Proving linearizability using forward simulations*

Ahmed Bouajjani[1], Michael Emmi[2], Constantin Enea[1], and Suha Orhun Mutluergil[3]

[1] IRIF, Univ. Paris Diderot, {abou,cenea}@irif.fr
[2] Nokia Bell Labs, michael.emmi@nokia.com
[3] Koc University, smutluergil@ku.edu.tr

Abstract. Linearizability is the standard correctness criterion for concurrent data structures such as stacks and queues. It allows to establish observational refinement between a concurrent implementation and an atomic reference implementation. Proving linearizability requires identifying linearization points for each method invocation along all possible computations, leading to valid sequential executions, or alternatively, establishing forward *and* backward simulations. In both cases, carrying out proofs is hard and complex in general. In particular, backward reasoning is difficult in the context of programs with data structures, and strategies for identifying statically linearization points cannot be defined for all existing implementations. In this paper, we show that, contrary to common belief, many such complex implementations, including, e.g., the Herlihy&Wing Queue and the Time-Stamped Stack, can be proved correct using only forward simulation arguments. This leads to simple and natural correctness proofs for these implementations that are amenable to automation.

1 Introduction

Programming efficient concurrent implementations of atomic collections, e.g., stacks and queues, is error prone. To minimize synchronization overhead between concurrent method invocations, implementors avoid blocking operations like lock acquisition, allowing methods to execute concurrently. However, concurrency risks unintended interoperation interference, and risks conformance to atomic reference implementations. Conformance is formally captured by *(observational) refinement*, which assures that all behaviors of programs using these efficient implementations would also be possible were the atomic reference implementations used instead.

Observational refinement can be formalized as a trace inclusion problem, and the latter can itself be reduced to an invariant checking problem, but this requires in general introducing history and prophecy variables [1]. Alternatively, verifying refinement requires in general establishing a forward simulation *and* a backward simulation [21]. While simulations are natural concepts, backward reasoning, corresponding to the use of prophecy variables, is in general hard and complex for programs manipulating data structures. Therefore, a crucial issue is to understand the limits of forward reasoning in establishing refinement. More precisely, an important question is to determine for

* An extended version of this paper including the missing proofs can be found at [8].

which concurrent abstract data structures, and for which classes of implementations, it is possible to carry out a refinement proof using only forward simulations.

To get rid of backward simulations (or prophecy variables) while preserving completeness w.r.t. refinement, it is necessary to have reference implementations that are *deterministic*. Interestingly, determinism allows also to simplify the forward simulation checking problem. Indeed, in this case, this problem can be reduced to an invariant checking problem. Basically, the simulation relation can be seen as an invariant of the system composed of the two compared programs. Therefore, existing methods and tools for invariant checking can be leveraged in this context.

But, in order to determine precisely what is meant by determinism, an important point is to fix the alphabet of observable events along computations. Typically, to reason about refinement between two library implementations, the only observable events are the calls and returns corresponding to the method invocations along computations. This means that only the external interface of the library is considered to compare behaviors, and nothing else from the implementations is exposed. Unfortunately, it can be shown that in this case, it is impossible to have deterministic atomic reference implementations for common data structures such as stacks and queues (see, e.g., [24]). Then, an important question is what is the necessary amount of information that should be exposed by the implementations to overcome this problem ?

One approach addressing this question is based on linearizability [18] and its correspondence with refinement [12, 7]. Linearizability of a computation (of some implementation) means that each of the method invocations can be seen as happening at some point, called *linearization point*, occurring somewhere between the call and return events of that invocation. The obtained sequence of linearization points along the computation should define a sequence of operations that is possible in the atomic reference implementation. Proving the existence of such sequences of linearization points, for all the computations of a concurrent library, is a complex problem [3, 5, 14]. However, proving linearizability becomes less complex when linearization points are fixed for each method, i.e., associated with the execution of a designated statement in its source code [5]. In this case, we can consider that libraries expose in addition to calls and returns, events signaling linearization points. By extending this way the alphabet of observable events, it becomes straightforward to define *deterministic* atomic reference implementations. Therefore, proving linearizability can be carried out using forward simulations when linearization points are fixed, e.g., [28, 4, 27, 2]. Unfortunately, this approach is not applicable to efficient implementations such as the LCRQ queue [22] (based on the principle of the Herlihy&Wing queue [18]), and the Time-Stamped Stack [10]. The proofs of linearizability of these implementations are highly nontrivial, very involved, and hard to read, understand and automatize. Therefore, the crucial question we address is what is precisely the kind of information that is necessary to expose in order to obtain deterministic atomic reference implementations for such data structures, allowing to derive simple and natural linearizability proofs for such complex implementations, based on forward simulations, that are amenable to automation ?

We observe that the main difficulty in reasoning about these implementations is that, linearization points of enqueue/push operations occurring along some given computation, depend in general on the linearization points of dequeue/pop operations that occur

arbitrarily far in the future. Therefore, since linearization points for enqueue/push operations cannot be determined in advance, the information that could be fixed and exposed can concern only the dequeue/pop operations.

One first idea is to consider that linearization points are fixed for dequeue/pop methods and only for these methods. We show that under the assumption that implementations expose linearizations points for these methods, it is possible to define deterministic atomic reference implementations for both queues and stacks. We show that this is indeed useful by providing a simple proof of the Herlihy&Wing queue (based on establishing a forward simulation) that can be carried out as an invariant checking proof.

However, in the case of Time-Stamped Stack, fixing linearization points of pop operations is actually too restrictive. Nevertheless, we show that our approach can be generalized to handle this case. The key idea is to reason about what we call *commit points*, and that correspond roughly speaking to the last point a method accesses to the shared data structure during its execution. We prove that by exposing commit points (instead of linearization points) for pop methods, we can still provide deterministic reference implementations. We show that using this approach leads to a quite simple proof of the Time-Stamped Stack, based on forward simulations.

2 Preliminaries

We formalize several abstraction relations between libraries using a simple yet universal model of computation, namely labeled transition systems (LTS). This model captures shared-memory programs with an arbitrary number of threads, abstracting away the details of any particular programming language irrelevant to our development.

A *labeled transition system* (LTS) $A = (Q, \Sigma, s_0, \delta)$ over the possibly-infinite alphabet Σ is a possibly-infinite set Q of states with initial state $s_0 \in Q$, and a transition relation $\delta \subseteq Q \times \Sigma \times Q$. The ith symbol of a sequence $\tau \in \Sigma^*$ is denoted τ_i, and ε denotes the empty sequence. An *execution* of A is an alternating sequence of states and transition labels (also called actions) $\rho = s_0, e_0, s_1 \dots e_{k-1}, s_k$ for some $k > 0$ such that $\delta(s_i, e_i, s_{i+1})$ for each i such that $0 \leq i < k$. We write $s_i \xrightarrow{e_i \dots e_{j-1}}_A s_j$ as shorthand for the subsequence $s_i, e_i, \dots, s_{j-1}, e_{j-1}, s_j$ of ρ, for any $0 \leq i \leq j < k$ (in particular $s_i \xrightarrow{\varepsilon} s_i$). The projection $\tau|\Gamma$ of a sequence τ is the maximum subsequence of τ over alphabet Γ. This notation is extended to sets of sequences as usual. A *trace* of A is the projection $\rho|\Sigma$ of an execution ρ of A. The set of executions, resp., traces, of an LTS A is denoted by $E(A)$, resp., $Tr(A)$. An LTS is *deterministic* if for any state s and any sequence $\tau \in \Sigma^*$, there is at most one state s' such that $s \xrightarrow{\tau} s'$. More generally, for an alphabet $\Gamma \subseteq \Sigma$, an LTS is Γ-*deterministic* if for any state s and any sequence $\tau \in \Gamma^*$, there is at most one state s' such that $s \xrightarrow{\tau'} s'$ and τ is a subsequence of τ'.

