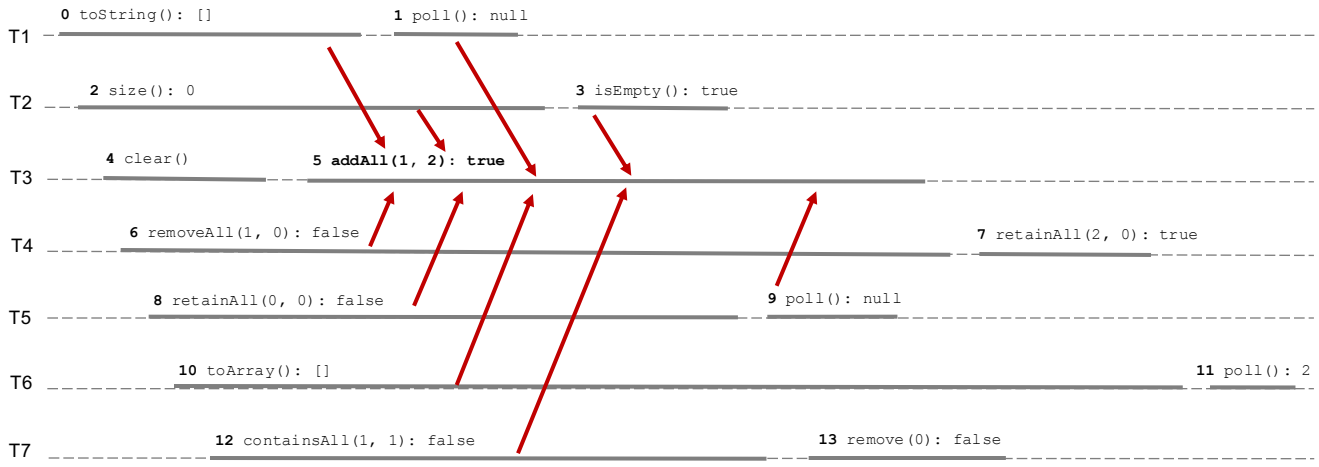


Figure 2. An execution history whose linearizability can be shown by a schedule satisfying all the constraints shown by the red arrows. There are 134,400 such schedules out of 1,004,640 total schedules. The linearizability depth of the history is 1 and a strong 1-hitting family with only 7 schedules is guaranteed to contain a witnessing schedule.

addAll	– Inserts all the arguments into the queue
clear	– Removes all the elements in the queue
containsAll	– Returns *true* if all the arguments are contained in the queue
isEmpty	– Returns *true* if the queue is empty
poll	– Retrieves and removes the head of the queue if the queue is non-empty and returns *null* otherwise
remove	– Removes the argument from the queue and returns *true* if it modifies the queue
removeAll	– Removes all the arguments from the queue and returns *true* if it modifies the queue
retainAll	– Retains only the elements in the argument and returns *true* if it modifies the queue
size	– Returns the size of the queue
toArray	– Returns an array representation of the queue
toString	– Returns a string representation of the queue

Figure 3. Operations in the history in Figure 2

and hence it is 1-linearizable. This is because any schedule which delays *addAll* maximally will schedule it after all the other operations and hence provide a witnessing schedule.

Now consider a family of schedules constructed in the following way. For each i with $1 \leq i \leq 7$, we delay scheduling operations from thread i until there are no other concurrent operations to schedule. The constructed family contains 7 schedules. It has the property that for each operation o there is a schedule where o appears at the latest possible moment. We call a set of schedules with this property a *strong* 1*-hitting family* of schedules. For any trace of linearizability depth 1, there is some schedule in a strong 1-hitting family that is a witnessing schedule for that trace. In particular, in the schedule for $i = 3$, the operation $5{-}addAll(1, 2)$ appears at the latest possible moment, which makes this schedule a witnessing schedule.

In general, witnessing schedules may need to enforce orderings relations between more than two operations. This leads to a generalization of execution histories of linearizability depth d, for $d > 1$, for which a witnessing schedule needs to order d operations relative to each other and maximally delayed with respect to all other operations. We call these histories d-linearizable histories. Consider a modification to the history in Figure 2 with an additional requirement that the operation $6{-}removeAll$ needs to be scheduled in between $0{-}toString$ and $5{-}addAll$. This modified history is called 2-linearizable since any schedule which schedules 6 before 5 and also maximally delays these two operations act as a witness schedule.

Correspondingly, we need to construct strong d-hitting families with a higher value of the parameter d. A strong 2-hitting family (and, in general, a strong d-hitting family) of schedules ensures that for each pair (or d-tuple) of events, there is a schedule which orders these events in the relative order given by the tuple and delays them maximally. Clearly, such a family is guaranteed to contain witness schedules for every history of linearizability depth 2 (or d, in general). We show an explicit construction of strong d-hitting families in Section 4.

In the rest of the paper, we make the following contributions. We formally define the notion of d-linearizability (for a parameter $d \geq 1$) for execution histories. We show a general algorithm to construct a strong d-hitting family of schedules and use it as the basis of a linearizability checker which prioritizes witness schedules sufficient to show linearizability for histories with low linearizability depth. For an execution history with m threads and n operations, the size of the family is theoretically bounded by $O(mn^{d-1})$. We empirically demonstrate that most linearizable histories from

a standard Java benchmark have a witness schedule already in a strong d-hitting family for $d \leq 5$ and moreover that the size of a strong d-hitting family is often much smaller than the theoretical bound.[1]

3 Linearizability

As our goal is to check linearizability of a concurrent data structure, we are only interested in the interactions between that concurrent data structure and the client program that invokes the operations of the data structure. We formalize these interactions using *execution histories*, following the standard definitions from the literature [12, 15].

Definition 3.1 (Action)**.** An *action* is either a tuple of the form $\langle id, call(m), t\rangle$, representing an *invocation* of the method m in the thread t, or a tuple of the form $\langle id, ret(m), t, r\rangle$, representing a *return* of the method m in the thread t with the result r (we use $r = \perp$ if the method does not have a return value). In both cases the action has an *action identifier* id.

For example, $\langle 1, call(poll), 0\rangle$ and $\langle 1, ret(poll), 0, null\rangle$ are the two actions for the invocation and return of the method $1{-}poll()$ in Figure 2.

