# Automatically Proving Linearizability

Viktor Vafeiadis

University of Cambridge

**Abstract**

This paper presents a practical automatic verification procedure for proving linearizability (i.e., atomicity and functional correctness) of concurrent data structure implementations. The procedure employs a novel instrumentation to verify logically pure executions, and is evaluated on a number of standard concurrent stack, queue and set algorithms.

## 1 Introduction

Linearizability [11] is the standard correctness requirement for concurrent implementations of abstract data structures (such as stacks, queues, sets and finite maps) packaged into a concurrent library (such as `java.util.concurrent`). It requires every library operation to be atomic (behave as if it were executed in one indivisible step) and to satisfy a given functional correctness specification.

The most common way to prove linearizability is to identify the so-called *linearization points* of each operation. These are program points where the entire effect of a method call logically takes place. Sadly, however, these linearization points are often rather complicated: they can depend on a non-local boolean condition and can even reside within other concurrently executing threads. This makes a brute force search for the linearization points infeasible.

We observe, however, that in practice such complicated linearization points arise only in operation executions that do not *logically* update the library's shared state. It is therefore possible to search for the linearization points for operations whose specification is always effectful (i.e., modifies the shared state), but we need a different approach to verify operations with a possibly pure specification (i.e., one not modifying the shared state).

This paper presents one such procedure for proving linearizability (see §4). For operations with a possibly pure specification, it instruments the library code with a certain 'pure linearizability checker,' derived from the specification, and runs a suitably powerful abstract interpreter to validate that there are no assertion violations. This effectively considers all possible linearization points in one go and results in a non-constructive linearizability proof. As a result, we have succeeded in verifying several concurrent stack and queue implementations, and have obtained mixed results for set implementations (see §5 for details).

**Related Work.** The related verification work can be classified in three groups.

First, there are various model-checking papers [22, 13, 4]. These do not prove correctness; they merely check short execution traces of a small number of threads. On the positive side,

```
Sequence AQ = @empty;

void enqueue(int e) {
  atomic {
    AQ = @app(AQ, @singl(e));
  }
}
```

```
int tryDequeue(void) { int ARes;
  atomic {
    if (AQ == @empty) return EMPTY;
    else { ARes = @hd(AQ);
           AQ = @tl(AQ);
           return ARes; }
} }
```

Figure 1: Specification of a concurrent queue object.

such tools do not require linearization points to be annotated, are good at quickly finding bugs, and return concrete counterexample traces for failed verifications.

Second, there are static analyses (shape analyses, in particular) [2, 3, 19]. With the exception of [2], these analyses work for an unbounded number of threads and result in a proof of linearizability. Unfortunately, all of these analyses require the programmer to specify the linearization points, a task that is quite difficult when the linearization points are conditional or within the source code of other concurrently executing operations, as we will shortly see.

Finally, there are manual verification efforts. Some (e.g.,[18]) are pencil and paper proofs in a particular program logic, others (e.g.,[5]) do a direct simulation proof in a mechanised proof assistant, while some more recent work [15] does part of the proof in a program logic and another part using operational reasoning on traces.

## 2 A Motivating Example: the M&S Queue

We start with a motivating example for the rest of the paper: the well known Michael & Scott non-blocking queue [14] (henceforth referred to as the M&S queue). Figure 1 contains the specification of the concurrent queue operations written in C-like pseudocode. The state of the queue is represented by the shared variable `AQ`, which holds a sequence of values. We use the following notation for mathematical sequences: `@empty` stands for the empty sequence; `@singl` constructs a sequence consisting of one element; `@app` concatenates two sequences; `@hd` returns the first element of a sequence; and `@tl` returns all but the first element of a sequence.

The queue supports two atomic operations: (1) `enqueue`, which adds an item to the end of the queue, and (2) `tryDequeue`, which removes and returns the first item of the queue if there is one, or returns `EMPTY`, if the queue is empty. Both operations are supposed to be *atomic*, i.e., executing in one step.

Figure 2 contains the M&S queue implementation which is significantly more complicated than the specification above. The queue is represented by two pointers into a null-terminated singly-linked list. The first pointer (`Q->head`) points to the beginning of the list and is updated by `tryDequeue` operations. The second pointer (`Q->tail`) is used to find the end of the list so that `enqueue` can locate the last node of the list. It does not necessarily point to the last node of the list, but it can lag behind. This is because there is no widely available hardware instruction that can change `Q->tail` and append one node onto the list in one atomic step. Consequently, `enqueue` first appends a node onto the list with the underlined `CAS` instruction, and later updates `Q->tail` with its final `CAS` instruction. In addition, whenever a concurrently executing thread notices that the tail pointer is lagging behind the end of the list, it tries to advance it using the `CAS(&Q->tail,tail,next)` instructions.

```
typedef struct Node_s *Node;

struct Node_s {
  int val;
  Node tl;
}

struct Queue {
  Node head, tail;
} *Q;

void enqueue(int value) {
  Node node, next, tail;
  node = new_node();
  node->val = value;
  node->tl = NULL;
  while(true) {
    tail = Q->tail;
    next = tail->tl;
    if (Q->tail != tail) continue;
    if (next == NULL) {
      if (CAS(&tail->tl,next,node))
        break;
    } else {
      CAS(&Q->tail,tail,next);
    }
  }
  CAS(&Q->tail,tail,node);
}
```

```
void init(void) {
  Node node = new_node();
  node->tl = NULL;
  Q = new_queue();
  Q->head = node;
  Q->tail = node;
}

int tryDequeue(void) {
  Node next, head, tail;
  int pval;
  while(true) {
    head = Q->head;
    tail = Q->tail;
    next = head->tl;
    if (Q->head != head) continue;
    if (head == tail) {
      if (next == NULL)
        return EMPTY;
      CAS(&Q->tail,tail,next);
    } else {
      pval = next->val;
      if (CAS(&Q->head,head,next))
        return pval;
    }
  }
}
```

Figure 2: The M&S non–blocking queue implementation.

In the remainder of this paper we shall define what it means for the implementation to satisfy its specification and present a method for proving this.

## 3 Linearizability

We take programs to consist of a sequential initialisation phase followed by a parallel composition of a fixed (but not statically bounded) number of threads, $T$. The state consists of a set of global variables, $G$, and a set of local variables per thread, $L_t$, that includes the thread's program counter, $\texttt{pc}_t$. As a convention, we will subscript thread-local variables with the corresponding thread identifier to distinguish them from the global variables. We model each thread as a transition relation on the valuations of the global and its local variables.