2.1 Libraries

Programs interact with libraries by calling named library *methods*, which receive *arguments* and yield *return values* upon completion. We fix arbitrary sets $\mathbb{M}$ and $\mathbb{V}$ of method names and argument/return values. We fix an arbitrary set $\mathbb{O}$ of operation identifiers, and for given sets $\mathbb{M}$ and $\mathbb{V}$ of methods and values, we fix the sets

$C = \{inv(m,d,k) : m \in \mathbb{M}, d \in \mathbb{V}, k \in \mathbb{O}\}$ and $R = \{ret(m,d,k) : m \in \mathbb{M}, d \in \mathbb{V}, k \in \mathbb{O}\}$ of *call actions* and *return actions*; each call action $inv(m,d,k)$ combines a method $m \in \mathbb{M}$ and value $d \in \mathbb{V}$ with an *operation identifier* $k \in \mathbb{O}$. Operation identifiers are used to pair call and return actions. We may omit the second field from a call/return action for methods that have no arguments or return values. For notational convenience, we take $\mathbb{O} = \mathbb{N}$ for the rest of the paper.

A *library* is an LTS over alphabet Σ such that $C \cup R \subseteq \Sigma$. We assume that the traces of a library satisfy standard well-formedness properties, e.g., return actions correspond to previous call actions. Given a standard library description as a set of methods, the LTS represents the executions of its most general client (that calls an arbitrary set of methods with an arbitrary set of threads in an unbounded loop). The states of this LTS consist of the shared state of the library together with the local state of each thread. The transitions correspond to statements in the methods' bodies, or call and return actions. An operation k is called *completed* in a trace τ when $ret(m,d,k)$ occurs in τ, for some m and d. Otherwise, it is called *pending*.

The projection of a library trace over $C \cup R$ is called a *history*. The set of histories of a library L is denoted by $H(L)$. Since libraries only dictate methods executions between their respective calls and returns, for any history they admit, they must also admit histories with weaker inter-operation ordering, in which calls may happen earlier, and/or returns later. A history h_1 is *weaker* than a history h_2, written $h_1 \sqsubseteq h_2$, iff there exists a history h_1' obtained from h_1 by appending return actions, and deleting call actions, s.t.: h_2 is a permutation of h_1' that preserves the order between return and call actions, i.e., if a given return action occurs before a given call action in h_1', then the same holds in h_2.

A library L is called *atomic* when there exists a set S of sequential histories such that $H(L)$ contains every weakening of a history in S. Atomic libraries are often considered as specifications for concurrent objects. Libraries can be made atomic by guarding their methods bodies with global lock acquisitions.

A library L is called a *queue implementation* when $\mathbb{M} = \{enq, deq\}$ (enq is the method that enqueues a value and deq is the method removing a value) and $\mathbb{V} = \mathbb{N} \cup \{\texttt{EMPTY}\}$ where `EMPTY` is the value returned by deq when the queue is empty. Similarly, a library L is called a *stack implementation* when $\mathbb{M} = \{push, pop\}$ and $\mathbb{V} = \mathbb{N} \cup \{\texttt{EMPTY}\}$. For queue and stack implementations, we assume that the same value is never added twice, i.e., for every trace τ of such a library and every two call actions $inv(m,d_1,k_1)$ and $inv(m,d_2,k_2)$ where $m \in \{enq, push\}$ we have that $d_1 \neq d_2$. As shown in several works [2, 6], this assumption is without loss of generality for libraries that are data independent, i.e., their behaviors are not influenced by the values added to the collection. All the queue and stack implementations that we are aware of are data independent. On a technical note, this assumption is used to define (Γ-)deterministic abstract implementations of stacks and queues in Section 4 and Section 5.

2.2 Refinement and Linearizability

Conformance of a library L_1 to a specification given as an "abstract" library L_2 is formally captured by *(observational) refinement*. Informally, we say L_1 refines L_2 iff every computation of every program using L_1 would also be possible were L_2 used instead. We assume that a program can interact with the library only through call and return actions,

and thus refinement can be defined as history set inclusion. Refinement is equivalent to the *linearizability* criterion [18] when L_2 is an atomic library [12, 7].

Definition 1. *A library* L_1 refines *another library* L_2 *iff* $H(L_1) \subseteq H(L_2)$.

Linearizability [18] requires that every history of a concurrent library L_1 can be "linearized" to a sequential history admitted by a library L_2 used as a specification. Formally, a sequential history h_2 with only complete operations is called a *linearization* of a history h_1 when $h_1 \sqsubseteq h_2$. A history h_1 is *linearizable* w.r.t. a library L_2 iff there exists a linearization h_2 of h_1 such that $h_2 \in H(L_2)$. A library L_1 is *linearizable* w.r.t. L_2, written $L_1 \sqsubseteq L_2$, iff each history $h_1 \in H(L_1)$ is linearizable w.r.t. L_2.

Theorem 1 ([12, 7]). *Let* L_1 *and* L_2 *be two libraries, such that* L_2 *is atomic. Then,* $L_1 \sqsubseteq L_2$ *iff* L_1 *refines* L_2.

In the rest of the paper, we discuss methods for proving refinement (and thus, linearizability) focusing mainly on queue and stack implementations.

3 Refinement Proofs

Library refinement is an instance of a more general notion of refinement between LTSs, which for some alphabet Γ of *observable actions* is defined as the inclusion of sets of traces projected on Γ. Library refinement corresponds to the case $\Gamma = C \cup R$. Typically, Γ-refinement between two LTSs A_1 and A_2 is proved using *simulation relations* which roughly, require that A_2 can mimic every step of A_1 using a (possibly empty) sequence of steps. Mainly, there are two kinds of simulation relations, forward or backward, depending on whether the preservation of steps is proved starting from a similar state forward or backward. It has been shown that Γ-refinement is equivalent to the existence of *backward simulations*, modulo the addition of history variables that record events in the implementation, and to the existence of *forward simulations* provided that the right-hand side LTS, A_2, is Γ-deterministic [1, 21]. We focus on proofs based on forward simulations because they are easier to automatize.

In general, forward simulations are *not* a complete proof method for library refinement because libraries are not $C \cup R$-deterministic (the same sequence of call/return actions can lead to different states depending on the interleaving of the internal actions). However, there are classes of atomic libraries, e.g., libraries with "fixed linearization points" (defined later in this section), for which it is possible to identify a larger alphabet Γ of observable actions (including call/return actions), and implementations that are Γ-deterministic. For queues and stacks, Section 4 and Section 5 define other such classes of implementations that cover all the implementations that we are aware of.

Let $A_1 = (Q_1, \Sigma, s_0^1, \delta_1)$ and $A_2 = (Q_2, \Sigma, s_0^2, \delta_2)$ be two LTSs over Σ_1 and Σ_2, respectively, and Γ an alphabet, such that $\Gamma \subseteq \Sigma_1 \cap \Sigma_2$.

Definition 2. *The LTS* A_1 Γ-refines A_2 *iff* $Tr(A_1)|\Gamma \subseteq Tr(A_2)|\Gamma$.

The notion of Γ-refinement instantiated to libraries (i.e., to LTSs defining libraries) implies the notion of refinement in Definition 1 for every Γ such that $C \cup R \subseteq \Gamma$.

We define a notion of *forward* simulation that can be used to prove Γ-refinement

Definition 3. *A relation* $F \subseteq Q_1 \times Q_2$ *is called a* Γ-forward simulation *from* A_1 *to* A_2 *iff* $F(s_0^1, s_0^2)$ *and:*

- *For all* $s, s' \in Q_1$, $\gamma \in \Gamma$, *and* $u \in Q_2$, *such that* $(s, \gamma, s') \in \delta_1$ *and* $F(s,u)$, *we have that there exists* $u' \in Q_2$ *such that* $F(s',u')$ *and* $u \xrightarrow{\sigma} u'$ *where* $\sigma_i = \gamma$, *for some i, and* $\sigma_j \in \Sigma_2 \setminus \Gamma$, *for all* $j \neq i$.
- *For all* $s, s' \in Q_1$, $e \in \Sigma_1 \setminus \Gamma$, *and* $u \in Q_2$, *such that* $(s, e, s') \in \delta_1$ *and* $F(s,u)$, *we have that there exists* $u' \in Q_2$ *such that* $F(s',u')$ *and* $u \xrightarrow{\sigma} u'$ *where* $\sigma \in (\Sigma_2 \setminus \Gamma)^*$.

A Γ-forward simulation states that every step of A_1 is simulated by a sequence of steps of A_2. To imply Γ-refinement, every step of A_1 labeled by an observable action $\gamma \in \Gamma$ should be simulated by a sequence of steps of A_2 where exactly one transition is labeled by γ and all the other transitions are labeled by non-observable actions. The dual notion of *backward* simulation where steps are simulated backwards can be defined similarly.