Definition 3.2 (History)**.** A *history* is a sequence of actions $h = [e_1, e_2, \ldots, e_n]$. A history is *sequential* if (i) it starts with an invocation and ends with a return, (ii) every invocation is immediately followed by a corresponding return, and (iii) every return other than the last is immediately followed by an invocation. A *thread subhistory* of a history h is a subsequence of all actions executed on the same thread. A history is *well-formed* if all of its thread subhistories are sequential.

For example, the history in Figure 2 involving the actions in the first three threads is

$[\langle 0, call(toString), 1\rangle, \langle 2, call(size), 2\rangle, \langle 4, call(clear), 3\rangle,$
$\langle 4, ret(clear), 3, \perp\rangle, \langle 5, call(addAll(1, 2)), 3\rangle,$
$\langle 0, ret(toString), 1, []\rangle, \langle 1, call(poll), 1\rangle, \langle 1, ret(poll), 1, null\rangle,$
$\langle 2, ret(size), 2, 0\rangle, \langle 3, call(isEmpty), 2\rangle,$
$\langle 3, ret(isEmpty), 2, true\rangle, \langle 5, ret(addAll(1, 2)), 3, true\rangle]$

Note that all three thread subhistories are sequential, hence the history is well-formed. In the rest of the paper we assume all histories are well-formed. With this assumption, the following notion is well-defined.

Definition 3.3 (Operation)**.** An *operation* is a pair $op_{id} = \langle callAction, retAction\rangle$, where $callAction = \langle id, call(m), t\rangle$ and $retAction = \langle id, ret(m), t, r\rangle$ are the matching invocation and return actions.

[1] Note that this does not imply that the execution histories have linearizability depth d: it is possible that a schedule in a strong d-hitting family "accidentally" witnesses linearizability for a history with higher depth. In other words, strongly d-hitting families are sufficient but not necessary to witness linearizability for a history of depth d.

As an example, for the method *size* in Figure 2, we write $op_2 = \langle\langle 2, call(size), 2\rangle, \langle 2, ret(size), 2, 0\rangle\rangle$. We will continue using shorthand notations $2{-}size()$, $2{-}size$, or simply 2 to refer to the same operation.

Definition 3.4 (Executed-before)**.** Given a history h, an operation $o_1 = \langle callAction_1, retAction_1\rangle$ is *executed before* another operation $o_2 = \langle callAction_2, retAction_2\rangle$ if $retAction_1$ appears before $callAction_2$ in h. If neither of the two operations is executed before the other, we say the operations are *concurrent*.

For example, the operation $0{-}toString()$ is executed before $1{-}poll()$ in Figure 2, and the operations $0{-}toString()$, $2{-}size()$, and $4{-}clear()$ are concurrent to each other.

For a given history, the set of all operations Op together with the *executed-before* relation forms a *partially ordered set* or *poset*.

Definition 3.5 (Poset)**.** A partially ordered set (poset) is a pair $\mathcal{P} = (X, \leq)$, where X is a set and $\leq$ is a partial order (i.e., reflexive, anti-symmetric, and transitive binary relation) on X. We write $x < y$ if $x \leq y$ and $x \neq y$. We also write $x \geq y$ and $x > y$ if $y \leq x$ and $y < x$, respectively.

Definition 3.6 (Maximal element)**.** A *maximal* element of a poset $\mathcal{P} = (X, \leq)$ is an element $x \in X$ that is not strictly smaller than any other element in X, i.e., $\forall y \in X : x \not< y$.

For example, in Figure 2, the operation $11{-}poll()$ is maximal in the poset of operations. If we exclude that operation, we have $7{-}retainAll(2, 0)$, $10{-}toArray()$, and $13{-}remove(0)$ as the maximal elements.

Definition 3.7 (Linear extension)**.** A *linear extension* of a poset $\mathcal{P} = (X, \leq_{\mathcal{P}})$ is a poset $\mathcal{Q} = (X, \leq_{\mathcal{Q}})$ such that $\forall x, y \in X : x \leq_{\mathcal{P}} y \implies x \leq_{\mathcal{Q}} y$, and moreover $\leq_{\mathcal{Q}}$ is a *total order*, i.e., $\forall x, y \in X : x \leq_{\mathcal{Q}} y$ or $y \leq_{\mathcal{Q}} x$. We will often use the word *schedule* instead of *linear extension*.

We are now ready to define the notion of linearizability of a history.

Definition 3.8 (Linearizability)**.** A history is *linearizable* if there exists a schedule s of its operations that is consistent with the sequential specification of the data structure, i.e., if the operations were executed sequentially according to s, they would return the same results as the ones recorded in the history.

In a history with n operations, a naive search for a schedule witnessing linearizability enumerates $n!$ schedules in the worst case. It is unlikely there is a significantly better approach, since the problem of deciding linearizability is NP-complete [15]. However, in practice we may be able to quickly discharge positive instances, that is, quickly find a schedule witnessing linearizability when there is one, by cleverly prioritizing some schedules over the others. We discuss one such approach in the next section.

4 Prioritization of Schedules to Check Linearizability

In the example in Section 2, a single operation, 5–*addAll*(1, 2), needed to be delayed relative to the other concurrent operations in order to ensure that the schedule witnesses linearizability. More generally, linearizability may depend on $d \geq 1$ operations being appropriately delayed, leading to the following definitions.

Definition 4.1 (Strong hitting [20])**.** Let $d \geq 1$ be an integer and $\mathcal{P}$ a poset. We say a schedule α for $\mathcal{P}$ *strongly hits* a d-tuple of elements $\langle x_0, \ldots, x_{d-1} \rangle$ if for every $y \in \mathcal{P}$, $y \geq_\alpha x_i$ in α for some $i \in \{0, \ldots, d-1\}$ implies $y \geq x_j$ in $\mathcal{P}$ for some $j \geq i$.

Informally, a schedule strongly hits a d-tuple $\langle x_0, \ldots, x_{d-1} \rangle$ if the only reason for an element y to appear after x_i in the schedule is that $y \geq x_j$ in the poset for some $j \geq i$. In other words, each x_i is maximally delayed in the schedule relative to the other elements in the tuple.