A library, $A$, consists of a constructor, $A_{\text{init}}$, and a number of operations (a.k.a., methods), $A_1, \ldots, A_n$, each expecting a single argument, $\texttt{arg}_t$, and returning their result in the thread-local variable $\texttt{res}_t$. A client of the library is any program that calls the library's constructor once in its initial sequential phase, and then can call any number of the library's methods possibly concurrently with one another. Let $C[A]$ be the transition relation denoting the composition of the client $C$ with the library $A$. We write $C[A]^*$ for its reflexive and transitive closure. In such a composition, we write $G^C$ (resp. $L_t^C$) for the global (resp. local) variables belonging to the client and, analogously, $G^A$ and $L_t^A$ for those belonging to the library. We assume that $G^C$ and $G^A$ are disjoint, and that $L_t^C \cap L_t^A = \{\texttt{arg}_t, \texttt{res}_t, \texttt{pc}_t\}$.

Linearizability [11] is a formalisation of the concept of atomicity. Briefly, it requires that every execution history consisting of calls to `enqueue` and `tryDequeue` is equivalent (up to reordering of events) to a legal, sequential history that preserves the order of non-overlapping methods in the original history. We say that a history is sequential if none of its methods overlap in time; moreover, it is legal if each method satisfies its specification.

In this paper, we prefer a slightly different definition of linearizability given in terms of *instrumented* clients.

**Definition 1.** *An* instrumented *client of a library $A$ is a client annotated with an auxiliary global variable $h$ as follows: (1) At the initial state, let $h = \epsilon$; (2) before every call to $A_i$ by thread $t$, append* ($\textsc{call}\ t, i, arg_t$) *to $h$; and (3) after each return from a call to $A_i$ by thread $t$, append* ($\textsc{ret}\ t, i, res_t$) *to $h$.*

In effect, the auxiliary variable $h$ records the observed execution history. Note that there is a gap in time between when a method returns and when the return is recorded in $h$. This gap allows us to define linearizability as follows:

**Definition 2** (Linearizability). *A library $A$ is* linearizable *with respect to a specification $B$ if and only if for all instrumented clients $C$ and every state $s$, if $(s_{\text{init}}, s) \in C[A]^*$, then there exists a state $s'$ such that $(s'_{\text{init}}, s') \in C[B]^*$ and $s(h) = s'(h)$, where $s_{\text{init}}$ and $s'_{\text{init}}$ are the initial states after calling $A_{\text{init}}$ and $B_{\text{init}}$ respectively.*

This definition is slightly easier to work with than the original one by Herlihy and Wing [11], because it uses syntactic equality on the recorded histories rather than equivalence up to reordering of non-overlapping calls of the actual histories. It is also more general as it corresponds to the original definition only if all of $B$'s methods are atomic. The same generalisation is found in the definition of Filipović et al. [7].

## 3.1 Proving Linearizability Using Linearization Points

The most common way of proving linearizability of a concurrent library is to find the so-called linearization points of each operation and to demonstrate that the chosen points are correct. These are points in the source code of the library which, when executed, are deemed to perform atomically the entire observable effect of the operation, and hence define the order in which the concurrent operations are to be linearized. Within each operation execution, exactly one linearization point must occur, but statically there can be multiple such points along different execution paths of the operation, some of which might not even be inside the operation!

**Linearization Points of the M&S Queue.** The linearization point of `enqueue` is the underlined `CAS` instruction, when it succeeds. Its effect is to link a node to the end of the concrete list, which logically corresponds to appending an item to the queue.

The `tryDequeue` method has two linearization points depending on the result. The linearization point for the empty case is the underlined assignment `next = head->tl`. This is a linearization point only if the same loop iteration later executes `return EMPTY`. The second linearization point is the underlined `CAS` instruction. Its effect is to advance the `Q->head` pointer, which logically removes the first element from the queue.

As presented, these linearization points are conditional: not every time the underlined instructions are executed, they are linearization points. Fortunately, the conditions of the two points involving `CAS` can easily be eliminated by unfolding the definition of `CAS`. For example, if we expand out the definition of the underlined `CAS` of `enqueue`, we get:

```
atomic { if (tail->tl == next) { tail->tl = node; break; } }
```

Thus, it is easy to identify the linearization point of `enqueue` with the assignment to `tail->tl` whenever that assignment is executed. We can do likewise with the second linearization point of `tryDequeue`.

In contrast, the first linearization point of `tryDequeue` is truly conditional. Specifying it formally requires an auxiliary prophecy variable [1] to record whether the program will later execute `return EMPTY` in the same loop iteration. The prophecy variable is needed because when executing the underlined read from `head->tl` we cannot tell whether the test `Q->head != head` on the following line will succeed. Therefore, the full condition is:

$$\neg\texttt{prophecy(Q->head!=head)} \land \texttt{head==tail} \land \texttt{next==NULL}\,.$$

In the next section, we will see a method for proving that `tryDequeue` is linearizable that avoids the conditions on this linearization point and the prophecy variable.

## 4 Automatic Proof Technique

### 4.1 Key Observation

It is clear from the M&S queue that linearization points are often conditional, and that some conditions can be quite involved. Searching for such complex conditions is clearly infeasible. We can, however, observe that

> Operations have complex conditional linearization points
> only in executions that do not *logically* modify the state.

For example, at the first linearization point of `tryDequeue`, the operation does not logically modify the state. That is, if we execute the `tryDequeue` specification at that point, the value of `AQ` will not be affected. It is, however, possible that `tryDequeue` updates the concrete state (e.g., by performing the `CAS(&Q->tail, tail, next)` in a previous loop iteration), but these updates do not affect the logical contents of the queue.

Quite surprisingly, this observation holds for most concurrent algorithms in the literature. To the best of our knowledge, it holds for all but two of the algorithms in Herlihy & Shavit's book [12]. A possible explanation as to why this is so is that logically effectful operation executions are much more difficult to optimise than the ones that only do not logically modify the state. Therefore, they tend to have simpler correctness arguments than the more heavily optimised logically pure executions. Notable exceptions where our observation does not hold are: (*i*) the Herlihy & Wing queue [11], (*ii*) the elimination-based stack of Hendler et al. [10], and (*iii*) RDCSS by Harris et al. [9]. Verifying these algorithms automatically is beyond the scope of this paper.

In the following, we shall distinguish between *pure* and *effectful* executions of the abstract operations, i.e. the operation specifications. We say that an abstract operation execution is pure if it does not modify the abstract state. Otherwise, we say that the execution is effectful.

## 4.2 Dealing with Logically Pure Executions

To verify logically pure executions, we introduce one auxiliary boolean array, $\texttt{can\_return}_{t,op}[]$, per thread and per library operation. Each array is indexed by the set of possible return values. While thread $t$ is executing the operation $op$, then $\texttt{can\_return}_{t,op}$ satisfies the following invariant: whenever an entry, $\texttt{can\_return}_{t,op}[v]$, in the array is true, then there exists an instant since the operation was called at which if the operation's specification had been executed, it would not have modified the global (abstract) state and would have returned $v$. Therefore, if $\texttt{can\_return}_{t,op}[\texttt{res}_t]$ is true when the operation returns, then we know that there existed a valid linearization point during the operation's execution.