The following shows the soundness and the completeness of Γ-forward simulations (when A_2 is Γ-deterministic). It is an instantiation of previous results [1, 21].

Theorem 2. *If there is a* Γ*-forward simulation from* A_1 *to* A_2, *then* A_1 Γ*-refines* A_2. *Also, if* A_1 Γ*-refines* A_2 *and* A_2 *is* Γ*-deterministic, then there is a* Γ*-forward simulation from* A_1 *to* A_2.

The linearization of a concurrent history can be also defined in terms of *linearization points*. Informally, a linearization point of an operation in an execution is a point in time where the operation is conceptually effectuated; given the linearization points of each operation, the linearization of a concurrent history is the sequential history which takes operations in the order of their linearization points. For some libraries, the linearization points of all the invocations of a method m correspond to the execution of a fixed statement in m's body. For instance, when method bodies are guarded with a global-lock acquisition, the linearization point of every method invocation corresponds to the execution of the body. When the linearization points are fixed, we assume that the library is an LTS over an alphabet that includes actions $lin(m,d,k)$ with $m \in \mathbb{M}$, $d \in \mathbb{V}$ and $k \in \mathbb{O}$, representing the linearization point of the operation k returning value d. Let *Lin* denote the set of such actions. The projection of a library trace over $C \cup R \cup Lin$ is called an *extended history*. A trace or extended history is called *Lin-complete* when every completed operation has a linearization point, i.e., each return action $ret(m,d,k)$ is preceded by an action $lin(m,d,k)$. A library L over alphabet Σ is called *with fixed linearization points* iff $C \cup R \cup Lin \subseteq \Sigma$ and every trace $\tau \in Tr(L)$ is *Lin*-complete.

Proving the correctness of an implementation L_1 of a concurrent object such as a queue or a stack with fixed linearization points reduces to proving that L_1 is a $(C \cup R \cup Lin)$-refinement of an abstract implementation L_2 of the same object where method bodies are guarded with a global-lock acquisition. As a direct consequence of Theorem 2, since the abstract implementation is $(C \cup R \cup Lin)$-deterministic, proving $(C \cup R \cup Lin)$-refinement is equivalent to finding a $(C \cup R \cup Lin)$-forward simulation from L_1 to L_2.

Section 4 and Section 5 extend this result to queue and stack implementations where the linearization point of the methods *adding* values to the collection is *not* fixed.

4 Queues With Fixed Dequeue Linearization Points

The classical abstract queue implementation, denoted $AbsQ_0$, maintains a sequence of enqueued values; dequeues return the oldest non-dequeued value, at the time of their

```
void enq(int x) {
  i = back++; items[i] = x;
}
int deq() {
  while (1) {
    range = back - 1;
    for (int i = 0; i <= range; i++) {
      x = swap(items[i],null);
      if (x != null) return x;
} } }
```

Fig. 1. The Herlihy & Wing Queue [18].

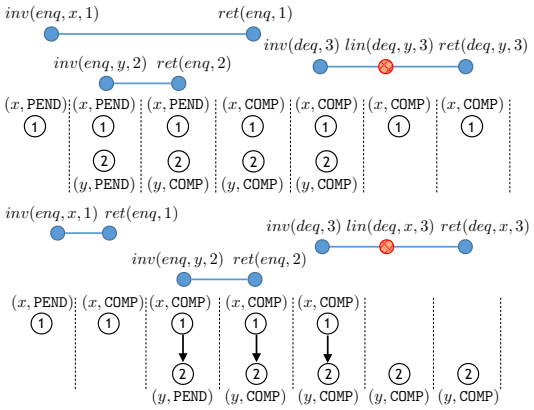

Fig. 2. Forward simulation with *AbsQ*. Lines depict operations, and circles depict call, return, and linearization point actions.

```
function loc: 𝕆 → {inv,lin,ret,⊥}
function arg, ret: 𝕆 → 𝕍
function present, pending: 𝕆 → 𝔹
function before: 𝕆×𝕆 → 𝔹

rule inv(enq,v,k):
  arg(k) := v
  present(k) := true
  pending(k) := true
  forall k1 with present(k1):
    if ¬pending(k1):
      before(k1,k) := true

rule ret(enq,k):
  pending(k) := false

rule inv(deq,k):
  pass

rule lin(deq,v,k):
  ret(k) := v
  if v = EMPTY:
    forall k' with present(k'):
      assert pending(k')
  else:
    let k1 = arg⁻¹(v)
    assert present(k1)
    forall k2 with present(k2):
      assert ¬before(k2,k1)
    present(k1) := false

rule ret(deq,v,k):
  assert ret(k) = v
```

Fig. 3. The *AbsQ* implementation; each rule α(_,k) implicitly begins with **assert** loc(k)=α and ends with the appropriate loc(k):=β.

linearization points, or EMPTY. Some implementations, like the queue of Herlihy and Wing [18], denoted HWQ and listed in Figure 1, are not forward-simulated by $AbsQ_0$, even though they refine $AbsQ_0$, since the order in which their enqueues are linearized to form $AbsQ_0$s sequence is not determined until later, when their values are dequeued.

In this section we develop an abstract queue implementation, denoted *AbsQ*, which maintains a partial order of enqueues, rather than a linear sequence. Since *AbsQ* does not force refining implementations to eagerly pick among linearizations of their enqueues, it forward-simulates many more queue implementations. In fact, *AbsQ* forward-simulates all queue implementations of which we are aware that are not forward-simulated by $AbsQ_0$, including HWQ, The Baskets Queue [19], The Linked Concurrent Ring Queue (LCRQ) [22], and The Time-Stamped Queue [10].

4.1 Enqueue Methods With Non-Fixed Linearization Points

We describe *HWQ* where the linearization points of the enqueue methods are not fixed. The shared state consists of an array items storing the values in the queue and a counter back storing the index of the first unused position in items. Initially, all the positions in

the array are `null` and `back` is 0. An enqueue method starts by reserving a position in `items` (`i` stores the index of this position and `back` is incremented so the same position cannot be used by other enqueues) and then, stores the argument `x` at this position. The dequeue method traverses the array `items` starting from the beginning and atomically swaps `null` with the encountered value. If the value is not `null`, then the dequeue returns that value. If it reaches the end of the array, then it restarts.

The linearization points of the enqueues are not fixed, they depend on dequeues executing in the future. Consider the following trace with two concurrent enqueues ($\texttt{i}(k)$ represents the value of `i` in operation k): $inv(enq,x,1)$, $inv(enq,y,2)$, $\texttt{i}(1) =$ `bck++`, $\texttt{i}(2) =$ `bck++`, $\texttt{items}[\texttt{i}(2)] = y$. Assuming that the linearization point corresponds to the assignment of `i`, the history of this trace should be linearized to $inv(enq,x,1)$, $ret(enq,1)$, $inv(enq,y,2)$, $ret(enq,2)$. However, a dequeue executing until completion after this trace will return y (only position 1 is filled in the array `items`) which is not consistent with this linearization. On the other hand, assuming that enqueues should be linearized at the assignment of `items[i]` and extending the trace with $\texttt{items}[\texttt{i}(1)] = x$ and a completed dequeue that in this case returns x, leads to the incorrect linearization: $inv(enq,y,2)$, $ret(enq,2)$, $inv(enq,x,1)$, $ret(enq,1)$, $inv(deq,3)$, $ret(deq,x,3)$.

The dequeue method has a fixed linearization point which corresponds to an execution of `swap` returning a non-null value. This action alone contributes to the effect of that value being removed from the queue. Every concurrent history can be linearized to a sequential history where dequeues occur in the order of their linearization points in the concurrent history. This claim is formally proved in Section 4.3.

Since the linearization points of the enqueues are determined by future dequeue invocations, there exists no forward simulation from *HWQ* to $AbsQ_0$. In the following, we describe the abstract implementation *AbsQ* for which such a forward simulation does exist.

4.2 Abstract Queue Implementation

Informally, *AbsQ* records the set of enqueue operations, whose argument has not yet been removed by a matching dequeue operation. In addition, it records the happens-before order between those enqueue operations: this is a partial order ordering an enqueue k_1 before another enqueue k_2 iff k_1 returned before k_2 was invoked. The linearization point of a dequeue can either remove a minimal enqueue k (w.r.t. the happens-before stored in the state) and fix the return value to the value d added by k, or fix the return value to `EMPTY` provided that the current state stores only pending enqueues (intuitively, the dequeue overlaps with all the enqueue operations stored in the current state and it can be linearized before all of them).

