Article:
Schellhorn, G., Derrick, J. and Wehrheim, H. (2014) A Sound and Complete Proof Technique for Linearizability of Concurrent Data Structures. ACM Transactions on Computational Logic, 15 (4). 31. ISSN 1529-3785

https://doi.org/10.1145/2629496

A Sound and Complete Proof Technique for Linearizability of Concurrent Data Structures

GERHARD SCHELLHORN, University of Augsburg, Germany
JOHN DERRICK, University of Sheffield, UK
HEIKE WEHRHEIM, University of Paderborn, Germany

Efficient implementations of data structures such as queues, stacks or hash-tables allow for concurrent access by many processes at the same time. To increase concurrency, these algorithms often completely dispose with locking, or only lock small parts of the structure. *Linearizability* is the standard correctness criterion for such a scenario — where a concurrent object is linearizable if all of its operations appear to take effect instantaneously some time between their invocation and return.

The potential concurrent access to the shared data structure tremendously increases the complexity of the verification problem, and thus current proof techniques for showing linearizability are all tailored to specific types of data structures. In previous work we have shown how simulation-based proof conditions for linearizability can be used to verify a number of subtle concurrent algorithms. In this paper, we now show that conditions based on *backward* simulation can be used to show linearizability of *every* linearizable algorithm, i.e., we show that our proof technique is both sound *and* complete. We exemplify our approach by a linearizability proof of a concurrent queue, introduced in Herlihy and Wing's landmark paper on linearizability. Except for their manual proof, none of the numerous other approaches have successfully treated this queue.

Our approach is supported by a full mechanisation: both the linearizability proofs for case studies like the queue, and the proofs of soundness and completeness have been carried out with an interactive prover, which is KIV.

Categories and Subject Descriptors: D.2.4 [**Software Engineering**]: Software/Program Verification

General Terms: Algorithms,Verification

Additional Key Words and Phrases: Z, refinement, concurrent access, linearizability, non-atomic refinement, theorem proving, KIV

ACM Reference Format:
ACM Trans. Comput. Logic V, N, Article A (January YYYY), 35 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

1. INTRODUCTION

The advent of multi- and many-core processors will see an increased usage of concurrent data structures. These are implementations of data structures like queues, stacks or hash-tables which allow for concurrent access by many processes at the same time. Indeed, already libraries such as `java.util.concurrent` offer a vast number of such concurrent data structures. To increase concurrency, these algorithms often completely dispose with locking,

Gerhard Schellhorn, Universität Augsburg, Institut für Informatik, 86135 Augsburg, Germany, schellhorn@informatik.uni-augsburg.de
John Derrick, Department of Computing, University of Sheffield, Sheffield, UK, J.Derrick@dcs.shef.ac.uk
Heike Wehrheim, Universität Paderborn, Institut für Informatik, 33098 Paderborn, Germany, wehrheim@uni-paderborn.de

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

or only lock small parts of the structure. Instead of locking, fine-grained synchronization schemes based on atomic operations (e.g., Compare-And-Swap CAS or Load-Link/Store-Conditional LL/SC) are employed in order to allow for a high degree of concurrency.

Whereas correctness for their sequential counter-parts is trivial, it can be complex and subtle for these concurrent algorithms, particularly ones that exploit the potential for concurrency to the full. For example, their design inevitably leads to race conditions. In fact, the designers of such algorithms do not aim at race-free but at *linearizable* algorithms. Linearizability [Herlihy and Wing 1990] requires that fine-grained implementations of access operations (e.g., insertion, lookup or removal of an element) appear as though they take effect instantaneously at some point in time, thereby achieving the same effect as an atomic operation [Herlihy and Wing 1990]:

> Linearizability provides the illusion that each operation applied by concurrent processes takes effect instantaneously at some point between its invocation and its response.