To ensure that the aforementioned invariant holds, we initialise every element of `can_return`[] to false at the beginning of the operation. Then, at any later point, we can set `can_return`[$v$] to true provided that executing the operation's specification does not modify the global (abstract) state and returns $v$. This is the task of the 'pure linearizability checker,' which we introduce below.

**Pure Linearizability Checker Construction.** Assuming that the specifications do not contain any loops or any function calls, we rewrite each specification as a non-deterministic choice of a number of execution paths consisting of assignments, assume statements, and terminated by a return command. For uniformity, we change specifications that do not return any value to return 0. For example, the `enqueue` and `tryDequeue` specifications become:

$$\begin{array}{lll} enq & \stackrel{\text{def}}{=} & \texttt{AQ=@app(AQ,@singl(e)); return 0} \\ deq(1) & \stackrel{\text{def}}{=} & \texttt{assume(AQ==@empty); return EMPTY} \\ deq(2) & \stackrel{\text{def}}{=} & \texttt{assume(AQ}\neq\texttt{@empty); ARes=@hd(AQ); AQ=@tl(AQ); return ARes} \end{array}$$

where `tryDequeue` corresponds to the non-deterministic choice among the paths $deq(1)$ and $deq(2)$. We say that one of these paths is *syntactically pure* if and only if it has no assignments to global variables. For example, $deq(1)$ is syntactically pure, but $enq$ and $deq(2)$ are not.

The pure linearizability checker is constructed by replacing `return` $v$ with the assignment `can_return`[$v$]`=true` along every syntactically pure specification path of the method, and by truncating the non-pure paths before their first effectful command (namely, an assignment to a global variable). This construction ensures that pure linearizability checkers set `can_return`[$v$] to true *only if* at the current point the specification does not modify the global state and returns $v$.

Going back to the queue specifications, the pure linearizability checker of `enqueue` is simply the empty command, because $enq$ is not syntactically pure. The pure linearizability checker of `tryDequeue` is

```
if(*) {assume(AQ==@empty); can_return[EMPTY]=true;}
else  {assume(AQ!=@empty); ARes=@hd(AQ);}
```

**Algorithm 1** $\textsc{ProveLinearizable}(op_{\text{init}}, spec_{\text{init}}, op_1, spec_1, \ldots, op_n, spec_n)$

```
 1: $iop_{\text{init}} \leftarrow (op_{\text{init}}; spec_{\text{init}})$
 2: for $i \leftarrow 1$ to $n$ do
 3:     $check_i \leftarrow \textsc{GeneratePureChecker}(spec_i)$
 4: $(\mathcal{C}, op_1, \ldots, op_n) \leftarrow \textsc{GetCandidateLinPoints}(op_1, \ldots, op_n)$
 5: for all $cand \in \mathcal{C}$ do
 6:     for $i \leftarrow 1$ to $n$ do
 7:         $iop_i \leftarrow \textsc{InstrumentLinPoints}(cand, op_i, spec_i)$
 8:     if $\textsc{Verify}(iop_{\text{init}}, iop_1, check_1, \ldots, iop_n, check_n)$ then
 9:         return 'Success'
10: return 'Failure'
```

In this case, as `ARes` is a dead local variable, the assignment can be removed, and the checker can be rewritten as follows:[1]

```
if(AQ==@empty) {can_return[EMPTY]=true;}
```

**Linearization Points in Other Threads.** Note that it is sound to execute the pure checker for a thread, $t$, at any point in time, even between atomic steps of other threads. Hence, we can also handle linearization points that are in the source code of other concurrently executing operations. Basically, when the static analyser verifies one thread with a compositional verification technique, it has a model of what updates all the other threads can do and how these updates affect the current thread. Thus, when symbolically evaluating the operation being verified, after each of its atomic commands, the static analyser also symbolically evaluates the model of what all the other threads can do, before proceeding with the operation's next atomic command. Therefore, if we instrument also the model of all the other threads' behaviour with calls to to the pure linearizability checker, then the static analyser is able to establish linearizability even in cases where some linearization points are within the code of a concurrently executing thread.

**Return Set Abstraction.** To ensure that the static analyser terminates, we often need to abstract the return set (i.e., the set of values $v$ such that `can_return`$[v]$ is true). While this is unnecessary for specifications, such as `tryDequeue`, whose pure executions return only one of a small number of results, it is crucial for specifications, such as `peek` on a stack or a queue, whose pure executions can return an unbounded number of different answers. So, to abstract over this set of variables we apply 'canonical abstraction' [16], which effectively remembers only which program variables and program constants are contained in the set. As there is only a finite set of variables and constants appearing in the input program, the range of this abstraction is finite, and hence the termination of the underlying static analysis is not affected.

## 4.3 Verification Procedure Outline

Algorithm 1 contains our procedure for proving linearizability. $\textsc{ProveLinearizable}$ takes as arguments the library's constructor ($op_{\text{init}}$) with its specification ($spec_{\text{init}}$), and the set of

[1]This simplification is for presentation purposes only. The implementation does not perform such simplifications.

library operations ($op_1, \ldots, op_n$) with their specifications ($spec_1, \ldots, spec_n$). The specifications are just normal methods that operate on the logical state, which is disjoint from the concrete state.

The algorithm consists of two phases. First, it instruments the constructor of the library, computes the pure checkers for each operation and generates a set of candidate linearization point assignments, $\mathcal{C}$. Then, it iterates over that set checking whether any of these assignments is valid. If a valid assignment is found, the procedure returns 'Success' indicating that linearizability has been proved; otherwise, it returns 'Failure.'

**Preparation Phase.** First, the algorithm instruments the library's constructor: $iop_{\text{init}}$ is simply the sequential composition of the constructor, $op_{\text{init}}$, and its specification, $spec_{\text{spec}}$. Next, pure checkers are generated, as described in §4.2.

Then, GetCandidateLinPoints is called. This, first, unfolds the definitions of `CAS` and `DCAS` in the various operations. This syntactic transformation exposes the trivial conditions governing the linearization points of effectful operations, so that the transformed operation has unconditional linearization points. For uniformity, it arranges that methods and specifications that do not return any results, return 0 instead.