Fig. 2 pictures two executions of *AbsQ* for two extended histories (that include dequeue linearization points). The state of *AbsQ* after each action is pictured as a graph below the action. The nodes of this graph represent enqueue operations and the edges happens-before constraints. Each node is labeled by a value (the argument of the enqueue) and a flag `PEND` or `COMP` showing whether the operation is pending or completed. For instance, in the case of the first history, the dequeue linearization point $lin(deq,y,3)$ is enabled because the current happens-before contains a *minimal* enqueue operation with argument y. Note that a linearization point $lin(deq,x,3)$ is also enabled at this state.

We define *AbsQ* with the abstract state machine given in Figure 3 which defines an LTS over the alphabet $C \cup R \cup Lin(deq)$. The state of *AbsQ* consists of several updatable functions: `loc` indicates the abstract control point of a given operation; `arg` and `ret` indicate the argument or return value of a given operation, respectively; `present` indicates whether a given enqueue operation has yet to be removed, and `pending` indicates whether it has yet to complete; `before` indicates the happens-before order between operations. Initially, `loc(k) = inv` for all `k`, and `present(k1) = pending(k1) = before(k1,k2) = false` for all `k1, k2`. Each rule determines the next state of *AbsQ* for the corresponding action. For instance, the `lin(deq,v,k)` rule updates the state of *AbsQ* for the linearization action of a dequeue operation with identifier `k` returning value `v`: when `v = EMPTY` then *AbsQ* insists via an assertion that any still-present enqueue must still be pending; otherwise, when $\texttt{v} \neq \texttt{k}$ then *AbsQ* insists that a corresponding enqueue is present, and that it is minimal in the happens-before order, before marking that enqueue as not present. Updates to the `loc` function, implicit in Figure 3, ensure that the invocation, linearization-point, and return actions of each operation occur in the correct order.

The following result states that the library *AbsQ* has exactly the same set of histories as the standard abstract library $AbsQ_0$.

Theorem 3. *AbsQ is a refinement of $AbsQ_0$ and vice-versa.*

A trace of a queue implementation is called $Lin(deq)$*-complete* when every completed dequeue has a linearization point, i.e., each return action $ret(deq,d,k)$ is preceded by an action $lin(deq,d,k)$. A queue implementation L over alphabet Σ, such that $C \cup R \cup Lin(deq) \subseteq \Sigma$, is called *with fixed dequeue linearization points* when every trace $\tau \in Tr(L)$ is $Lin(deq)$-complete.

The following result shows that $C \cup R \cup Lin(deq)$-forward simulations are a sound and complete proof method for showing the correctness of a queue implementation with fixed dequeue linearization points (up to the correctness of the linearization points). It is obtained from Theorem 3 and Theorem 2 using the fact that the alphabet of *AbsQ* is exactly $C \cup R \cup Lin(deq)$ and *AbsQ* is deterministic. The determinism of *AbsQ* relies on the assumption that every value is added at most once. Without this assumption, *AbsQ* may reach a state with two enqueues adding the same value being both minimal in the happens-before. A transition corresponding to the linearization point of a dequeue from this state can remove any of these two enqueues leading to two different states. Therefore, *AbsQ* becomes non-deterministic. Note that this is independent of the fact that *AbsQ* manipulates operation identifiers.

Corollary 1. *A queue implementation L with fixed dequeue linearization points is a $C \cup R \cup Lin(deq)$-refinement of $AbsQ_0$ iff there exists a $C \cup R \cup Lin(deq)$-forward simulation from L to AbsQ.*

4.3 A Correctness Proof For Herlihy&Wing Queue

We describe a forward simulation F_1 from *HWQ* to *AbsQ*. The description of *HWQ* in Fig. 1 defines an LTS whose states contain the shared array `items` and the shared counter `back` together with a valuation for the local variables `i`, `x`, and `range`, and the

control location of each operation. A transition is either a call or a return action, or a statement in one of the two methods `enq` or `deq`.

An *HWQ* state s is related by F_1 to *AbsQ* states t where the predicate present is true for all the enqueues in s whose argument is stored in the array `items`, and all the pending enqueues that have not yet written to the array `items` (and only for these enqueues). We refer to such enqueues in s as present enqueues. Also, $\mathsf{pending}(k)$ is `true` in t whenever k is a pending enqueue in s, $\mathsf{arg}(k) = d$ in t whenever the argument of the enqueue k in s is d, and for every dequeue operation k such that $\mathtt{x}(k) = d \neq \mathtt{null}$, we have that $\mathtt{y}(k) = d$ (recall that y is a local variable of the dequeue method in *AbsQ*). The order relation before in t satisfies the following constraints:

(a) pending enqueues are maximal, i.e., for every two present enqueues k and k' such that k' is pending, we have that $\neg\mathsf{before}(k',k)$,

(b) before is consistent with the order in which positions of `items` have been reserved, i.e., for every two present enqueues k and k' such that $\mathtt{i}(k) < \mathtt{i}(k')$, we have that $\neg\mathsf{before}(k',k)$,

(c) if the position i reserved by an enqueue k has been "observed" by a non-linearized dequeue that in the current array traversal may "observe" a later position j reserved by another enqueue k', then k can't be ordered before k', i.e., for every two present enqueues k and k', and a dequeue k_d, such that

$$\mathtt{canRemove}(k_d,k') \wedge (\mathtt{i}(k) < \mathtt{i}(k_d) \vee (\mathtt{i}(k) = \mathtt{i}(k_d) \wedge \mathtt{afterSwapNull}(k_d))) \quad (1)$$

we have that $\neg\mathsf{before}(k,k')$. The predicate $\mathtt{canRemove}(k_d,k')$ holds when k_d visited a `null` item in `items` and the position $i(k')$ reserved by k' is in the range of (k_d) i.e., $(\mathtt{x}(k_d) = \mathtt{null} \wedge \mathtt{i}(k_d) < \mathtt{i}(k') \leq \mathtt{range}(k_d)) \vee (\mathtt{i}(k_d) = \mathtt{i}(k') \wedge \mathtt{beforeSwap}(k_d) \wedge \mathtt{items}[\mathtt{i}(k')] \neq \mathtt{null})$. The predicate $\mathtt{afterSwapNull}(k_d)$ (resp., $\mathtt{beforeSwap}(k_d)$) holds when the dequeue k_d is at the control point after a `swap` returning `null` (resp., before a `swap`).

The constraints on before ensure that a present enqueue whose argument is about to be removed by a dequeue operation is minimal. Thus, let k' be a present enqueue that inserted its argument to `items`, and k_d a pending dequeue such that $\mathtt{canRemove}(k_d,k')$ holds and k_d is just before its swap action at the reserved position of k' i.e., $\mathtt{i}(k_d) = \mathtt{i}(k')$. Another pending enqueue k cannot be ordered before k' since pending enqueues are maximal by (a). Regarding the completed and present enqueues k, we consider two cases: $i(k) > i(k')$ and $i(k) < i(k')$. For the former case, the constraint (b) ensures $\neg\mathsf{before}(k,k')$ and for the latter case the constraint (c) ensures $\neg\mathsf{before}(k,k')$. Consequently, k' is a minimal element w.r.t. before just before k_d removes its argument.

Next, we show that F_1 is indeed a $C \cup R \cup Lin(deq)$-forward simulation. Let s and t be states of *HWQ* and *AbsQ*, respectively, such that $(s,t) \in F_1$. We omit discussing the trivial case of transitions labeled by call and return actions which are simulated by similar transitions of *AbsQ*.

We show that each internal step of an enqueue or dequeue, except a `swap` returning a non-null value in dequeue (which represents its linearization point), is simulated by an *empty* sequence of *AbsQ* transitions, i.e., for every state s' obtained through one of these steps, if $(s,t) \in F_1$, then $(s',t) \in F_1$ for each *AbsQ* state t. Essentially, this consists in proving the following property, called *monotonicity*: the set of possible before relations associated by F_1 to s' doesn't exclude any order before associated to s.

Concerning enqueue rules, let s' be the state obtained from s when a pending enqueue k reserves an array position. This enqueue must be maximal in both t and any state t' related to s' (since it's pending). Moreover, there is no dequeue that can "observe" this position before restarting the array traversal. Therefore, item (c) in the definition of F_1 doesn't constrain the order between k and some other enqueue neither in s nor in s'. Since this transition doesn't affect the constraints on the order between enqueues different from k (their local variables remain unchanged), monotonicity holds. This property is trivially satisfied by the second step of enqueue which doesn't affect `i`.