Since the original approach based on *possibilities*, a number of other approaches to prove linearizability have appeared, and a number of algorithms have been shown to be linearizable [Doherty et al. 2004; Colvin et al. 2005; Abrial and Cansell 2005; Hesselink 2007; Vafeiadis et al. 2006; Amit et al. 2007; Calcagno et al. 2007; Derrick et al. 2008; 2011b]. Some of the proofs of correctness are manual, whereas others are partly supported by theorem provers (for instance with PVS) or fully automatic. The latter, however, either require user-annotations to the algorithm or – in the case of model checkers – only prove correctness for a limited number of parallel processes.

The proof techniques vary as well, and range from using shape analysis or separation logic to rely-guarantee reasoning and simulation-based methods. The simulation-based methods show that an abstraction (or simulation or refinement) relation exists between the abstract specification of the data structure and its concurrent implementation.

Whilst great progress has been made in the state-of-the-art in this area, a number of weaknesses remain. For example, apart from our own [Derrick et al. 2008], all current approaches only argue informally that their proof technique actually implies the original linearizability criterion of [Herlihy and Wing 1990]. Rather, they have focused on providing an efficient and practically applicable proof technique. We, however, have aimed to ensure that we always have a mechanized proof of soundness of our method in addition to any individual proof of correctness for a specific algorithm.

In addition, inspecting the current approaches, one finds that a number of techniques (including our own so far) get adapted every time a new type of algorithm is treated. Every new "trick" designers build into their algorithms to increase performance (e.g., like a mutual push and pop elimination for stacks, or lazy techniques) requires an extension of the verification approach. This is the case because the verification approaches are usually kept as simple as possible to ease their application, and in particular to allow for a high degree of automatization. Still, our motivation was to present a proof technique which can be used to prove linearizability of *every* linearizable algorithm: we have derived a proof method that is *sound and complete* for linearizability.

The approach is again a simulation-based method, this time based on *backward simulations* — a technique usually applied for certain classes of data refinement. What is unusual with the method is that in data refinement, two types of simulation (forward *and* backward) are necessary in order to give completeness [He Jifeng et al. 1986; de Roever and Engelhardt 1998; Derrick and Boiten 2001]. However, for linearizability, we show that backward simulation alone is already sufficient (and furthermore, that backward simulations are sound with respect to linearizability). More precisely, we show that a fine-grained implementation is linearizable with respect to an abstract atomic specification of the data structure if and only if there is a backward simulation between the specification and the implementation. The use of simulations for showing linearizability is not new; however, current refinement-based approaches (e.g., [Doherty et al. 2004; Doherty and Moir 2009; Colvin and Groves 2005]) are based on both backward *and* forward simulations. Note that our completeness result does not imply a direct method for deriving a backward simulation for an existing linearizable data structure; it just states the existence of such a simulation. This is in line with other completeness results, e.g., for data refinement or for specific proof calculi: completeness only states the existence of a proof, never a way of deriving this proof.

We nevertheless exemplify our approach to using just backward simulations by verifying the queue implementation of Herlihy and Wing [Herlihy and Wing 1990]. None of the current approaches to linearizability have treated this algorithm; and it is also not clear whether the many approaches tailored towards heap usage (like separation logic or shape analysis based techniques) can successfully verify the queue, as the complexity in the interaction between concurrent processes in the queue is not due to a shared heap. There is no heap involved at all, rather, the underlying store is an unbounded array. Along with this queue example we also show how to systematically construct the backward simulations needed in the linearizability proofs. Although the complete methodology determines the most general backward simulation to use, in practice, it is often more convenient to use a smaller relation. We do this in the queue verification, and to this end, we provide a number of guidelines for deriving backward simulations in general.

Last, but not least, to fit our desire for a provenly correct methodology, we have a complete mechanization of our approach. It is complete in the sense that we both carry out the backward simulation proofs for our examples (here, the queue) with an interactive prover (which is KIV [Reif et al. 1998]), *and* have verified within KIV that the general soundness and completeness proof of our technique is correct. Proofs for the general theory as well as for the case study can be found online at [KIV 2011].