Then, along each execution path of each operation, it chooses one command writing to the shared state as the effectful linearization point. If the operation's specification has pure executions (e.g., `tryDequeue`), it also can choose no linearization point on some of its execution paths in the hope that the execution path corresponds to pure execution of its specification. Obviously, memory writes appearing within loops are discarded since they can be executed multiple times. This process produces one linearization point assignment: a set of program points that are to be treated as (unconditional) linearization points of the method they belong to. GetCandidateLinPoints returns the set, $\mathcal{C}$, of all possible linearization point assignments.

**Checking Phase.** Each operation $op_i$ is instrumented with its specification $spec_i$ by adding the two new auxiliary local variables:

- `lres`, holding the result of the abstract method call at the effectful linearization point if this has occurred, or the reserved value `UNDEF` otherwise,
- `can_return`, an array storing the allowed return values of any pure linearization points that have been executed so far,

and the following code:

- At the beginning of the method, InstrumentLinPoints sets `lres` to `UNDEF`. and all the elements of `can_return`[] to false.
- At the chosen candidate linearization points in *cand* that are in the source code of $op_i$, it inserts an assertion checking that the linearization point has not occurred followed by a call to the abstract method:

  `assert(lres==UNDEF); lres=`$spec_i$`(args)`

  where `args` are the arguments of $op_i$ (which we assume are not modified by $op_i$). The assertion about `lres` and the subsequent assignment ensure that the candidate linearization point is executed at most once along every execution path.

- Finally, at the method's return point(s), it inserts the following check:

  $$\texttt{assert(lres==res} \vee \texttt{(lres==UNDEF} \wedge \texttt{can\_return}[\texttt{res}]\texttt{))}$$

  where `res` is the variable storing the concrete method's return value. This check ensures that either an effectful linearization point has occurred and that the method returned the same result as its specification, or that no effectful linearization point has occurred, but there has been a pure linearization point whose return value matches the concrete return value.

The instrumented operations are validated by calling Verify. Verify takes as arguments the library's instrumented constructor ($iop_{\text{init}}$), its instrumented operations ($iop_1, \dots, iop_n$), and one command per operation that is to be inserted at each point during the execution of that operation. These are the just the previously computed pure linearization checkers: $spec_1, \dots, spec_n$. Note that these checkers have to be passed as arguments to Verify (and cannot simply be instrumented in the source code of the operations), because we want to allow linearization points of pure executions to reside in the code of other threads. To handle this case, Verify also inserts the checkers in its abstractions of the other threads' behaviour. This instrumentation cannot be done statically before calling Verify because these abstractions have not yet been computed.

Verify constructs the most general client of the library and uses an automatic static analysis to prove that the library is memory safe and that the assertions in any `assert` statements in the library are always satisfied. The most general client is a top-level program which models all possible usages of the library. It consists of the initialization routine followed by an unbounded parallel composition of threads, each of which non-deterministically executes one of public methods of the library in a loop. So, if so assertion violations occur for the most general client of the library, then no library assertion violations will occur for *any* client of the library.

## 4.4 Soundness

To prove soundness of our algorithm, we first show that the instrumentation described in §4.2 and §4.3 implies linearizability:

**Theorem 1** (Instrumentation Correctness)**.** *If a library A is instrumented as described in §4.2 and §4.3 with respect to the specification B, and an execution of a client of the instrumented library did not violate any of assertions, then that execution was linearizable.*

The proof of this theorem is quite technical and can be found in the associated technical report [21]. Briefly, for each operation, we can pick the instant when `lres` was set as its linearization point, or if `lres` was never set, then the point when $\texttt{can\_return}[r]$ was first set to true, where $r$ is the eventual return value of the operation.

The soundness of ProveLinearizable follows directly from Theorem 1 and the specification of Verify, which checks absence of library assertion violations for any execution for any valid client of the library.

**Theorem 2** (Soundness)**.** *If* ProveLinearizable($init, init\_spec, op_1, spec_1, \dots, op_n, spec_n$) *returns 'Success,' then the library consisting of the constructor init and methods $op_1, \dots, op_n$ is linearizable with respect to its specification ($init\_spec$, $spec_1$, $\dots$, $spec_n$).*

## 4.5 Implementation

We have implemented the algorithm for proving linearizability within Cave, an automatic verification tool for concurrent algorithms based on RGSep. We take Verify to be the RGSep action inference algorithm [20], adapted to execute the corresponding pure checker, $check_i$, symbolically at every step of the 'stabilisation' calculations within each instrumented operation, $iop_i$. Thus, in effect, Verify simulates the pure checker after every atomic command of the current thread accessing the shared state and also after every atomic command of other concurrently executing threads.

**Implementation Optimisations.** Before executing Alg. 1, Cave first calls Verify with arguments ($init$, $op_1$, `skip`, …, $op_n$, `skip`) to check that the uninstrumented library is memory safe: that it does not dereference any invalid pointers and that it does not violate any assertions. The purpose of this initial call is threefold:

- First, it aids debugging. If action inference cannot verify that the uninstrumented program is safe (either because the program is erroneous, or because the analysis is imprecise), there is no way that it will succeed in verifying the instrumented programs. Thus, it is better to fail quickly, and give a simpler error message to the user.
- Second, it can help quickly prune the search space of linearization point assignments. Action inference distinguishes updates to shared memory locations from updates to thread-local data, as only the former have an action associated with them. Thus, we can ignore any candidate linearization point assignments that involve thread-local accesses.
- Third, the set of RGSep actions inferred by this phase can then used as a starting point for the following Verify calls within Alg. 1, thereby making later action inference calls reach their fix-point in a single iteration.

We can further optimise the call to Verify in Alg. 1 in two ways. First, it can fail immediately if the correlation between the abstract state and the concrete state is lost. This allows us to fail much more quickly on erroneous linearization point assignments. Second, it first tries to prove linearizability by inlining the instrumented checkers only within the source code of the current thread (i.e., only at the beginning of every stabilisation), and if that fails to establish linearizability, then also after every stabilisation iteration. This alleviates the cost of inserting the pure checkers within the abstraction of other threads, when this is not needed to prove linearizability.