To prove monotonicity in the case of dequeue internal steps different from its linearization point, it is important to track the non-trivial instantiations of item (c) in the definition of before over the two states s and s', i.e., the triples (k,k',k_d) for which (1) holds. Instantiations that are enabled only in s' may in principle lead to a violation of monotonicity (since they restrict the orders before associated to s'). For the two steps that begin an array traversal, i.e., reading the index of the last used position and setting `i` to 0, there exist no such new instantiations in s' because the value of `i` is either not set or 0. The same is true for the increment of `i` in a dequeue k_d since the predicate $\texttt{afterSwapNull}(k_d)$ holds in state s. The execution of `swap` returning `null` in a dequeue k_d enables new instantiations (k,k',k_d) in s', thus adding potentially new constraints $\neg\mathsf{before}(k,k')$. We show that these instantiations are however vacuous because k must be pending in s and thus maximal in every order before associated by F_1 to s. Let k and k' be two enqueues such that together with the dequeue k_d they satisfy the property (1) in s' but not in s. We write $\texttt{i}_s(k)$ for the value of the variable `i` of operation k in state s. We have that $\texttt{i}_{s'}(k) = \texttt{i}_{s'}(k_d) \leq \texttt{i}_{s'}(k')$ and $\texttt{items}[\texttt{i}_{s'}(k_d)] = \texttt{null}$. The latter implies that the enqueue k didn't execute the second statement (since the position it reserved is still `null`) and it is pending in s'. The step that swaps the null item does not modify anything except the control point of k_d that makes $\texttt{afterSwapNull}(k_d)$ true in s' . Hence, $i_s(k) = i_s(k_d) \leq i_s(k')$ and $\texttt{items}[i_s(k_d)] = \texttt{null}$ is also true. Therefore, k is pending in s and maximal. Hence, $\mathsf{before}(k,k')$ is not true in both s and s'.

Finally, we show that the linearization point of a dequeue k of *HWQ*, i.e., an execution of `swap` returning a non-null value d, from state s and leading to a state s' is simulated by a transition labeled by $lin(deq,d,k)$ of *AbsQ* from state t. By the definition of *HWQ*, there is a unique enqueue k_e which filled the position updated by k, i.e., $\texttt{i}_s(k_e) = i_s(k)$ and $\texttt{x}_{s'}(k) = \texttt{x}_s(k_e)$.

We show that k_e is minimal in the order before of t which implies that k_e could be chosen by $lin(deq,d,k)$ step applied on t. As explained previously, instantiating item (c) in the definition of before with $k' = k_e$ and $k_d = k$, and instantiating item (b) with $k = k_e$, we ensure the minimality of k_e. Moreover, the state t' obtained from t through a $lin(deq,d,k)$ transition is related to s' because the value added by k_e is not anymore present in `items` and $\mathsf{present}(k_e)$ doesn't hold in t'.

5 Stacks With Fixed Pop Commit Points

The abstract implementation in Section 4 can be adapted to stacks, the main modification being that the linearization point $lin(pop,d,k)$ with $d \neq \texttt{EMPTY}$ is enabled when k is added by a push which is maximal in the happens-before order stored in the state. However, there are stack implementations, e.g., Time-Stamped Stack [10] (*TSS*, for short),

```
struct Node{
  int data;
  int ts;
  Node* next;
  bool taken;
};

bool CAS(bool data, bool a, bool b);

Node* pools[maxThreads];
int TS = 0;

void push(int x) {
  Node* n =
    new Node(x,MAX_INT, null,false);
  n->next = pools[myTID];
  pools[myTID] = n;
  int i = TS++;
  n->ts = i;
}
```

```
int pop() {
  bool success = false;
  int maxTS = -1;
  Node* youngest = null;
  while ( !success ) {
    maxTS = -1; youngest = null;
    for(int i=0; i<maxThreads; i++) {
      Node* n = pools[i];
      while (n->taken && n->next != n)
        n = n->next;
      if(maxTS < n->ts) {
        maxTS = n->ts; youngest = n;
      }
    }
    if (youngest != null)
      success =
        CAS(youngest->taken, false, true);
  }
  return youngest->data;
}
```

Fig. 4. The Time-Stamped Stack [10].

which cannot be proved correct using forward simulations to this abstract implementation because the linearization points of the pop operations are not fixed. Exploiting particular properties of the stack semantics, we refine the ideas used in *AbsQ* and define a new abstract implementation for stacks, denoted as *AbsS*, which is able to simulate such implementations. Forward simulations to *AbsS* are complete for proving the correctness of stack implementations provided that the point in time where the return value of a pop operation is determined, called *commit point*, corresponds to a fixed statement in the pop method.

5.1 Pop Methods With Fixed Commit Points

We explain the meaning of the commit points on a simplified version of the Time-Stamped Stack [10] (*TSS*, for short) given in Fig. 4. This implementation maintains an array of singly-linked lists, one for each thread, where list nodes contain a data value (field `data`), a timestamp (field `ts`), the next pointer (field `next`), and a Boolean flag indicating whether the node represents a value removed from the stack (field `taken`). Initially, each list contains a sentinel dummy node pointing to itself with timestamp -1 and the flag `taken` set to `false`.

Pushing a value to the stack proceeds in several steps: adding a node with maximal timestamp in the list associated to the thread executing the push (given by the special variable `myTID`), asking for a new timestamp (given by the shared variable `TS`), and updating the timestamp of the added node. Popping a value from the stack consists in traversing all the lists, finding the first element which doesn't represent a removed value (i.e., `taken` is `false`) in each list, and selecting the element with the maximal timestamp. A compare-and-swap (CAS) is used to set the `taken` flag of this element to `true`. The procedure restarts if the CAS fails.

The push operations don't have a fixed linearization point because adding a node to a list and updating its timestamp are not executed in a single atomic step. The nodes

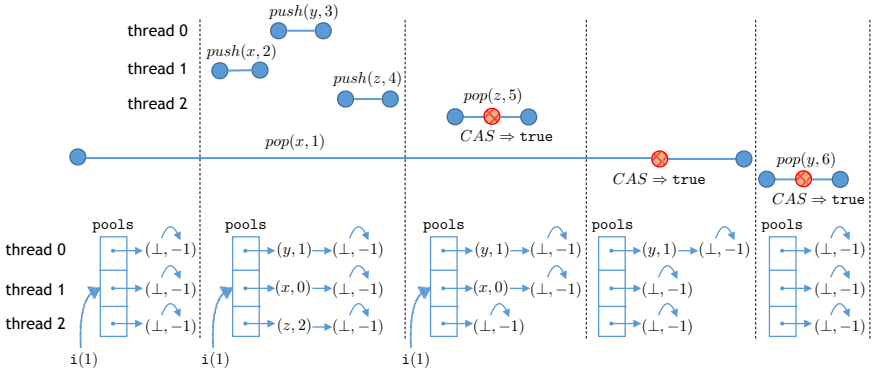

Fig. 5. An execution of *TSS*. An operation is pictured by a line delimited by two circles denoting the call and respectively, the return action. Pop operations with identifier k and removing value d are labeled $pop(d,k)$. Their representation includes another circle that stands for a successful CAS which is their commit point. The library state after an execution prefix delimited at the right by a dotted line is pictured in the bottom part (the picture immediately to the left of the dotted line). A pair (d,t) represents a list node with `data` $= d$ and `ts` $= t$, and `i`(1) denotes the value of `i` in the pop with identifier 1. We omit the nodes where the field `taken` is `true`.

can be added in an order which is not consistent with the order between the timestamps assigned later in the execution. Also, the value added by a push that just added an element to a list can be popped before the value added by a completed push (since it has a maximal timestamp). The same holds for pop operations: The only reasonable choice for a linearization point is a successful CAS (that results in updating the field `taken`). Fig. 5 pictures an execution showing that this action doesn't correspond to a linearization point, i.e., an execution for which the pop operations in every correct linearization are not ordered according to the order between successful CASs. In every correct linearization of that execution, the pop operation removing x is ordered before the one removing z although they perform a successful CAS in the opposite order.