The structure of the paper is as follows. The next section introduces our running example, the Herlihy and Wing queue. We use Z to formalize the pseudo-code of the algorithm, which will subsequently be verified in KIV. In section 3 we formally define linearizability of an abstract and a concurrent data type, and define refinement and simulations between them. The proof methodology is then derived in section 4. This is achieved by augmenting the original data types with history information and using a particular type of finalization, and showing that the abstract and concurrent data type are linearizable if and only if the augmented concurrent data type is a backward simulation of the augmented abstract data type. The proof uses a particular intermediate layer in its construction, reminiscent of the definition of possibilities in [Herlihy and Wing 1990].

We return to the queue of Herlihy and Wing in section 5 to mechanically verify a proof of its linearizability using a specific backward simulation.

For algorithms with complex linearization points, such as the queue considered here, global backward simulations are needed, since by their nature such algorithms have non-local behavior. However, for many simpler case studies it has been informally argued that simpler, thread-local proof obligations are sufficient. Section 7 sketches, how the thread-local proof obligations we defined in [Derrick et al. 2011b] can be derived from the general backward simulation.

Finally, in sections 8 and 9 we discuss related work and some conclusions. This paper is an extended version of [Schellhorn et al. 2012] in which we present the underlying theory in full detail, the most important aspects of the KIV proof and add guidelines for the derivation of backward simulations.

2. HERLIHY-WING QUEUE

The running example we use throughout this paper is the concurrent implementation of an abstract queue, as specified by Herlihy and Wing in their original paper [Herlihy and

Wing 1990]. We will mix the informal description with a formalization in the specification language Z [Woodcock and Davies 1996], a state-based specification formalism which allows the specification of data types by defining their state (variables), initial values and operations, to describe an abstract queue and its concurrent implementation. The complete system then consists of a number of processes, each capable of executing its queue operations on the shared data structure. The key question we are interested in is whether the concurrent implementation is *correct* with respect to the abstract specification – even though the atomic steps of the concrete can be interleaved in a manner not feasible in the abstract specification.

As usual, the queue is a data structure with two operations: *enqueue* appends new elements to the end of the queue and *dequeue* removes elements from the front of the queue. We have one *enqueue* and *dequeue* for each process $p \in P$. Abstractly, we can specify the queue as follows. The state, initialization and operations are given as *schemas*, consisting of variable declarations plus predicates. We assume a given type T for queue elements.

$AState$
$q : \text{seq}\ T$

$AInit$
$AState$
$q = \langle\,\rangle$

$AEnq_p$
$\Delta AState$
$el? : T$
$q' = q \frown \langle el? \rangle$

$ADeq_p$
$\Delta AState$
$el! : T$
$q = \langle el! \rangle \frown q'$

The first two schemas in the specification fix the state, $AState$, of the data structure queue (a sequence of elements), and its initial value (the empty list given via its initialization $AInit$). As usual in Z, a sequence $q = \langle a_1, \dots a_n \rangle$ of length n (written $\#q = n$) is a function from the interval from 1 to n (written $1..n$) to elements of T. Operator $q1 \frown q2$ concatenates two sequences, an element is selected with $q(k)$ for $k \in 1..n$. The two operations *enqueue* and *dequeue* for all the processes are written as $AEnq_p$ and $ADeq_p$ (A for abstract specification and p for the process executing it). In such a specification the primed variables refer to the value of variables in the after-state. Unprimed as well as primed variables of a particular state schema are introduced into operation schemas by the Δ-notation. Input and output variables are decorated by ? and !. In general, we can model all such data structures as abstract data types $ADT = (AState, AInit, (AOp_{p,i})_{p \in P, i \in I})$, where the set I is used for enumerating the abstract operations. Each operation $AOp_{p,i}$ is defined over the state space $AState$ and can additionally specify inputs and outputs. Inputs are denoted here by $in?$ and outputs by $out!$, with types IN_i and OUT_i (dependent on the operation i, but not on the process p executing it) respectively.