## 5 Experimental Evaluation

We have successfully applied Cave to a number of practical concurrent stack, queue, and set algorithms from the literature, which are reported in Fig. 3. For some algorithms, we have considered two versions: one being just core algorithm as normally published, and one being a mostly straightforward extension of the algorithm providing supplementary operations. We present our results for both versions in the same line separating the corresponding numbers with a slash. For each algorithm, we record the number of lines of code excluding comments, blank lines, and the specifications (**Lines**), the number of public methods of the library (**Ops**), the

| **Data structure** | **Lines** | **Ops** | **Eff** | **Pure** | **LpO** | **Time(s)** |
|---|---|---|---|---|---|---|
| DCAS stack | 52/93 | 2/7 | 2/4 | 1/4 | 0 | 0.1/0.2 |
| Treiber stack [17] | 52/93 | 2/7 | 2/4 | 1/4 | 0 | 0.1/0.2 |
| M&S two-lock queue [14] | 54/85 | 2/4 | 2/3 | 1/2 | 0 | 2.0/16.5 |
| M&S non-block. queue [14] | 82/127 | 2/4 | 2/3 | 1/2 | 0 | 1.7/4.9 |
| DGLM non-block. queue [5] | 82/126 | 2/4 | 2/3 | 1/2 | 0 | 1.8/7.6 |
| Pessimistic set [12] | 100 | 3 | 2 | 3 | 0 | 392.9 |
| V&Y DCAS-based set [22] | 101 | 3 | 2 | 3 | 0 | 51.0 |
| ORVYY lazy set [15] | 94 | 3 | 2 | 3 | 1 | 521.5 |

Figure 3: Verification statistics for a collection of stack, queue, and set benchmarks.

number of effectful methods (**Eff**), the number of methods with pure executions (**Pure**), the number of methods with linearization points in other threads (**LpO**) and the total verification time in seconds (**Time**).

**Stack & Queue Benchmarks.** The DCAS and Treiber stack algorithms use non-blocking synchronisation, respectively performing a DCAS or a CAS to update the top of the stack. The basic versions of the stack algorithm provide just `push` and `tryPop` operations. The `tryPop` operation has a pure execution in case the stack was empty, in which case it returns a special value (similar to the `tryDequeue` of Fig. 1). The extended implementations also provide a variant of `pop` which blocks if the stack is empty, a `peek` operation, a blocking operation waiting for the stack to become empty, and for testing for emptiness and clearing the stack.

The queue algorithms support `enqueue` and `tryDequeue` operations with the specifications shown in Fig. 1. The extended versions have two further operations: a blocking dequeue and an emptiness test. The first algorithm is a lock-based design due to Michael and Scott that uses a different lock to protect each end of the queue. The second one is due to the same authors and was presented in Fig. 2. The DGLM queue is a variant of M&S non-blocking queue that was proposed by Doherty et al. [5] and verified in the PVS theorem prover.

**Set Benchmarks.** These have three operations: adding an element to the set, removing one element from the set, and testing for membership in the set. The first two operations are effectful, but have pure executions whenever the item to be added (resp. removed) was already in the set (resp. not in the set). In all cases, the set is represented as a sorted singly linked list with two sentinel nodes.

The pessimistic set has a lock per list node, acquired in a hand-over-hand fashion. The V&Y DCAS-set [22] traverses the list optimistically (i.e., with no synchronisation) and then validates that the traversal was correct. The ORVYY lazy set [15] also performs optimistic traversals and uses a bit for marking nodes that are about to be deleted. This allows it to have an efficient wait-free `contains` implementation. The ORVYY lazy set is particularly interesting, because one of the linearization points of `contains` lies within code of a different thread.

We have also run CAVE on two further set algorithms: the V&Y CAS-based set [22] and the HHLMSS lazy set [12, §9.7], but it failed to prove linearizability. Verification of the first example failed because one of the calls to VERIFY timed out, probably due to the current naïve axiomatisation of sorted sequences in the analyser. In the second algorithm, the correct abstraction map lies outside of the abstract domain of our implementation of Verify and, hence, was not be found.[2]

**Discussion.** As it can be seen from the execution times, the stack algorithms are rather easy to verify. This is mainly because these algorithms have rather simple data structure invariants (e.g. the stack is represented by a null-terminated singly-linked list), which can be found easily by the underlying shape analysis. In contrast, the queue algorithms and especially the set algorithms have much more complicated data structure invariants (e.g. the set being represented by a sorted list with special sentinel nodes and there can be multiple arbitrarily long chains of deleted nodes pointing into the sorted list), which take significantly more effort to prove. In all these algorithms, the search space for the effectful candidate linearization point assignments was quite small and did not increase significantly by adding a few more operations.

Since our tool relies on abstract interpretation, our verification procedure is incomplete: it is unable to verify many correct programs that lie outside its domain (such as the aforementioned HHLMSS lazy set), and does not provide concrete counterexample traces when the verification fails. Moreover, Cave cannot prove linearizability of effectful executions whose linearization points are inside the code of different threads (such as RDCSS and the elimination-based stack), unless these linearization points are somehow annotated by the programmer. It can, however, prove linearizability of method executions having linearization points within different threads, provided that these executions are logically effect-free, as was the case with the ORVYY lazy set.

The main observation of this paper that enabled these verification results was to distinguish executions of the abstract operations (i.e., the specifications) that are pure from those that are effectful. This is related to Elmas et al. [6], who in the context of runtime refinement-violation detection treat operations with a pure specification differently than ordinary operations. Flanagan et al. [8] also had a somewhat related concept of purity, but in their work there is no notion of an abstract operation, and purity is applied only to the implementation. None of the algorithms verified here could have been verified with brute-force search for linearization points.

## 6 Conclusion

This paper presented a practical technique for automatically proving linearizability. This was implemented in a tool, Cave, which expects a library to be verified together with its atomic functional correctness specification and attempts to prove that the library is linearizable with respect to its specification. We have applied our tool to a number of concurrent stack, queue, and set algorithms, some of which were mechanically verified for the first time.

As this is the first automatic technique for verifying functional correctness of non-trivial concurrent programs, there are several ways in which it can be improved. One such way would be to deal with effectful linearization points in other threads that are 'similar' to a linearization point in the thread being verified (where two program statements are deemed 'similar' if they are abstracted by the same RGSep action). More practically, our prover should be combined with

[2]The abstraction map for the HHLMSS lazy set is the set of the values of unmarked nodes that are reachable from the head of the list. In contrast, the ORVYY lazy set has a simpler abstraction map: it is the set of the values of all the nodes that are reachable from the head of the list. While it is plausible to extend the analyser to infer such complicated abstraction maps automatically, it is probably better to leave them as input by the programmer.

lightweight methods for proving atomicity (e.g., [8]) and with testing techniques for eliminating incorrect linearization point assignments quickly. Further, as such provers become increasingly sophisticated, it will be important to generate proof objects that can be independently checked by a trusted computer program. Last, but not least, there is a never-ending challenge in devising more powerful and more efficient abstract domains for the underlying static analyses used in procedures such as Verify. In particular, improving the support for arrays would enable us to reason about several more concurrent algorithms, such as concurrent hashtables.

**Acknowledgements.** I would like to thank the anonymous reviewers for their useful feedback that helped improve the paper. This work was supported by EPSRC grant EP/F036345.