An interesting property of the successful CASs in pop operations is that they fix the return value, i.e., the return value is `youngest->data` where `youngest` is the node updated by the CAS. We call such actions *commit points*. More generally, commit points are actions that access shared variables, from which every control-flow path leads to the return control point and contains no more accesses to the shared memory (i.e., after a commit point, the return value is computed using only local variables).

When the commit points of pop operations are fixed to particular implementation actions (e.g., a successful CAS) we assume that the library is an LTS over an alphabet that contains actions $com(pop,d,k)$ with $d \in \mathbb{V}$ and $k \in \mathbb{O}$ (denoting the commit point of the pop with identifier k and returning d). Let $Com(pop)$ be the set of such actions.

5.2 Abstract stack implementation

We define an abstract stack *AbsS* over alphabet $C \cup R \cup Com(pop)$ that essentially, similarly to *AbsQ*, maintains the happens-before order of the pushes whose value has not yet been removed by a matching pop. Pop operations are treated differently since the commit points are not necessarily linearization points. Intuitively, a pop can be linearized

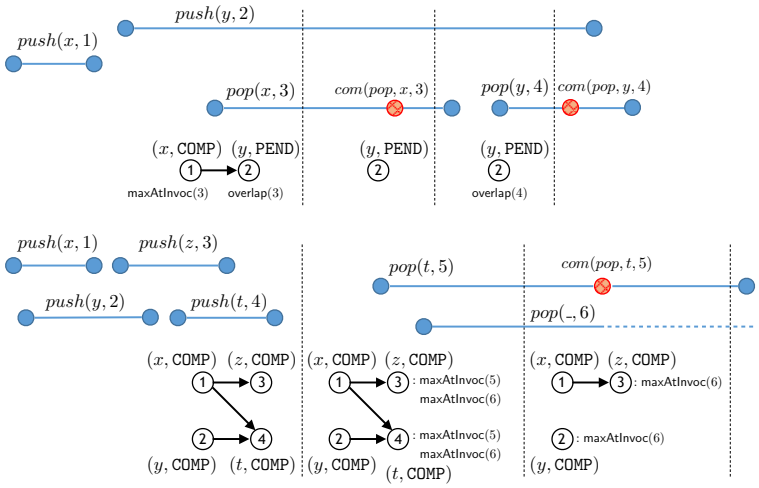

Fig. 6. Simulating stack histories with *AbsS*.

before its commit point. Each pop operation starts by taking a snapshot of the completed push operations which are maximal in the happens-before, more precisely, which don't happen before another completed push operation. Also, the library maintains the set of push operations overlapping with each pop operation. The commit point $com(pop, d, k)$ with $d \neq \texttt{EMPTY}$ is enabled if either d was added by one of the push operations in the initial snapshot, or by a push happening earlier when arguments of pushes from the initial snapshot have been removed, or by one of the push operations that overlaps with pop k. The commit point $com(pop, \texttt{EMPTY}, k)$ is enabled if all the values added by push operations happening before k have been removed. The effect of the commit points is explained below through examples.

Fig. 6 pictures two executions of *AbsS* for two extended histories (that include pop commit points). For readability, we give the state of *AbsS* only after several execution prefixes delimited at the right by a dotted line. We focus on pop operations – the effect of push calls and returns is similar to enqueue calls and returns in *AbsQ*. Let us first consider the history on the top part. The first state we give is reached after the call of pop with identifier 3. This shows the effect of a pop invocation: the greatest completed pushes according to the current happens-before (here, the push with identifier 1) are marked as $\mathsf{maxAtInvoc}(3)$, and the pending pushes are marked as $\mathsf{overlap}(3)$. As a side remark, any other push operation that starts after pop 3 would be also marked as $\mathsf{overlap}(3)$. The commit point $com(pop, x, 3)$ (pictured with a red circle) is enabled because x was added by a push marked as $\mathsf{maxAtInvoc}(3)$. The effect of the commit point is that push 1 is removed from the state (the execution on the bottom shows a more complicated case). For the second pop, the commit point $com(pop, y, 4)$ is enabled because y was added by a push marked as $\mathsf{overlap}(4)$. The execution on the bottom shows an example where the marking $\mathsf{maxAtInvoc}(k)$ for some pop k is updated at commit points. The pushes 3 and 4 are marked as $\mathsf{maxAtInvoc}(5)$ and $\mathsf{maxAtInvoc}(6)$ when the pops 5 and 6 start. Then, $com(pop, t, 5)$ is enabled since t was added by $push(t, 4)$ which is marked as $\mathsf{maxAtInvoc}(5)$. Besides removing $push(t, 4)$, the commit

```
function loc: ℕ → {inv,com,ret,⊥}
function arg, ret: ℕ → 𝕍
function present, pending: ℕ → 𝔹
function before: ℕ × ℕ → 𝔹
function maxAtInvoc: ℕ → ℘(ℕ)
function overlap: ℕ → ℘(ℕ)

rule inv(push,x,k):
  present(k) := true
  pending(k) := true
  arg(k) := x
  forall k1 with present(k1):
    if ¬pending(k1):
      before(k1,k) := true
  forall k1:
    overlap(k1) := overlap(k1)∪{k}

rule ret(push,k):
  pending(k) := false

rule ret(pop,y,k):
  assert ret(k) = y

rule inv(pop,k):
  forall k1 with present(k1):
    if pending(k1):
      overlap(k) := overlap(k) ∪ {k1}
    else:
      if ∀k2.present(k2)∧before(k1,k2)⇒pending(k2):
        maxAtInvoc(k) = maxAtInvoc(k) ∪ {k1}

rule com(pop,y,k):
  ret(k) := y
  if y = EMPTY:
    assert maxAtInvoc(k) = ∅
  else:
    let k1 := arg⁻¹(y)
    assert present(k1)
    assert k1 ∈ maxAtInvoc(k) ∪ overlap(k)
    present(k1) := false
    forall k2 with k1 ∈ maxAtInvoc(k2):
      maxAtInvoc(k2) := maxAtInvoc(k2) \ {k1}
      forall k3 with before_I(k3,k1):
        if ∀k4≠k1.before_I(k3,k4) ⇒ k4∈overlap(k2):
          maxAtInvoc(k2) := maxAtInvoc(k2) ∪ {k3}
```

Fig. 7. The *AbsS* implementation; each rule α`(_,k)` implicitly begins with **assert** `loc(k)`=α and ends with the appropriate `loc(k)`:=β; and `before`$_I$ denotes the transitive reduction of `before`.

point produces a state where a pop committing later, e.g., pop 6, can remove y which was added by a predecessor of $push(t,4)$ in the happens-before (y could become the top of the stack when t is removed). This history is valid because $push(y,2)$ can be linearized after $push(x,1)$ and $push(z,3)$. Thus, push 2, a predecessor of the push which is removed, is marked as maxAtInvoc(6). Push 1 which is also a predecessor of the removed push is not marked as maxAtInvoc(6) because it happens before another push, i.e., push 3, which is already marked as maxAtInvoc(6) (the value added by push 3 should be removed before the value added by push 1 could become the top of the stack).

The description of *AbsS* as an abstract state machine is given in Fig. 7. Compared to *AbsQ*, the state contains two more updatable functions maxAtInvoc and overlap. For each pop operation k, maxAtInvoc(k) records the set of completed push operations which were maximal in the happens-before (defined by before) when pop k was invoked, or happening earlier provided that the values of all the pushes happening later than one of these maximal ones and before pop k have been removed. Also, overlap(k) contains the push operations overlapping with a pop k. Initially, `loc(k) = inv` for all `k`, `present(k1) = pending(k1) = before(k1,k2) = false`, and `maxAtInvoc(k1) = overlap(k1) =` $\emptyset$, for all `k1, k2`. The rules for actions of push methods are similar to those for enqueues in *AbsQ*, except that every newly invoked push operation `k` is added to the set overlap(`k1`) for all pop operations `k1` (since `k` overlaps with all the currently pending pops). The rule `inv(pop,k)`, marking the invocation of a pop, sets maxAtInvoc(`k`) and overlap(`k`) as explained above. The rule `com(pop,EMPTY,k)` is enabled when the set maxAtInvoc(`k`) is empty (otherwise, there would be push operations happening before pop `k` which makes the return value `EMPTY` incorrect). Also, `com(pop,y,k)` with `y`$\neq$`EMPTY` is enabled when `y` was added by a push `k1` which belongs to maxAtInvoc(`k`) $\cup$ overlap(`k`). This rule may also update maxAtInvoc(`k2`) for

other pending pops k2. More precisely, whenever maxAtInvoc(k2) contains the push k1, the latter is replaced by the immediate predecessors of k1 (according to before) that are followed exclusively by pushes overlapping with k2.