The queue is implemented by an array AR of unbounded length. All (used) slots of the array are initialized with a special value $null \notin T$, signaling 'no element present'. A back pointer *back* into the array stores the current upper end of the array where elements are enqueued. Dequeues operate on the lower end of the array. An *enq* operation thus simply gets a local copy of *back*, increments *back* (these two steps are executed atomically) and then stores the element to be enqueued in the array. We can describe the operations in pseudo-code as follows, where we annotate each line with a line number ($E1$ etc):

```
E0   enq(lv : T)
E1   (k,back) := (back, back+1); /* increment */
E2   AR[k]:= lv;                 /* store     */
E3   return
```

```
D0    deq(): T
D1    lback := back; k:=0; lv := null
D2    if k < lback goto D3 else goto D1
D3    (lv, AR[k])  := (AR[k], lv);     /* swap */
D4    if lv != null then goto D6 else goto D5
D5    k := k + 1; goto D2
D6    return(lv)
```

The *deq* operation proceeds in several steps: first, it gets a local copy of *back* and initializes a counter k and a local variable, *lv*, which is used to store the dequeued element. It then walks through the array trying to find an element to be dequeued. Steps $D2$ and $D5$ in the code above constitute a loop consecutively visiting the array elements.

At every position k visited in the loop the array contents $AR[k]$ is swapped with variable *lv*. If the dequeue finds a proper non-empty element this way ($lv \neq null$), this will be returned, otherwise searching is continued. In the case where no element can be found in the entire array, *deq* restarts the search. Note that if no *enq* operations occur, *deq* will thus run forever.

As we said, the complete system consists of a number of processes, each capable of executing its queue operations on a shared data structure. For the concrete implementation therefore these two algorithms can be executed concurrently by any number of processes — where the individual steps in the operations are atomic. So, for instance, we could have a process p executing $E1$ and then a process q executing $D1$ and (one of the branches of) $D2$, then p continuing with $E2$, a third process r starting yet another *deq* with step $D1$ and so on. Every interleaving of steps is possible. Formally, we can model the complete system (not just for a single process) as a data type with the following state.

CState

$AR : \mathbb{N} \rightarrow T \cup \{null\}$
$back : \mathbb{N}$
$kf : P \rightarrow \mathbb{N}$
$lbackf : P \rightarrow \mathbb{N}$
$lvf : P \rightarrow T \cup \{null\}$
$pcf : P \rightarrow PC$

CInit

$CState$

$\forall\, i : \mathbb{N} \bullet AR(i) = null$
$back = 0$

In addition to AR and *back* (explained above), the state contains local variables for every process from P, thus we have functions from P to the domain of the local variables. In addition, every process p has a variable $pcf(p)$ to denote the program counter. The program counter can take values from $PC = \{N, E1, E2, E3, D1, D2, D3, D4, D5, D6\}$, N denoting the idle state of a process (the idle state needs to be the same state for each operation so we use N as opposed to $E0$ and $D0$ here).

To model the operations, we introduce an operation denoting the invocation of an enqueue ($enq0$) or dequeue ($deq0$) (i.e., statements $E0$ and $D0$), then we have one operation for every line in the program (except for *if-statements* which are split into a true (e.g., $deq2t$) and a false ($deq2f$) case). Each operation then corresponds to the granularity of atomicity in the concurrent implementation.

Operation names are indexed by the process executing them. Thus in total we have operations $enq0_p$, $enq1_p$, $enq2_p$, $enq3_p$, $deq0_p$, $deq1_p$, $deq2t_p$, $deq2f_p$, $deq3_p$, $deq4t_p$, $deq4f_p$, $deq5_p$ and $deq6_p$ for every $p \in P$. Below we see a formal definition of some of the operations.

Z Notation: We assume the reader is familiar with the basics of the Z notation and its use of schemas. In fact, in our use of the schema notation we take the Object-Z approach of just specifying the changed values, and apply this at a functional level as well. This simply makes the specifications much more readable (otherwise we would have to add a lot of predicates of the form $back' = back$ etc.) and has no semantic consequence. Formula $pcf'(p) = E1$ denotes the expansion $pcf' = pcf \oplus \{p \mapsto E1\}$ that overwrites $pcf(p)$ with $E1$, and similarly for the other variables and functions in the schemas. The ordering of statements within the programs for the operations are ensured by the use of program counters and appropriate preconditions for operations.