## References

[1] Abadi, M., Lamport, L.: The existence of refinement mappings. Theoretical Computer Science, 82(2):253–284 (1991)

[2] Amit, D., Rinetzky, N., Reps, T., Sagiv, M., Yahav, E.: Comparison under abstraction for verifying linearisability. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590. Springer, Heidelberg (2007)

[3] Berdine, J., Lev-Ami, T., Manevich, R., Ramalingam, G., Sagiv, S.: Thread quantification for concurrent shape analysis. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 399–413. Springer, Heidelberg (2008)

[4] Burckhardt, S., Dern, C., Tan, R., Musuvathi, M.: Line-up: a complete and automatic linearizability checker. In: PLDI 2010. ACM, New York (2010)

[5] Doherty, S., Groves, L., Luchangco, V., Moir, M.: Formal verification of a practical lock-free queue algorithm. In: de Frutos-Escrig, D., Núñez, M. (eds.) FORTE 2004. LNCS, vol. 3235, pp. 97–114. Springer, Heidelberg (2004)

[6] Elmas, T., Tasiran, S., Qadeer, S.: VYRD: verifying concurrent programs by runtime refinement-violation detection. In: PLDI, pp. 27–37. ACM, New York (2005)

[7] Filipović, I., O'Hearn, P.W., Rinetzky, N., Yang, H.: Abstraction for concurrent objects. In: Castagna, G. (ed.) ESOP 2009. LNCS, vol. 5502, pp. 252–266. Springer, Heidelberg (2009)

[8] Flanagan, C., Freund, S.N., Qadeer, S.: Exploiting purity for atomicity. IEEE Trans. Software Eng., 31(4):275–291 (2005)

[9] Harris, T., Fraser, K., Pratt, I.A.: A practical multi-word compare-and-swap operation. In: *16th International Symposium on Distributed Computing*, pp. 265–279 (2002)

[10] Hendler, D., Shavit, N., Yerushalmi, L.: A scalable lock-free stack algorithm. In: SPAA 2004, pp. 206–215. ACM, New York (2004)

[11] Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM TOPLAS, 12(3):463–492. ACM, New York (1990)

[12] Herlihy, M.P., Shavit, N.: The art of multiprocessor programming. Morgan Kaufmann (2008)

[13] Liu, Y., Chen, W., Liu, Y.A., Sun, J.: Model checking linearizability via refinement. In: Cavalcanti, A., Dams, D. (eds.) FM 2009, LNCS, vol. 5850, pp. 321–337. Springer, Heidelberg (2009)

[14] Michael, M., Scott, M.: Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In: PODC 1996. ACM, New York (1996)

[15] O'Hearn, P.W., Rinetzky, N., Vechev, M.T., Yahav, E., Yorsh, G.: Verifying linearizability with hindsight. In: PODC 2010. ACM, New York (2010)

[16] Sagiv, S., Reps, T.W., Wilhelm, R.: Parametric shape analysis via 3-valued logic. In: POPL 1999, pp. 105–118. ACM, New York (1999)

[17] Treiber, R.K.: Systems programming: Coping with parallelism. Tech. report RJ5118, IBM Almaden Res. Ctr. (1986)

[18] Vafeiadis, V.: Modular fine-grained concurrency verification. PhD thesis. Tech. report UCAML-CL-TR-726, Univ. of Cambridge Computer Laboratory (2007)

[19] Vafeiadis, V.: Shape-value abstraction for verifying linearizability. In: Jones, N.D., Müller-Olm, M. (eds.) VMCAI 2009. LNCS, vol. 5403, pp. 335–348. Springer, Heidelberg (2009)

[20] Vafeiadis, V.: RGSep action inference. In: Barthe, G., Hermenegildo, M. (eds.) VMCAI 2010. LNCS, vol. 5944, pp. 345–361. Springer, Heidelberg (2010)

[21] Vafeiadis, V.: Automatically proving linearizability (long version). Tech. report UCAML-CL-TR-778, Univ. of Cambridge Computer Laboratory (2010)

[22] Vechev, M.T., Yahav, E.: Deriving linearizable fine-grained concurrent objects. In: Gupta, R., Amarasinghe, S.P. (eds.) PLDI, pp. 125–135. ACM, New York (2008)

## A Soundness

We say that an *annotated* library is one where we have added the auxiliary code dealing with `lres`, `can_return`, etc. as described in Sec. 4. Note that a library is annotated always with respect to a given specification. A *verified* library is one in which, in every possible execution of the library, no assertion violations occur in the library's source code. Our main soundness theorem then follows from the following theorem:

**Theorem 3** (Soundness)**.** *Every verified annotated library $A$ is linearizable with respect to its specification.*

To prove this theorem, we use some auxiliary definitions and lemmas. We use $s$ to range over states (functions from global and thread-local variables to their values) and $\tau$ to range over traces (sequences of states). We write $|\tau|$ to denote the length of the trace $\tau$ and $\tau_n$ to denote the $n^{\text{th}}$ state of the trace (where $0 \leq n < |\tau|$).

We define $\mathsf{Traces}(R)$ to be the set of traces generated from the initial state, $s_{\text{init}}$, and the transition relation, $R$. Formally,

$$\mathsf{Traces}(R) \stackrel{\text{def}}{=} \{\tau \mid \tau_0 = s_{\text{init}} \land (\forall i.\ 0 \leq i < |\tau| - 1 \Rightarrow (\tau_i, \tau_{i+1}) \in R)\}$$

We write $R^*$ for the reflexive and transitive closure of $R$, and $C[A]$ for the relation on states acting as the combination of a client $C$ with the library $A$.

For notational simplicity, we assume that the internal state of $A$'s specification is a single variable `abs_state`. Further, we assume that we have the following functions for distinguishing transitions of the client from those of the library:

- $\textsc{InA}_i(s)$ is true if and only if the program counter, $s(\texttt{pc}_i)$, is in the code of a call to one of $A$'s methods.
- $\textsc{Beg}_i(s)$ returns the program counter of the beginning of the call to $A_{\text{op}}$ that $\texttt{pc}_i$ is currently in (and is undefined if $\neg\textsc{InA}_i(s)$).
- $\textsc{End}_i(s)$ returns the program counter of the end of the call to $A_{\text{op}}$ that $\texttt{pc}_i$ is currently in (and is undefined if $\neg\textsc{InA}_i(s)$).