Definition 4.2 (d-Linearizability)**.** Let $d \geq 1$ be an integer. A history is d-linearizable if there exist operations $o_0, \ldots, o_{d-1}$ such that every schedule that strongly hits $\langle o_0, \ldots, o_{d-1} \rangle$ is a witness to linearizability. The smallest d such that the history is d-linearizable is the *linearizability depth* of that history.

The history in the example in Section 2 is 1-linearizable. The modified version of the history, where we additionally required the operation 6–*removeAll* to be scheduled in between operations 0–*toString* and 5–*addAll*, is 2-linearizable, since the ordering requirement is ensured by strongly hitting the pair of operations $\langle 6, 5 \rangle$.

Proposition 4.3. *The linearizability depth of a linearizable history with n operations is at most n.* □

Our approach to checking linearizability of a given history is centered around the notion of *strong d-hitting families* [20], which are sets of schedules that strongly hit every d-tuple of operations. For increasing values of d, we enumerate a strong d-hitting family, checking if any of the schedules is a linearizability witness. Once d reaches the linearizability depth of the history, by definition the strong d-hitting family is guaranteed to contain a linearizability witness. Proposition 4.3 ensures that we can stop the search once d reaches the number of operations in the history. Note that for a linearizable history, the search may stop before reaching the linearizability depth: we may get lucky and stumble upon a linearizability witness for smaller values of d.

The practical efficiency of the approach follows from two factors. First is our observations that for concurrent data structures in Java's `java.util.concurrent` package most linearizable histories have small linearizability depth. Second is the fact that for small values of d we can effectively construct strong d-hitting families of small size.

4.1 Strong Hitting Families

We first formally define strong hitting families and then give an algorithm to construct them. We follow the approach presented in Ozkan et al. [20].

Definition 4.4 (Strong hitting family [20])**.** Let $d \geq 1$ be an integer and $\mathcal{P}$ a poset. A set of schedules $\mathcal{F}$ is called a *strong d-hitting family* for $\mathcal{P}$ if for every d-tuple of elements in $\mathcal{P}$ there is a schedule in $\mathcal{F}$ that strongly hits it.

For a poset of n elements, there is a simple way to construct a strong d-hitting family: For each d-tuple of elements $\langle x_0, \ldots, x_{d-1} \rangle$, simply construct a dedicated schedule that maximally delays these d elements while respecting their relative order given by the tuple. The strong d-hitting family constructed in this way contains $O(n^d)$ schedules. Ozkan et al. [20] show that the construction can be improved by considering a partition of the poset into *chains*.

Definition 4.5 (Chain partition)**.** Let $\mathcal{P} = (X, \leq)$ be a poset. A subset $\lambda \subseteq X$ is a *chain* if it is totally ordered, that is, $\forall x, y \in \lambda : x \leq y$ or $y \leq x$. A *chain partition* is a decomposition of the poset as $X = \lambda_1 \cup \ldots \cup \lambda_m$, where each λ_i is a non-empty chain and the chains are pairwise disjoint, that is, $\lambda_i \cap \lambda_j = \emptyset$ whenever $i \neq j$.

Since the elements in a chain are totally ordered, they can all be strongly hit by a single schedule: There are no concurrent elements in a chain, so maximally delaying one element does not prevent the other elements to be maximally delayed as well. This observation is the basis of the construction by Ozkan et al. Instead of constructing one schedule for each d-tuple of elements, we construct one schedule for each chain and a $(d-1)$-tuple of elements: The elements in the chain are maximally delayed before the elements in the $(d-1)$-tuple, and the elements in the $(d-1)$-tuple are maximally delayed while respecting the relative ordering given by the tuple, as before. If there are m chains in the chain partition of the poset, the construction gives a strong d-hitting family of size $O(mn^{d-1})$—an improvement over $O(n^d)$.

In our setting, the poset of operations for a given history can be naturally partitioned into chains based on the threads. Recall that by Definition 3.2 all thread subhistories of a well-formed history are sequential. By Definition 3.4, this directly translates to operations on each thread forming a chain. For example, the history in Figure 2 can be partitioned into seven chains: $\lambda_1 = [0, 1]$, $\lambda_2 = [2, 3]$, $\lambda_3 = [4, 5]$, $\lambda_4 = [6, 7]$, $\lambda_5 = [8, 9]$, $\lambda_6 = [10, 11]$, and $\lambda_7 = [12, 13]$.

Given a poset of n operations partitioned into m thread-based chains and a linearizability depth parameter d, Algorithm 1 constructs a strong d-hitting family for the poset. The algorithm is based on the proof of Theorem 6 in Ozkan et al. [20]. The main idea is to maintain a family of partial schedules indexed by the *schedule indices* of the form $\langle tid, \langle x_1, \ldots, x_{d-1} \rangle \rangle$, where $tid \in \{1, \ldots, m\}$ is a thread identifier and $x_i \in \{1, \ldots, n\}$ for $1 \leq i < d$ are distinct numbers

ALGORITHM 1: Constructs a strong d-hitting family of schedules for a poset of operations and a linearizability depth parameter d

Input: poset of operations $(Op, \leq)$
Input: depth parameter d
Output: d-hitting family of schedules for $(Op, \leq)$

```
1  n ← number of operations in Op
2  m ← number of threads in Op
3  scheduleIndices ← generateIndices(n, m, d)
4  for schIndex in scheduleIndices do
5  |  schedules[schIndex] ← []
6  for op in topologicalSort(Op) do
7  |  for schIndex = ⟨tid, ⟨x_1, ..., x_{d-1}⟩⟩ in scheduleIndices do
8  |  |  schedule ← schedules[schIndex]
9  |  |  ⟨x_{i_1}, ..., x_{i_l}⟩ ← all x from ⟨x_1, ..., x_{d-1}⟩ s.t. operation
   |  |    op' with op'.id = x has already been inserted
10 |  |  ⟨op_{i_1}, ..., op_{i_l}⟩ ← operations with ids ⟨x_{i_1}, ..., x_{i_l}⟩
11 |  |  if op.id in ⟨x_1, ..., x_{d-1}⟩ then
12 |  |  |  i ← the index of op.id in ⟨x_1, ..., x_{d-1}⟩
13 |  |  |  if l = 0 or i > i_l or op > op_{i_l} then
14 |  |  |  |  insert op at the end of schedule
15 |  |  |  else
16 |  |  |  |  j ← the least index s.t. i < i_j and op is
   |  |  |  |    concurrent with all of op_{i_j}, op_{i_{j+1}}, ..., op_{i_l}
17 |  |  |  |  insert op right before op_{i_j} in schedule
18 |  |  else if op.tid = tid then
19 |  |  |  if l = 0 or op > op_{i_l} then
20 |  |  |  |  insert op at the end of schedule
21 |  |  |  else
22 |  |  |  |  j ← the least index s.t. op is concurrent with all
   |  |  |  |    of op_{i_j}, op_{i_{j+1}}, ..., op_{i_l}
23 |  |  |  |  insert op right before op_{i_j} in schedule
24 |  |  else
25 |  |  |  op' ← the last operation in schedule s.t. op' < op
26 |  |  |  insert op right after op' in schedule
27 return schedules
```