The abstract state machine in Fig. 7 defines an LTS over the alphabet $C \cup R \cup Com(pop)$. Let $AbsS_0$ be the standard abstract implementation of a stack where elements are stored in a sequence, push and pop operations adding and removing an element from the beginning of the sequence in one atomic step, respectively. For $\mathbb{M} = \{push, pop\}$, the alphabet of $AbsS_0$ is $C \cup R \cup Lin$. The following result states that the library *AbsS* has exactly the same set of histories as $AbsS_0$.

Theorem 4. *AbsS is a refinement of $AbsS_0$ and vice-versa.*

A trace of a stack implementation is called *$Com(pop)$-complete* when every completed pop has a commit point, i.e., each return $ret(pop,d,k)$ is preceded by an action $com(pop,d,k)$. A stack implementation L over Σ, such that $C \cup R \cup Com(pop) \subseteq \Sigma$, is called *with fixed pop commit points* when every trace $\tau \in Tr(L)$ is $Com(pop)$-complete.

As a consequence of Theorem 2, $C \cup R \cup Com(pop)$-forward simulations are a sound and complete proof method for showing the correctness of a stack implementation with fixed pop commit points (up to the correctness of the commit points).

Corollary 2. *A stack L with fixed pop commit points is a $C \cup R \cup Com(pop)$-refinement of AbsS iff there is a $C \cup R \cup Com(pop)$-forward simulation from L to AbsS.*

Linearization points can also be seen as commit points and thus the following holds.

Corollary 3. *A stack implementation L with fixed pop linearization points where transition labels $lin(pop,d,k)$ are substituted with $com(pop,d,k)$ is a $C \cup R \cup Com(pop)$-refinement of $AbsS_0$ iff there is a $C \cup R \cup Com(pop)$-forward simulation from L to AbsS.*

5.3 A Correctness Proof For Time-Stamped Stack

We describe a forward simulation F_2 from *TSS* to *AbsS*, which is similar to the one from *HWQ* to *AbsQ* for the components of an *AbsS* state which exist also in *AbsQ* (i.e., different from maxAtInvoc and overlap).

Thus, a *TSS* state s is related by F_2 to *AbsS* states t where present(k) is true for every push operation k in s such that k has not yet added a node to pools or its node is still present in pools (i.e., the node created by the push has taken set to false). Also, pending(k) is true in t iff k is pending in s.

To describe the constraints on the order relation before and the sets maxAtInvoc and overlap in t, we consider the following notations: $\mathtt{ts}_s(k)$, resp., $\mathtt{TID}_s(k)$, denotes the timestamp of the node created by the push k in state s (the ts field of this node), resp., the id of the thread executing k. By an abuse of terminology, we call $\mathtt{ts}_s(k)$ the timestamp of k in state s. Also, $k \rightsquigarrow_s k'$ when intuitively, a traversal of pools would encounter the node created by k before the one created by k'. More precisely, $k \rightsquigarrow_s k'$ when $\mathtt{TID}_s(k) < \mathtt{TID}_s(k')$, or $\mathtt{TID}_s(k) = \mathtt{TID}_s(k')$ and the node created by k' is reachable from the one created by k in the list pointed to by $\mathtt{pools}[\mathtt{TID}_s(k)]$.

The order relation before satisfies the following: (1) pending pushes are maximal, (2) before is consistent with the order between node timestamps, i.e., $\mathtt{ts}_s(k) \leq \mathtt{ts}_s(k')$

implies $\neg\mathsf{before}(k',k)$, and (3) before includes the order between pushes executed in the same thread, i.e., $\mathtt{TID}_s(k) = \mathtt{TID}_s(k')$ and $\mathtt{ts}_s(k) < \mathtt{ts}_s(k')$ implies $\mathsf{before}(k,k')$.

The components maxAtInvoc and overlap satisfy the following constraints (their domain is the set of identifiers of pending pops):

***Frontiers*:** By the definition of *TSS*, a pending pop p in s could, in the future, remove the value added by a push k which is maximal (w.r.t. before) or a push k which is completed but followed only by pending pushes (in the order relation before). Therefore, for all pop operations p which are pending in s, we have that $k \in \mathsf{overlap}(p) \cup \mathsf{maxAtInvoc}(p)$, for every push k such that $\mathsf{present}(k) \wedge (\mathsf{pending}(k) \vee (\forall k'.\mathsf{present}(k') \wedge \mathsf{before}(k,k') \rightarrow \mathsf{pending}(k'))$.

***TraverseBefore*:** a pop p with $\mathtt{youngest}(p) \neq \mathtt{null}$ that reached the node `n` overlaps with every present push that created a node with a timestamp greater than $\mathtt{youngest}(p)\rightarrow\mathtt{ts}$ and which occurs in `pools` before the node n. Formally, if $\mathtt{youngest}_s(p) = \mathtt{n}_s(k) \neq \mathtt{null}$, $\mathtt{n}_s(p) = \mathtt{n}_s(k_1)$, $k_2 \rightsquigarrow_s k_1$, $\mathsf{present}(k_2)$, and $\mathtt{ts}_s(k_2) \geq \mathtt{ts}_s(k)$, then $k_2 \in \mathsf{overlap}(p)$, for each p,k_1,k_2.

***TraverseBeforeNull*:** a pop p with $\mathtt{youngest}(p) = \mathtt{null}$ overlaps with every push that created a node which occurs in `pools` before the node reached by p, i.e., $\mathtt{youngest}_s(p) = \mathtt{null}$, $\mathtt{n}_s(p) = \mathtt{n}_s(k_1)$, $k_2 \rightsquigarrow_s k_1$, and $\mathsf{present}(k_2)$ implies $k_2 \in \mathsf{overlap}(p)$, for each p,k_1,k_2.

***TraverseAfter*:** if the variable `youngest` of a pop p points to a node which is not taken, then this node was created by a push in $\mathsf{maxAtInvoc}(p) \cup \mathsf{overlap}(p)$ or the node currently reached by p is followed in `pools` by another node which was created by a push in $\mathsf{maxAtInvoc}(p) \cup \mathsf{overlap}(p)$. Formally, for each p,k_1,k_2, if $\mathtt{youngest}_s(p) = \mathtt{n}_s(k_1)$, $\mathtt{n}_s(k_1)\texttt{->taken} = \mathtt{false}$, and $\mathtt{n}_s(p) = \mathtt{n}_s(k_2)$, then one of the following holds:

- $k_1 \in \mathsf{maxAtInvoc}(p) \cup \mathsf{overlap}(p)$, or
- there exists a push k_3 in s such that $\mathsf{present}(k_3)$, $k_3 \in \mathsf{maxAtInvoc}(p) \cup \mathsf{overlap}(p)$, $\mathtt{ts}_s(k_3) > \mathtt{ts}_s(k_1)$, and either $k_2 \rightsquigarrow_s k_3$ or $\mathtt{n}_s(k_2) = \mathtt{n}_s(k_3)$ and p is at a control point before the assignment statement that changes the variable `youngest`.

The functions maxAtInvoc and overlap satisfy more constraints which can be seen as invariants of *AbsS*, e.g., $\mathsf{maxAtInvoc}(p)$ and $\mathsf{overlap}(p)$ do not contain predecessors of pushes from $\mathsf{maxAtInvoc}(p)$ (for each p,k_1,k_2, $\mathsf{before}(k_1,k_2)$ and $k_2 \in \mathsf{maxAtInvoc}(p)$ implies $k_1 \notin \mathsf{maxAtInvoc}(p) \cup \mathsf{overlap}(p)$). They can be found in [8].

Note that F_2 cannot satisfy the reverse of ***Frontiers***, i.e., every push in $\mathsf{overlap}(p) \cup \mathsf{maxAtInvoc}(p)$, for some p, is maximal or followed only by pending pushes (w.r.t., before). This is because the linearization points of pop operations are not fixed and they can occur anywhere in between their invocation and commit points. Hence, any push operation which was maximal or followed only by pending pushes in the happens-before in between the invocation and the commit can be removed by a pop. And such a push may no longer satisfy the same properties in the state s.