We define the function $\mathit{Res}_i(\tau, n)$ to check whether thread $i$ has an outstanding call at $\tau_n$ to one of $A$'s methods which will return at some later point $m$ without having executed an effectful linearization point ($\tau_m(\texttt{lres}_i) = \texttt{UNDEF}$), and if so, to return the call's returned result:

$$\mathit{Res}_i(\tau, n) \stackrel{\text{def}}{=} \begin{cases} \tau_m(\texttt{res}_i) & \text{if } n < m < |\tau| \land \tau_m(\texttt{lres}_i) = \texttt{UNDEF} \land \neg\textsc{InA}_i(\tau_m) \\ & \quad \land (\forall k.\ n \leq k < m \Rightarrow \textsc{InA}(\tau_k)) \\ \text{undefined} & \text{otherwise} \end{cases}$$

Note that this function is well-defined. It searches for the first $m > n$ such that $\textsc{InA}(\tau_m)$ is false, and returns the value of $\texttt{res}_i$ at the state $\tau_m$ provided that $\tau_m(\texttt{lres}_i) = \texttt{UNDEF}$. One can think of $\mathit{Res}_i(\tau, n)$ as a 'prophecy variable' at state $\tau_n$ holding the future return value of thread $i$'s outstanding library invocation.

Next, we define the predicate $\mathit{CR}_i(\tau, n, t)$ to hold whenever `can_return` is set for $i$'s eventual return value, $\mathit{Res}_i(\tau, n)$, in the current state ($\tau_n$) if $i < t$, or in the previous state if $i \geq t$:

$$\mathit{CR}_i(\tau, n, t) \stackrel{\text{def}}{=} \mathsf{defined}(\mathit{Res}_i(\tau, n)) \land \left( \begin{array}{rl} & i < t \land \tau_n(\texttt{can\_return}_i[\mathit{Res}_i(\tau, n)]) \\ \lor & i \geq t \land \tau_{n-1}(\texttt{can\_return}_i[\mathit{Res}_i(\tau, n)]) \end{array} \right)$$

In essense, $t$ counts the number of pure linearization checkers executed at the current state, $\tau_n$. As constructed, threads from 0 to $t - 1$ have executed the pure checker, and hence $\mathit{CR}_i$ looks at the value of $\texttt{can\_return}_i[\mathit{Res}_i(\tau, n)]$ in the current state, $\tau_n$. In contrast, threads from $t$ to $T - 1$ have not yet executed the pure checker, and hence $\mathit{CR}_i$ evaluates $\texttt{can\_return}_i[\mathit{Res}_i(\tau, n)]$ in the previous state, $\tau_{n-1}$.

Further, we define the function $f$ which maps an instrumented concrete trace $\tau \in \mathsf{Traces}(C[A])$, an index $0 \leq n < |\tau|$, and a thread number $t$ to an abstract state. (Again, $t$ represents the number of pure linearization checkers executed at the current state.)

$$f(\tau,n,t) \overset{\text{def}}{=} \lambda v. \begin{cases} \textsc{Beg}_i(\tau_n) & \text{if } v = \texttt{pc}_i \wedge \textsc{InA}_i(\tau_n) \wedge \tau_n(\texttt{lres}_i) = \texttt{UNDEF} \wedge \neg \mathit{CR}_i(\tau,n,t) \\ \textsc{End}_i(\tau_n) & \text{if } v = \texttt{pc}_i \wedge \textsc{InA}_i(\tau_n) \wedge (\tau_n(\texttt{lres}_i) \neq \texttt{UNDEF} \vee \mathit{CR}_i(\tau,n,t)) \\ \tau_n(\texttt{lres}_i) & \text{if } v = \texttt{res}_i \wedge \textsc{InA}_i(\tau_n) \wedge \tau_n(\texttt{lres}_i) \neq \texttt{UNDEF} \\ \mathit{Res}_i(\tau,n) & \text{if } v = \texttt{res}_i \wedge \textsc{InA}_i(\tau_n) \wedge \tau_n(\texttt{lres}_i) = \texttt{UNDEF} \wedge \mathit{CR}_i(\tau,n,t) \\ \text{undefined} & \text{if } v = \texttt{res}_i \wedge \textsc{InA}_i(\tau_n) \wedge \tau_n(\texttt{lres}_i) = \texttt{UNDEF} \wedge \neg \mathit{CR}_i(\tau,n,t) \\ \tau_n(v) & \text{otherwise} \end{cases}$$

For each thread $i$, if $\texttt{pc}_i$ is within a library call, $f$ maps the program counter $\texttt{pc}_i$ to the beginning or the end of the call depending on whether a linearization point has occurred. Similarly, $f$ maps the result register, $\texttt{res}_i$, to the value of $\texttt{lres}_i$ after an effectful linearization point, or to $\mathit{Res}_i(\tau,n)$ after a pure linearization point. For all other variables $v$, $f(\tau,n,t)(v) = \tau_n(v)$.

Within a library call, it maps the program counters to either the beginning or end of the call depending on whether the auxiliary state records that a valid linearization point has occurred or not. In the former case, it also maps the `res` variable to the recorded abstract result. Finally, it maps all other variables (as well as $\texttt{pc}_i$ and $\texttt{res}_i$ when $\neg\textsc{InA}_i(\tau_n)$) as in $\tau_n$. This includes the client's variables, $\texttt{arg}_i$, all the auxiliary variables (such as `abs_state`, $\texttt{lres}_i$, $\texttt{can\_return}_i$ and `h`, but not any local variables of the library $A$. The function, $f$, has the following properties:

**Lemma 4.** *For all instrumented clients $C$ of a verified, annotated library $A$ with specification $B$, all traces $\tau \in \mathsf{Traces}(C[A])$ and all $n < |\tau|$, $(f(\tau,n,T), f(\tau,n+1,0)) \in C[B]^*$.*

*Proof.* Let $t$ be the thread that did the transition from $\tau_n$ to $\tau_{n+1}$. The local variables of all other threads are unchanged by the transition: $\forall i \neq t.\ \forall v.\ \tau_{n+1}(v_i) = \tau_n(v_i)$. We do a case split on what $t$'s transition can be:

- Case $\textsc{InA}_t(\tau_n) \wedge \textsc{InA}_t(\tau_{n+1})$:

  (a) If $\tau_{n+1}(\texttt{lres}_t) = \tau_n(\texttt{lres}_t)$, then $f(\tau,n+1,0) = f(\tau,n,T)$ and so the transition is included in $C[B]^*$.

  (b) If $\tau_n(\texttt{lres}_t) = \texttt{UNDEF} \wedge \tau_{n+1}(\texttt{lres}_t) \neq \texttt{UNDEF}$, then the program has assigned a value to the auxiliary variable $\texttt{lres}_t$. By inspection, whenever this happens, the abstract operation is also called updating $\tau_n(\texttt{abs\_state})$ to $\tau_{n+1}(\texttt{abs\_state})$ and $\tau_{n+1}(\texttt{lres})$ contains the result of the call.

  Hence, noting that $\forall i.\ \mathit{CR}_i(\tau,n+1,0) = \mathit{CR}_i(\tau,n,T)$, there is a $C[B]$ transition from $f(\tau,n,T)$ to $f(\tau,n+1,0)$.