representing operation identifiers. Thus each schedule index corresponds to a choice of a chain and $d-1$ elements discussed earlier. In the pseudocode we assume that an unspecified function *generateIndices* (line 3) generates all possible schedule indices based on n, m, and d. The schedules are initially empty (lines 4–5). In the main loop (lines 6–26), the algorithm iterates over the operations and inserts them into the schedules in the position determined by the schedule index. At the end, the construction ensures that for every d-tuple of operations $\langle op_0, \ldots, op_{d-1}\rangle$, the schedule indexed by the schedule index $\langle op_0.tid, \langle op_1.id, \ldots, op_{d-1}.id\rangle\rangle$ strongly hits $\langle op_0, op_1, \ldots, op_{d-1}\rangle$.

For the correctness of the algorithm it is important that the operations are iterated over in an "upgrowing" manner, that is, in each iteration the operation should be maximal among the operations which are already inserted into the schedule. This requirement is equivalent to iterating over the operations according to an arbitrary schedule, which can be ensured by topologically sorting the operations. In the pseudocode we assume there is an unspecified function *topologicalSort* (line 6) that topologically sorts the given poset of operations.

Let us now discuss how an operation op is inserted into *schedule* defined in line 8 and indexed by the schedule index $schIndex = \langle tid, \langle x_1, \ldots, x_{d-1}\rangle\rangle$. There are three cases to consider. The first case corresponds to strongly hitting op since its identifier is in $(x_1, \ldots, x_{d-1})$, the second case corresponds to strongly hitting op since it is in the chain tid, and the third case corresponds to the case where op need not to be strongly hit.

Let $x_{i_1}, \ldots, x_{i_l}$ with $i_1 < \ldots < i_l$ be the operation identifiers among $x_1, \ldots, x_{d-1}$ of the operations that were inserted into the schedules before op; let $op_{i_1}, \ldots, op_{i_l}$ be the corresponding operations to the identifiers $x_{i_1}, \ldots, x_{i_l}$. The invariant maintained by the algorithm is that for every operation op_0 with $op_0.tid = tid$, *schedule* strongly hits the tuple $\langle op_0, op_{i_1}, \ldots, op_{i_l}\rangle$.

1. In the first case (line 11), the operation identifier $op.id$ is among $x_1, \ldots, x_{d-1}$, meaning that op needs to be strongly hit, that is, placed as late as possible in the schedule. Let i be the index such that $op.id = x_i$. If op should be scheduled after all of $op_{i_1}, \ldots, op_{i_l}$ (that is, $l = 0$ or $i > i_l$ in line 13), or if op cannot be scheduled before op_{i_l} (that is, $op > op_{i_l}$ in line 13), we insert op at the end of the schedule (line 14). Otherwise $i < i_l$ and op is concurrent with op_{i_l}. Let j be the least index such that $i < i_j$ and op is concurrent with all of $op_{i_j}, op_{i_{j+1}}, \ldots, op_{i_l}$. We insert op immediately before op_{i_j} in the schedule (line 17). Note that the insertion is possible in all cases because of the invariant and the fact that op is maximal among the operations previously inserted into the schedule.
2. In the second case (line 18), the operation op is in the thread tid (that is, $op.tid = tid$), thus it needs to be strongly hit. If it cannot be scheduled before op_{i_l} (that is, $l = 0$ or $op > op_{i_l}$ in line 19), we insert op at the end of the schedule (line 20). Otherwise op is concurrent with op_{i_l}. Let j be the least index such that op is concurrent with all of $op_{i_j}, op_{i_{j+1}}, \ldots, op_{i_l}$. We insert op immediately before op_{i_j} in the schedule (line 23). Again, the insertion is possible in all cases because of the invariant and the maximality of op.
3. In the third case (line 24), the operation op is not to be strongly hit, so we schedule it as early in the schedule as possible. That is, we find the last operation op' such that $op' < op$ and schedule op immediately after op'.

It is not difficult to see that the invariant of the algorithm is preserved in all three cases, which ensures correctness.

The size of the generated strong d-hitting family. The algorithm generates one schedule (not necessarily unique) for each schedule index $\langle tid, \langle x_1, \ldots, x_{d-1} \rangle \rangle$. There are m choices for tid and $\binom{n}{d-1}(d-1)!$ choices for the tuple $\langle x_1, \ldots, x_{d-1} \rangle$. Hence, the size of the family is bounded by $O(mn^{d-1})$.

5 Implementation

Our implementation collects a set of concurrent execution histories on concurrent data structures and checks their linearizability by comparing the collected outcomes to the outcomes obtained in the d-hitting families of schedules.

Concurrent data structures checked for linearizability. We check the linearizability of the Java library of concurrent collections, specifically the data structures in the package `java.util.concurrent`, which contain implementations of queues, deques, sets, and key-value maps (given in Table 1). Some methods in this library (e.g., those implementing non-basic data type operations such as *addAll* for a concurrent queue) admit non-atomic behaviors resulting in non-linearizable execution histories.

Testing data structure implementations. We generate the tests using the Violat [9] framework, which stress tests the concurrent data structures with a high level of parallelism. It automatically generates tests which run a number of threads invoking the methods of the concurrent data structures. For each test execution, Violat creates a history file keeping the information about which threads run which operations, the operation arguments, the timestamps of the invocation and return of the operations, and the return values.