Based on the values stored in $youngest_s(p)$ and $n_s(p)$, for some pop p, the other three constraints identify other push operations that overlap with p, or they were followed only by pending pushes when p was invoked. ***TraverseBefore*** and ***TraverseBeforeNull*** state that pushes which add new nodes to the pools seen by p in the past, are overlapping with p. ***TraverseAfter*** states that either the push adding the current

youngest node $youngest_s(p)$ is in $\mathsf{overlap}_s(p) \cup \mathsf{maxAtInvoc}_s(p)$, or there is a node that p will visit in the future which is in $\mathsf{overlap}_s(p) \cup \mathsf{maxAtInvoc}_s(p)$.

The proof that F_2 is indeed a forward simulation from *TSS* to *AbsS* follows the same lines as the one given for the Herlihy&Wing Queue. It can be found in [8].

6 Related Work

Many techniques for linearizability verification, e.g., [28, 4, 27, 2], are based on forward simulation arguments, and typically only work for libraries where the linearization point of every invocation of a method m is fixed to a particular statement in the code of m. The works in [25, 9, 11, 29] deal with *external* linearization points where the action of an operation k can be the linearization point of a concurrently executing operation k'. We say that the linearization point of k' is external. This situation arises in read-only methods like the `contains` method of an optimistic set [23], libraries based on the elimination back-off scheme, e.g., [15], or flat combining [16, 13]. In these implementations, an operation can do an update on the shared state that becomes the linearization point of a concurrent read-only method (e.g., a `contains` returning `true` may be linearized when an `add` method adds a new value to the shared state) or an operation may update the data structure on behalf of other concurrently executing operations (whose updates are published in the shared state). In all these cases, every linearization point can still be associated syntactically to a statement in the code of a method and doesn't depend on operations executed in the future (unlike *HWQ* and *TSS*). However, identifying the set of operations for which such a statement is a linearization point can only be done by looking at the whole program state (the local states of all the active operations). This poses a problem in the context of compositional reasoning (where auxiliary variables are required), but still admits a forward simulation argument. For manual proofs, such implementations with external linearization points can still be defined as LTSs that produce *Lin*-complete traces and thus still fall in the class of implementations for which forward simulations are enough for proving refinement. These proof methods are not complete and they are not able to deal with implementations like *HWQ* or *TSS*.

There also exist linearizability proof techniques based on backward simulations or alternatively, prophecy variables, e.g., [26, 24, 20]. These works can deal with implementations where the linearization points are not fixed, but the proofs are conceptually more complex and less amenable to automation.

The works in [17, 6] propose reductions of linearizability to assertion checking where the idea is to define finite-state automata that recognize violations of concurrent queues and stacks. These automata are simple enough in the case of queues and there is a proof of *HWQ* based on this reduction [17]. However, in the case of stacks, the automata become much more complicated and we are not aware of a proof for an implementation such as *TSS* which is based on this reduction.

7 Acknowledgements

This work is supported in part by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No 678177).

Bibliography

[1] M. Abadi and L. Lamport. The existence of refinement mappings. *Theor. Comput. Sci.*, 82(2):253–284, 1991.

[2] P. A. Abdulla, F. Haziza, L. Holík, B. Jonsson, and A. Rezine. An integrated specification and verification technique for highly concurrent data structures. In *TACAS*, pages 324–338, 2013.

[3] R. Alur, K. L. McMillan, and D. Peled. Model-checking of correctness conditions for concurrent objects. *Inf. Comput.*, 160(1-2):167–188, 2000.

[4] D. Amit, N. Rinetzky, T. W. Reps, M. Sagiv, and E. Yahav. Comparison under abstraction for verifying linearizability. In *CAV '07*, volume 4590 of *LNCS*, pages 477–490, 2007.

[5] A. Bouajjani, M. Emmi, C. Enea, and J. Hamza. Verifying concurrent programs against sequential specifications. In *ESOP '13*, volume 7792 of *LNCS*, pages 290–309. Springer, 2013.

[6] A. Bouajjani, M. Emmi, C. Enea, and J. Hamza. 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*, pages 95–107, 2015.

[7] A. Bouajjani, M. Emmi, C. Enea, and J. Hamza. 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*, pages 651–662, 2015.

[8] A. Bouajjani, M. Emmi, C. Enea, and S. O. Mutluergil. Proving linearizability using forward simulations. *CoRR*, abs/1702.02705, 2017. URL http://arxiv.org/abs/1702.02705.

[9] J. Derrick, G. Schellhorn, and H. Wehrheim. *Verifying Linearisability with Potential Linearisation Points*, pages 323–337. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. ISBN 978-3-642-21437-0.

[10] M. Dodds, A. Haas, and C. M. Kirsch. A scalable, correct time-stamped stack. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 233–246, 2015.

[11] C. Dragoi, A. Gupta, and T. A. Henzinger. Automatic linearizability proofs of concurrent objects with cooperating updates. In *CAV '13*, volume 8044 of *LNCS*, pages 174–190. Springer.

[12] I. Filipovic, P. W. O'Hearn, N. Rinetzky, and H. Yang. Abstraction for concurrent objects. *Theor. Comput. Sci.*, 411(51-52):4379–4398, 2010.

[13] M. Gorelik and D. Hendler. Brief announcement: an asymmetric flat-combining based queue algorithm. In *ACM Symposium on Principles of Distributed Computing, PODC '13, Montreal, QC, Canada, July 22-24, 2013*, pages 319–321, 2013.

[14] J. Hamza. On the complexity of linearizability. In *Networked Systems - Third International Conference, NETYS 2015, Agadir, Morocco, May 13-15, 2015, Revised Selected Papers*, pages 308–321, 2015.

[15] D. Hendler, N. Shavit, and L. Yerushalmi. A scalable lock-free stack algorithm. In *SPAA 2004*, pages 206–215. ACM.

[16] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *SPAA 2010: Proceedings of the 22nd Annual ACM Symposium on Parallelism in Algorithms and Architectures, Thira, Santorini, Greece, June 13-15, 2010*, pages 355–364, 2010.

[17] T. A. Henzinger, A. Sezgin, and V. Vafeiadis. Aspect-oriented linearizability proofs. In *CONCUR*, pages 242–256, 2013.

[18] M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[19] M. Hoffman, O. Shalev, and N. Shavit. The baskets queue. In E. Tovar, P. Tsigas, and H. Fouchal, editors, *Principles of Distributed Systems, 11th International Conference, OPODIS 2007, Guadeloupe, French West Indies, December 17-20, 2007. Proceedings*, volume 4878 of *Lecture Notes in Computer Science*, pages 401–414. Springer, 2007. ISBN 978-3-540-77095-4.

[20] H. Liang and X. Feng. Modular verification of linearizability with non-fixed linearization points. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 459–470, 2013.

[21] N. A. Lynch and F. W. Vaandrager. Forward and backward simulations: I. untimed systems. *Inf. Comput.*, 121(2):214–233, 1995.

[22] A. Morrison and Y. Afek. Fast concurrent queues for x86 processors. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '13, Shenzhen, China, February 23-27, 2013*, pages 103–112, 2013.

[23] P. W. O'Hearn, N. Rinetzky, M. T. Vechev, E. Yahav, and G. Yorsh. Verifying linearizability with hindsight. In *PODC '10*, pages 85–94. ACM.

[24] G. Schellhorn, H. Wehrheim, and J. Derrick. How to prove algorithms linearisable. In *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, pages 243–259, 2012.

[25] V. Vafeiadis. Automatically proving linearizability. In *CAV '10*, volume 6174 of *LNCS*, pages 450–464.

[26] V. Vafeiadis. *Modular fine-grained concurrency verification*. PhD thesis, University of Cambridge, 2008.

[27] V. Vafeiadis. Shape-value abstraction for verifying linearizability. In *VMCAI '09: Proc. 10th Intl. Conf. on Verification, Model Checking, and Abstract Interpretation*, volume 5403 of *LNCS*, pages 335–348. Springer, 2009.

[28] V. Vafeiadis, M. Herlihy, T. Hoare, and M. Shapiro. Proving correctness of highly-concurrent linearisable objects. In *PPOPP '06*, pages 129–136. ACM.

[29] H. Zhu, G. Petri, and S. Jagannathan. Poling: SMT aided linearizability proofs. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II*, pages 3–19, 2015.