$enq0_p$
$\Delta CState$
$lv? : T$
$pcf(p) = N \wedge pcf'(p) = E1$
$lvf'(p) = lv?$

$enq1_p$
$\Delta CState$
$pcf(p) = E1 \wedge pcf'(p) = E2$
$kf'(p) = back \wedge back' = back + 1$

$deq2t_p$
$\Delta CState$
$kf(p) < lbackf(p)$
$pcf(p) = D2 \wedge pcf'(p) = D3$

$deq6_p$
$\Delta CState$
$el! : T$
$pcf(p) = D6 \wedge pcf'(p) = N$
$el! = lvf(p)$

Formally, this defines our concrete data type $CDT = (CState, CInit, (COp_j)_{j \in J})$. Our objective is to show that the concrete data type is linearizable with respect to the abstract data type, i.e., all runs of the concrete data type with whatever kind of interleaving of operations of concurrent processes resemble correct queue operations. (We will formalize the definition itself later.)

To illustrate the issue we first take a look at one run to demonstrate the interplay between the operations of different processes. So imagine the following processes executing their individual concrete operations on the shared data type. Here, the steps of processes are given and (relevant parts of) the resulting state.

(1) Process 3 starts an enqueue for the element a ($E0$) and executes $E1$:
$kf(3) = 0, pcf(3) = E2, back = 1$ $AR = [-, -, -, \ldots]$
Sequence of operations so far: $\langle enq0_3, enq1_3 \rangle$
(2) Process 1 starts an enqueue for the element b and executes $E1$:
$kf(1) = 1, pcf(1) = E2, back = 2$ $AR = [-, -, -, \ldots]$
Sequence of operations so far: $\langle enq0_3, enq1_3, enq0_1, enq1_1 \rangle$
(3) Process 2 starts a dequeue and executes $D0$ and $D1$:
$kf(2) = 0, pcf(2) = D2, lbackf(2) = 2, lvf(2) = null$ $AR = [-, -, -, \ldots]$
Sequence of operations so far: $\langle enq0_3, enq1_3, enq0_1, enq1_1, deq0_2, deq1_2 \rangle$
(4) Process 4 starts an enqueue for the element c and executes $E0, E1, E2$:
$kf(4) = 2, pcf(4) = E3, back = 3$ $AR = [-, -, c, -, \ldots]$
Sequence of operations (continued): $\langle \ldots, enq0_4, enq1_4, enq2_4 \rangle$
(5) Process 2 continues its dequeue with $D2, D3, D4, D5, D2$:
$kf(2) = 1, pcf(2) = D3, lbackf(2) = 2, back = 3$ $AR = [-, -, c, -, \ldots]$
(6) Process 3 finishes its enqueue executing $E2$ and $E3$:
$pcf(3) = N$ $AR = [a, -, c, -, \ldots]$
(7) Process 4 finishes its enqueue executing $E3$:
$pcf(4) = N$ $AR = [a, -, c, -, \ldots]$
(8) Process 2 finishes the dequeue and returns c:
$pcf(2) = N, kf(2) = 2, lbackf(2) = 2, back = 3$ $AR = [a, -, -, -, \ldots]$
(9) Process 1 finishes the enqueue:
$pcf(1) = N$ $AR = [a, b, -, -, \ldots]$

The question of correctness is as follows: does this concrete run of operations (and its visible effect on the shared data structure) in some sense correspond to a run of the abstract specification?

In fact, it does in this case, specifically the above concrete run has (for instance) the following matching run of the abstract data type:

$$\langle AEnq_4(c), ADeq_2(c), AEnq_3(a), AEnq_1(b)\rangle$$

But notice that this correct abstract run is not the initially most obvious. Specifically, one would be tempted to look at the ordering of the invocations of the operations: $\langle AEnq_3, AEnq_1, ADeq_2, AEnq_4\rangle$, or failing that, the returns (i.e., responses) of the operations: $\langle AEnq_3, AEnq_4, ADeq_2, AEnq_1\rangle$. Neither of these is the correct abstract sequence necessary to give the right queue: in the first case the dequeue should return a, and in the second case as well, which it however does not.