We build the poset for each history file and check their linearizability by checking whether schedules from a strong d-hitting family produce the recorded outcomes. Note that the main goal of the Violat framework is to *expose* linearizability violations. It checks the linearizability of a history by checking whether the obtained outcomes of the operations are contained in a precomputed set of acceptable return-value outcomes, which might label a non-linearizable history as linearizable, but the reported violations are real violations.

Constructing strong hitting families. We construct the strong d-hitting family of schedules following Algorithm 1, and test the data structures on the set of distinctly generated schedules.

Automatically generating tests. We automatically generate a tester class for each history, which runs the schedules of operations to be explored for checking the linearizability of that history. Briefly, a test class for a history encapsulates the set of schedules to be tested (the strong d-hitting family) and the expected outcomes of the operations. It declares a separate method for each schedule to be tested. Each of these methods invoke a particular order of the operations and collect the outcomes (i.e., the return values of the methods or the thrown exceptions). Finally, they check whether the resulting outcomes are the same with the corresponding method outcomes in the history. If this is the case, the history is found to be linearizable with a witness schedule. We use Java parser library [17] to programmatically create the tester classes.

6 Experiments

In this section, we present our experimental work which empirically shows that exploring schedules from a strong d-hitting family for small values of d is sufficient to show the linearizability of a linearizable history.

We evaluate our approach on the execution histories generated using the Violat framework [9], which stress tests the Java concurrent collection library. The summary of the collected histories is given in Table 1. For each data structure, the column **#Histories** shows the number of histories under analysis and **#Ops** shows the number of operations in each history. The columns **max(c)**, μ**(c)**, and σ**(c)** list the maximum, average, and the standard deviation of the level of concurrency (i.e. the maximum number of mutually concurrent operations). For example, the level of concurrency is $\mathbf{c} = 7$ for the history in Figure 2. The columns **max(cp)**, μ**(cp)**, and σ**(cp)** list the maximum, average, and the standard deviation of the number of concurrent operation pairs. The number of concurrent operation pairs refers to the number of operation pairs which are concurrent and thus may be scheduled in any order. For the history in Figure 2, the number of concurrent operation pairs $\mathbf{cp} = 48$. As the table shows, the histories checked for linearizability are highly concurrent, which makes it impractical to explore all possible schedules.

For each non-sequential history, i.e., where the operations are not totally ordered, we constructed strong d-hitting families of schedules and automatically generated test classes running these schedules. The size of the d-hitting families of schedules for the set of histories collected from the data structures is given in Table 2. Each row summarizes the size of the strong d-hitting families for the concurrent data type given in the first column. For the increasing values of d, the table lists the maximum (max), average (μ), and the standard deviation (σ) of the number of schedules constructed for the d-hitting family. We count only the distinct schedules in each d-hitting family (recall that Algorithm 1 may produce the same schedule multiple times), so the size of the strong d-hitting family varies depending on the number of concurrent operation pairs.

The size of the hitting families of schedules grows exponentially in the parameter d. In some of our examples, the generated number of schedules for parameter values $d \geq 7$ approaches one million. Still, this number is small in comparison to the number of all possible schedules.

Table 1. Data structures in `java.util.concurrent` package checked for linearizability

Class name	#Histories	#Ops	max(c)	μ(c)	σ(c)	max(cp)	μ(cp)	σ(cp)
ArrayBlockingQueue	1029	14	7	2.21	0.91	52	3.94	7.71
ConcurrentHashMap	1055	14	2	2.00	0.00	13	8.42	1.93
ConcurrentLinkedDeque	681	12	6	2.37	0.78	34	4.26	4.85
ConcurrentLinkedQueue	780	14	7	2.27	0.85	49	4.21	7.01
ConcurrentSkipListMap	1506	14	7	2.28	0.56	48	4.04	3.62
ConcurrentSkipListSet	417	18	6	2.54	1.28	63	10.00	15.77
LinkedBlockingDeque ‘	1131	8	4	2.63	0.82	18	5.99	4.57
LinkedBlockingQueue	1199	9	3	2.54	0.50	15	7.28	3.40
LinkedTransferQueue	1355	15	5	2.10	0.45	43	4.79	5.14
PriorityBlockingQueue	1311	14	7	2.32	1.04	59	4.94	9.57

Table 2. The maximum, average, and the standard deviation of the number of schedules generated for the increasing values of d for the sets of histories

	#Sch ($d = 1$)			**#Sch ($d = 2$)**			**#Sch ($d = 3$)**			**#Sch ($d = 4$)**			**#Sch ($d = 5$)**		
Class name	*max*	μ	σ	*max*	μ	σ	*max*	μ	σ	*max*	μ	σ	*max*	μ	σ
ArrayBlockingQueue	7	2	1	76	6	12	656	21	83	4493	96	483	24639	412	2290
ConcurrentHashMap	2	2	0	16	11	3	66	26	10	157	39	23	268	47	34
ConcurrentLinkedDeque	6	3	1	57	7	8	378	17	42	1898	42	171	7321	98	568
ConcurrentLinkedQueue	7	2	1	74	7	11	580	22	78	3718	88	446	19168	351	2101
ConcurrentSkipListMap	7	3	1	70	7	6	534	13	33	3352	27	162	17028	65	687
ConcurrentSkipListSet	6	3	1	89	15	24	875	94	222	6660	568	1499	40200	2876	8072
LinkedBlockingDeque	4	3	1	26	9	7	113	22	27	355	42	66	696	60	110
LinkedBlockingQueue	3	3	0	20	10	5	76	21	16	175	33	30	298	39	42
LinkedTransferQueue	5	2	1	60	7	8	474	17	56	2736	55	300	12038	187	1238
PriorityBlockingQueue	7	2	1	85	7	14	805	32	117	6043	167	757	36825	802	4057

We checked the linearizability of the histories in Table 1 by running the hitting families of schedules for increasing values of d, and we counted linearizable histories whose linearizability can be shown by a strong d-hitting family of schedules. Table 3 provides these results. Each row summarizes the results collected from the histories of a concurrent data type. The columns **#Hists** and **#Lin** list the number of histories (after the elimination of sequential histories) and the number of linearizable histories, respectively. For values of d from 1 to 5, the other columns show the number (#Lin) and the percentage (%) of linearizable histories whose linearizability can be shown by a strong d-hitting family.