  (c) The third possibility (namely, $\tau_n(\texttt{lres}_t) \neq \texttt{UNDEF} \wedge \tau_{n+1}(\texttt{lres}_t) = \texttt{UNDEF}$) cannot arise as `UNDEF` cannot be assigned to $\texttt{lres}_t$ other than at the beginning of an operation.

- Case $\textsc{InA}_t(\tau_n) \wedge \neg\textsc{InA}_t(\tau_{n+1})$:

  This corresponds to returning from an operation. Hence for every variable $v \neq \texttt{pc}_t$, $\tau_{n+1}(v) = \tau_n(v)$. From the assertion check at the operation's return node, we know that:

  $$\tau_n(\texttt{lres}_t) = \tau_n(\texttt{res}_t) \vee \tau_n(\texttt{lres}_t) = \texttt{UNDEF} \wedge \tau_n(\texttt{can\_return}_t[\tau_n(\texttt{res}_t)])$$

  In both cases, it follows that $f(\tau,n,T)(\texttt{res}_t) = \tau_n(\texttt{res}_t)$ and $f(\tau,n,T)(\texttt{pc}_t) = \textsc{End}_t(\tau_n) = \tau_n(\texttt{pc}_t)$ (note that in the first case we know that $\tau_n(\texttt{res}_t) \neq \texttt{UNDEF}$). Moreover, we have

$f(\tau, n+1, 0)(\texttt{res}_t) = \tau_{n+1}(\texttt{res}_t) = \tau_n(\texttt{res}_t)$. Hence, $\forall v \neq \texttt{pc}_t$, $f(\tau, n+1, 0)(v) = f(\tau, n, T)(v)$, and, thus, the transition is included in $C[B]$.

- Case $\neg\textsc{InA}_t(\tau_n) \wedge \textsc{InA}_t(\tau_{n+1})$:

  This corresponds to call into an operation of $A$. By the instrumentation at the beginning of the library's operations, we have:

  $$\tau_{n+1}(\texttt{lres}_t) = \texttt{UNDEF} \wedge (\forall v.\ \tau_{n+1}(\texttt{can\_res}_t[v]))$$

  Hence, $f(\tau, n+1, 0)(\texttt{pc}_t) = \textsc{Beg}_t(\tau_{n+1}) = \tau_{n+1}(\texttt{pc}_t)$, $f(\tau, n+1, 0)(\texttt{res}_t)$ is undefined, and $\forall v \notin \{\texttt{pc}_t, \texttt{res}_t\}$, $f(\tau, n+1, 0)(v) = f(\tau, n, T)(v)$.
  Therefore, the transition $(f(\tau, n, T), f(\tau, n+1, 0))$ is included in $C[B]$.

- Case $\neg\textsc{InA}_t(\tau_n) \wedge \neg\textsc{InA}_t(\tau_{n+1})$:

  This corresponds to a context transition, and the pair $(f(\tau, n, T), f(\tau, n+1, 0))$ is included in the respective $C[B]$ transition. □

**Lemma 5.** *For all instrumented clients $C$ of a verified, annotated library $A$ with specification $B$, all $\tau \in \mathsf{Traces}(C[A])$, all $n$ such that $0 \leq n < |\tau|$, and all $i$ such that $0 \leq i < T$, we have $(f(\tau, m, i), f(\tau, m, i+1)) \in C[B]^*$.*

*Proof.* If $\neg\textsc{InA}_i(\tau_n)$ then $f(\tau, n, i+1) = f(\tau, n, i)$ and the conclusion holds trivially. If $n = 0$, from the definition of $\mathsf{Traces}$, $\neg\textsc{InA}_i(\tau_n)$, and hence the conclusion is trivial. Therefore, we are left with the case where $n > 0$ and $\textsc{InA}_i(\tau_n)$.

Let $r = \mathit{Res}_i(\tau, n)$. If $r$ is undefined, then $f(\tau, n, i+1) = f(\tau, n, i)$ and the conclusion holds trivially. So, we assume that $r$ is defined. There are two cases to consider:

- Case $\neg\tau_{n-1}(\texttt{can\_return}_i[r]) \wedge \tau_n(\texttt{can\_return}_i[r])$: The only difference between $f(\tau, n, i)$ and $f(\tau, n, i+1)$ is that $f(\tau, n, i+1)(\texttt{pc}_i) = \textsc{End}_i(\tau_n)$ and $f(\tau, n, i+1)(\texttt{res}_i) = r$ whereas $f(\tau, n, i)(\texttt{pc}_i) = \textsc{Beg}_i(\tau_n)$ and $f(\tau, n, i)(\texttt{res}_i) = \mathit{undef}$.

  For the auxiliary assignment $\texttt{can\_return}_i[r] := \mathit{true}$ to have occurred, we must have $\mathsf{PureCond}(\mathit{spec}, r)$ be true, which in turn means that $C[B]$ can do a transition from $f(\tau, n, i)$ to $f(\tau, n, i+1)$.

- Case $\tau_{n-1}(\texttt{can\_return}_i[r]) \vee \neg\tau_n(\texttt{can\_return}_i[r])$:

  Then, $f(\tau, n, i+1) = f(\tau, n, i)$, and the conclusion holds trivially. □

From these two lemmas, by an inductive argument, we conclude that:

**Corollary 6.** *For all $\tau \in \mathsf{Traces}(C[A])$, all $m, n < |\tau|$ and all $i, j \leq T$, if $(m, i) \leq_{\text{lex}} (n, j)$, then $(f(\tau, m, i), f(\tau, n, j)) \in C[B]^*$.*

Finally, we can proceed with the proof of the soundness theorem:

*Proof of Theorem 3.* From Definition 2, we have to prove that for every instrumented client $C$, for every state $s$ such that $(s_{\text{init}}, s) \in C[A]^*$, there exists a state $s'$ such that $(s'_{\text{init}}, s') \in C[B]^*$ and $s'(h) = s(h)$ (where $s_{\text{init}}$ (resp. $s'_{\text{init}}$) is the initial states after the constructor of $A$ (resp. $B$) has been called).

Fix $C$ and $s$. Since $(s_{\text{init}}, s) \in C[A]^*$, there is a trace $\tau$ such that $\tau_0 = s_{\text{init}}$, $\tau_{|\tau|} = s$, and $(\tau_i, \tau_{i+1}) \in C[A]$ for all $i < |\tau|$. Pick $s' = f(\tau, |\tau|, T)$ as the witness to the existential. From Corollary 6, we know that $(f(\tau, 0, T), f(\tau, |\tau|, T)) \in C[B]^*$. From the definition of $f$, we get $f(\tau, 0, T) = s'_{\text{init}}$ and $f(\tau, |\tau|, T)(h) = \tau_{|\tau|}(h) = s(h)$, as required. □