The results show that for a large portion of linearizable histories linearizability can be shown by exploring a strong d-hitting family for $d \leq 5$. In particular, $d \leq 5$ suffices for all linearizable histories collected from `ConcurrentHashMap`, `ConcurrentLinkedDeque`, `ConcurrentLinkedQueue`, `ConcurrentSkipListMap`, `LinkedBlockingDeque`, `LinkedBlockingQueue`, and `PriorityBlockingQueue`. For a few histories whose linearizability cannot be shown with $d = 5$, we increased the value of d. In particular, with $d = 6$ we can show linearizability of two additional histories for `ArrayBlockingQueue`, three for `ConcurrentSkipListSet`, and two for `LinkedTransferQueue`. With $d \leq 7$, almost all linearizable histories in the experiments are shown to be linearizable. The percentages of linearizable histories are summarized and shown in Figure 4.

The experimental results presented in this section demonstrate the practicality of prioritizing the search by the linearizability depth. Small values of d suffice for discharging the vast majority of linearizable histories, allowing one to switch to more expensive approaches for histories with high linearizability depth or non-linearizable histories.

Table 3. The total number of linearizable histories and the number of histories whose linearizability is shown by exploring strong d-hitting families

			$d = 1$		$d = 2$		$d = 3$		$d = 4$		$d = 5$	
Class name	**#Hists**	**#Lin**	#Lin	%	#Lin	%	#Lin	%	#Lin	%	#Lin	%
ArrayBlockingQueue	723	714	651	91.2	685	95.9	702	98.3	707	99.0	709	99.3
ConcurrentHashMap	1054	959	707	73.7	857	89.4	949	99.0	959	100	-	-
ConcurrentLinkedDeque	520	520	387	74.4	478	91.9	515	99.0	519	99.8	520	100
ConcurrentLinkedQueue	619	616	525	85.2	608	98.7	615	99.8	616	100	-	-
ConcurrentSkipListMap	1358	1321	1311	99.2	1319	99.8	1320	99.9	1321	100	-	-
ConcurrentSkipListSet	417	417	288	69.1	363	87.1	381	91.4	402	96.4	412	98.8
LinkedBlockingDeque	1106	967	743	76.8	920	95.1	963	99.6	967	100	-	-
LinkedBlockingQueue	1185	975	533	54.7	808	82.9	928	95.2	972	99.7	975	100
LinkedTransferQueue	1261	1229	878	71.4	1144	93.1	1203	97.9	1217	99.0	1226	99.8
PriorityBlockingQueue	982	955	793	83.0	910	95.3	935	97.9	949	99.4	955	100

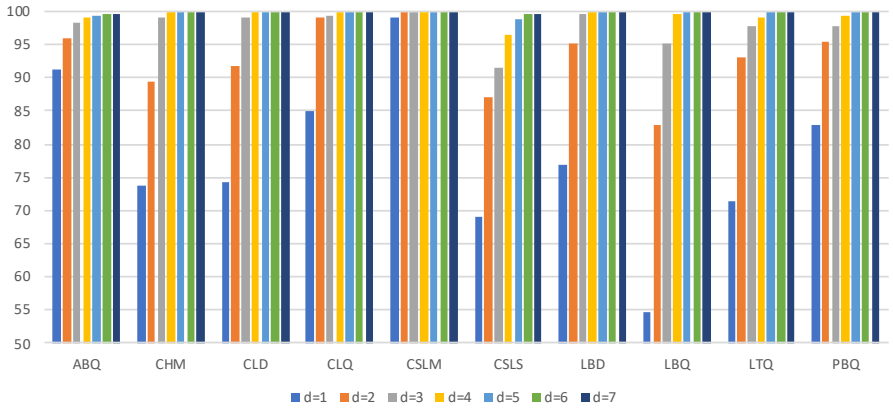


Figure 4. The percentage of linearizable histories whose linearizability can be shown by exploring schedules from a strong d-hitting family, for $1 \leq d \leq 7$.

7 Related Work

Gibbons and Korach [13] show that checking the linearizability of even a single execution history is NP-complete. Alur et al. [1] and Hamza [14] show that checking if all executions of a data structure implementation with a bounded number of threads are linearizable is EXPSPACE-complete. The problem is undecidable for an unbounded number of threads [3]. Decidability holds for certain data types such as stacks, queues, and registers [4]. Emmi et al. [11] and Emmi and Enea [10] show that for certain data types, called collection data types, linearizability is polynomial-time checkable. Although the collection types cover many important cases, this result does not hold for all concurrent data structures.

There is a large amount of work for showing linearizability of data structures [8]. In some lines of work, the programmer is required to provide the *linearization points* of the concurrent operations [2, 19, 21, 25]. Then, the operations are considered to take effect instantaneously at these points. This reduces the complexity of the analysis as it only considers a single schedule. In contrast, we focus on a fully automated approach for showing linearizability that does not require any annotations for linearization points.

Automated techniques usually explore the space of all schedules to find a witness. Wing and Gong [24] present an approach to checking linearizability by using a backtracking algorithm. Given a history and a specification, the algorithm tries to linearize the history recursively starting from a minimal operation. If the outcome of the linearized minimal operation is consistent with the specification, it continues with the linearization with a minimal operation in the remaining part of the history. Otherwise, it backtracks and tries to linearize another operation. Lowe [18] improves the

algorithm so that it eliminates redundant search when it encounters a configuration equivalent to another one analyzed earlier. He also suggests a variant of the algorithm, which linearizes a set of concurrent minimal operations instead of a single one. *P-composability* [16] generalizes the locality principle of linearizability to operations on the same concurrent object and exploits this in the backtracking linearization algorithm. Paraglider [22] uses the SPIN model checker to enumerate all possible linearizations of a history and checks each one against a given sequential specification. Line-up [6] does not take specifications as input but generates them by enumerating all sequential behaviors of the operations. Then, it checks whether the concurrent executions correspond to a sequential execution of the same operations. Our paper structures the search on the space of schedules using linearizability depth.

Owing to the large number of possible linearizations of execution histories, such enumerative approaches become impractical for histories with more than a few operations. For a scalable detection of violations, some lines of work employ approximations for linearizability to structure the search. Our work is in the spirit of Bouajjani et al. [5], which parametrically weakens the preorder on histories and detects violations up to that parameter. Violat [9] checks the notion of *atomicity of the test harnesses* using a precomputed set of possible outcomes, whose violation implies linearizability violation.

Different from the approximation approaches to detect violations, in this work we present a parametrized approximation analysis for proving linearizability of execution histories. Essentially, these two approaches can be applied complementarily to each other. One might show the linearizability of the histories using strong d-hitting families, which are not detected to have violations by some other methods.

Hitting families of schedules were introduced in Chistikov et al. [7] for testing asynchronous programs. Strong d-hitting families are defined in Ozkan et al. [20], which generalizes the construction of hitting families for any poset presented online. In asynchronous and distributed programs, the poset is formed by the asynchronous events created in the system at run time, where the ordering relation between them is based on the dependency model of the system. In the execution of these systems, the poset of events is not known before the execution and there may not be a clear chain decomposition. Ozkan et al. [20] use an online chain decomposition algorithm which partitions the poset into a number of total orders online. In our setting, we start with collected execution histories. This allows us to extract the partial order up front and the chain decomposition of the poset corresponds to partitioning operations by the threads.

Acknowledgments

This research was funded in part by the Deutsche Forschungsgemeinschaft (DFG) project 389792660-TRR 248 and by the European Research Council Grant Agreement No. 610150 (ERC Synergy Grant ImPACT (http://www.impact-erc.eu/)).

References

[1] Rajeev Alur, Kenneth L. McMillan, and Doron A. Peled. 2000. Model-Checking of Correctness Conditions for Concurrent Objects. *Inf. Comput.* 160, 1-2 (2000), 167–188. https://doi.org/10.1006/inco.1999.2847

[2] Daphna Amit, Noam Rinetzky, Thomas W. Reps, Mooly Sagiv, and Eran Yahav. 2007. Comparison Under Abstraction for Verifying Linearizability. In *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*. Springer, 477–490. https://doi.org/10.1007/978-3-540-73368-3_49

[3] Ahmed Bouajjani, Michael Emmi, Constantin Enea, and Jad Hamza. 2013. Verifying Concurrent Programs against Sequential Specifications. In *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*. Springer, 290–309. https://doi.org/10.1007/978-3-642-37036-6_17

[4] Ahmed Bouajjani, Michael Emmi, Constantin Enea, and Jad Hamza. 2015. On Reducing Linearizability to State Reachability. In *Automata, Languages, and Programming - 42nd International Colloquium, ICALP 2015, Kyoto, Japan, July 6-10, 2015, Proceedings, Part II*. Springer, 95–107. https://doi.org/10.1007/978-3-662-47666-6_8

[5] Ahmed Bouajjani, Michael Emmi, Constantin Enea, and Jad Hamza. 2015. Tractable Refinement Checking for Concurrent Objects. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*. ACM, 651–662. https://doi.org/10.1145/2676726.2677002

[6] Sebastian Burckhardt, Chris Dern, Madanlal Musuvathi, and Roy Tan. 2010. Line-up: a complete and automatic linearizability checker. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010*. ACM, 330–340. https://doi.org/10.1145/1806596.1806634

[7] Dmitry Chistikov, Rupak Majumdar, and Filip Niksic. 2016. Hitting Families of Schedules for Asynchronous Programs. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*. Springer, 157–176. https://doi.org/10.1007/978-3-319-41540-6_9

[8] Brijesh Dongol and John Derrick. 2015. Verifying Linearisability: A Comparative Survey. *ACM Comput. Surv.* 48, 2 (2015), 19:1–19:43. https://doi.org/10.1145/2796550

[9] Michael Emmi and Constantin Enea. 2017. Exposing Non-Atomic Methods of Concurrent Objects. *CoRR* abs/1706.09305 (2017). arXiv:1706.09305 http://arxiv.org/abs/1706.09305

[10] Michael Emmi and Constantin Enea. 2018. Sound, complete, and tractable linearizability monitoring for concurrent collections. *PACMPL* 2, POPL (2018), 25:1–25:27. https://doi.org/10.1145/3158113

[11] Michael Emmi, Constantin Enea, and Jad Hamza. 2015. Monitoring refinement via symbolic reasoning. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*. ACM, 260–269. https://doi.org/10.1145/2737924.2737983

[12] Ivana Filipović, Peter W. O'Hearn, Noam Rinetzky, and Hongseok Yang. 2010. Abstraction for concurrent objects. *Theor. Comput. Sci.* 411, 51-52 (2010), 4379–4398. https://doi.org/10.1016/j.tcs.2010.09.021

[13] Phillip B. Gibbons and Ephraim Korach. 1997. Testing Shared Memories. *SIAM J. Comput.* 26, 4 (1997), 1208–1244. https://doi.org/10.1137/S0097539794279614

[14] Jad Hamza. 2015. On the Complexity of Linearizability. In *Networked Systems - Third International Conference, NETYS 2015, Agadir, Morocco, May 13-15, 2015, Revised Selected Papers*. Springer, 308–321. https://doi.org/10.1007/978-3-319-26850-7_21

[15] Maurice Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (1990), 463–492. https://doi.org/10.1145/78969.78972

[16] Alex Horn and Daniel Kroening. 2015. Faster Linearizability Checking via P-Compositionality. In *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*. Springer, 50–65. https://doi.org/10.1007/978-3-319-19195-9_4

[17] JavaParser. 2018. JavaParser for Processing Java Code. http://javaparser.org/

[18] Gavin Lowe. 2017. Testing for linearizability. *Concurrency and Computation: Practice and Experience* 29, 4 (2017). https://doi.org/10.1002/cpe.3928

[19] Peter W. O'Hearn, Noam Rinetzky, Martin T. Vechev, Eran Yahav, and Greta Yorsh. 2010. Verifying linearizability with hindsight. In *Proceedings of the 29th Annual ACM Symposium on Principles of Distributed Computing, PODC 2010, Zurich, Switzerland, July 25-28, 2010*. ACM, 85–94. https://doi.org/10.1145/1835698.1835722

[20] Burcu Kulahcioglu Ozkan, Rupak Majumdar, Filip Niksic, Mitra Tabaei Befrouei, and Georg Weissenbacher. 2018. Randomized testing of distributed systems with probabilistic guarantees. *PACMPL* 2, OOPSLA (2018), 160:1–160:28. https://doi.org/10.1145/3276530

[21] Viktor Vafeiadis. 2010. Automatically Proving Linearizability. In *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*. Springer, 450–464. https://doi.org/10.1007/978-3-642-14295-6_40

[22] Martin T. Vechev and Eran Yahav. 2008. Deriving linearizable fine-grained concurrent objects. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*. Springer, 125–135. https://doi.org/10.1145/1375581.1375598

[23] Martin T. Vechev, Eran Yahav, and Greta Yorsh. 2009. Experience with Model Checking Linearizability. In *Model Checking Software, 16th International SPIN Workshop, Grenoble, France, June 26-28, 2009. Proceedings*. Springer, 261–278. https://doi.org/10.1007/978-3-642-02652-2_21

[24] Jeannette M. Wing and C. Gong. 1993. Testing and Verifying Concurrent Objects. *J. Parallel Distrib. Comput.* 17, 1-2 (1993), 164–182. https://doi.org/10.1006/jpdc.1993.1015

[25] Shao Jie Zhang. 2011. Scalable automatic linearizability checking. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*. ACM, 1185–1187. https://doi.org/10.1145/1985793.1986037

A Artifact appendix

A.1 Abstract

Our artifact contains the source code of our implementation and a data set of history files used in our experiments. Our implementation has two main parts for: *i*) generating strong hitting families of schedules of operations in a given history and *ii*) generation of Java source files such that each file invokes the strong hitting family of schedules for a particular history and depth d parameter. Then, we run these produced files to check the linearizability of the histories. We evaluated our implementation on sets of histories collected from different concurrent data structures in `java.util.concurrent` package. In the artifact, we provide all the history files used in our experimental work.

We also provide a main script file which reproduces the results in our paper. For each benchmark set, it *i*) generates the Java source files for strong d-hitting families of schedules, *ii*) compiles and runs these files, and *iii*) processes the output files and collects the results reported in the paper.

A.2 Artifact check-list (meta-information)

- **Algorithm:** Algorithm for construction of strong hitting families of schedules is given in Algorithm 1 in the paper.
- **Program:** The source code in the package `lin/scheduleGen` implements the algorithm for constructing strong hitting families of schedules. The source code in the package `lin/testGen` produces the Java test files for checking linearizability.
- **Compilation:** Scala Build Tool (sbt) and Java compiler
- **Binary:** Binary not included.
- **Data set:** The data set of history files are provided in the compressed file `example/histories.zip`.
- **Run-time environment:** Our artifact has been developed and tested on MacOS 10.13.6.
- **Hardware:** We ran our experiments on a machine with a 2.6 GHz Intel Core i7 Processor and 16 GB memory.
- **Output:** The experiments produce three sets of output files:
- **Publicly available?:** Yes.
- **Artifacts publicly available?:** Yes.
- **Artifacts functional?:** Yes.
- **Artifacts reusable?:** Yes.
- **Results validated?:** Yes.

A.3 Description

A.3.1 How delivered

Our artifact is publicly available on GitHub: https://github.com/burcuku/check-lin. It has a Zenodo DOI: https://doi.org/10.5281/zenodo.1890165.

A.3.2 Hardware dependencies

Our experiments produce Java source files which take about 20 GB together with their compiled class files. The artifact requires a machine with enough disk space for the generated files.

A.3.3 Software dependencies

Our software requires Java 1.8, Scala 2.12, Scala Build Tool, and Python 3.6 (with the *statistics* package) installations.

A.3.4 Data sets

We use sets of history files collected from the `ArrayBlockingQueue (ABQ)`, `ConcurrentHashMap (CHM)`, `ConcurrentLinkedDeque (CLD)`, `ConcurrentLinkedQueue (CLQ)`, `ConcurrentSkipListMap (CSLM)`, `ConcurrentSkipListSet (CSLS)`, `LinkedBlockingDeque (LBD)`, `LinkedBlockingQueue (LBQ)`, `LinkedTransferQueue (LTQ)` and `PriorityBlockingQueue (PBQ)` data structures in Java's `java.util.concurrent` package.

The data sets can be extracted from `example/histories.zip`.

A.4 Installation

The project can be cloned from the GitHub repository:

```
$ git clone https://github.com/burcuku/check-lin
```

The contents of the compressed file `example/histories.zip` should be extracted into `example` folder so that `example/histories` directory keeps a folder for the histories of each concurrent data structure.

A.5 Experiment workflow

The experimental results used in the paper can be produced by running the main script:

```
$ cd check-lin
$ python scripts/main.py
```

This script creates the Java source files for linearizability checking, compiles and runs them. It also processes the produced statistics files (which appear in `stats` and `out` folders) and collects the results in the folder `results`.

Note: It takes approximately 1–3 hours to run the script for each concurrent data structure data set, and in total approximately 20 hours for the whole script on a machine with a 2.6 GHz Intel Core i7 Processor and 16 GB memory.

Alternatively, the script can be run separately for each history set, by providing the set of history files to check for:

```
$ python scripts/main.py ABQ
```

The set of history file can be one of `ABQ`, `CHM`, `CLD`, `CLQ`, `CSLM`, `CSLS`, `LBD`, `LBQ`, `LTQ` or `PBQ`.

A.6 Evaluation and expected result

The main script collects the results in the folder `results` which contains three files:

- The file `results/table1.txt` keeps the properties of the processed history files for each data structure which is given in **Table 1** in the paper.
- The file `results/table2.txt` keeps the number of schedules generated for each data structure for increasing d values which is given in **Table 2** in the paper.
- The file `results/table3.txt` keeps the number and percentage of the linearizable history files shown by strong gitting families of schedules for increasing d values which is given in **Table 3** in the paper.

A.7 Experiment customization

The artifact can be used for checking linearizability of any execution history provided that the history information is recorded in a certain format. We use the format generated by the Violat [9] framework. More information for checking linearizability of a single history file is provided in our project's GitHub repository.