Rather, instead of the start or the end, it is the point in time where the *effect* of the operation on the abstract queue becomes visible to other operations (this is known as the *linearizability* point, or LP) that determines the corresponding abstract order. The effect of enqueue of c becomes visible first even though it puts the element into position 2 of the array, not 0. This is because it is the first position which the dequeue observes to be non-empty as it has already gone past position 0 with its local array index k when the element a is placed in position 0.

At this moment one would be tempted to exactly find the LP of each operation, e.g., it might be $E2$ for enqueue and $D5$ for dequeue. Such an approach of finding fixed linearization points in the concurrent algorithm works for some non-locking algorithms such as Treiber's stack [Treiber 1986] or Michael and Scott's queue [Michael and Scott 1996]. However, for other more complex algorithms the point where the effects become visible changes from run to run according to the contents of the input/output or data structure. Examples of algorithms that are like this include the lazy set algorithm [Heller et al. 2005]. However, the behavior of the above is even more subtle — the LP in one operation *depends on what other operations have already started* — i.e., depends on the whole global history. To see this note that if the dequeue has not already gone past position 1 when process 1 enqueues b, the dequeue would return b rather than c. Thus, it is not only the ordering of enqueue operations of different processes which determines their abstract order, but it is also the progression of dequeues (of which we can of course also have several ones).

This intricate interplay of enqueues and dequeues makes this algorithm the hardest type for which to verify linearizability. Indeed, at first glance the hopes of finding some *local* proof obligations (as opposed to reasoning about global histories) would seem remote, let alone finding a complete methodology. However, this is what the following sections seek to explore. We begin with the basic definitions.

3. BACKGROUND

Our proof technique is grounded in the theory of refinement and simulations. However, before we start with the basic terminology for these, we want to formally define the notion of linearizability. In this, we essentially follow Herlihy and Wing's formalization.

3.1. Histories and Linearizability

Linearizability is a notion of correctness relating two specifications: a sequential abstract specification and a concurrent implementation. Here, these are both given as data types consisting of state space, the initialization and a number of operations executed by some process, i.e., like in the example we have $ADT = (AState, AInit, (AOp_{p,i})_{p\in P, i\in I})$ where $AInit \subseteq AState$, and $AOp_{p,i} \subseteq IN_i \times AState \times AState \times OUT_i$. The concurrent implementation is given as $CDT = (CState, CInit, (COp_{p,j})_{j\in J, p\in P})$ where $CInit \subseteq CState$. We assume a function $abs : J \rightarrow I$ that tells which concrete operation is a step of the implementation of which abstract one. We will place some restrictions on abs below, e.g., it will be an 1-to-n mapping between abstract and concrete. The concrete operations can be partitioned into three classes. Invoking operations ($deq0_p$ and $enq0_p$ in the example) have input of type $IN_{abs(j)}$. Returning operations ($deq6_p$ and $enq3_p$) have output of type $OUT_{abs(j)}$, all others have neither input nor output, i.e., these operations are just relations over $CState$.

The last example already discussed the issue of finding an appropriate run of the abstract data type for every run of the concrete implementation. Here, we will make the notion of "appropriate" or "matching" abstract run more precise.

Formally, linearizability can be defined by comparing the *histories* formed by runs of the abstract and concrete data type. Histories are finite sequences of events. Events can be invocations and returns of particular operations by particular processes from the set P.

$$\begin{aligned}&Event ::= inv\langle\langle P \times I \times IN_i \rangle\rangle \mid ret\langle\langle P \times I \times OUT_i \rangle\rangle\\ &History = \operatorname{seq} Events\end{aligned}$$

Events are defined using Z notation for free data types. Typical elements of the type are $inv(p, i, in)$ and $ret(p, i, out)$, where p is a process, i is an operation index, and $in \in IN_i$ and $out \in OUT_i$ are elements of the input resp. output type of $AOp_{p,i}$. Setting $I = \{enq, deq\}$, a possible history of our queue implementation is

$$\begin{aligned}h = \langle &inv(3, enq, a), inv(1, enq, b), inv(2, deq,), inv(4, enq, c), ret(3, enq,),\\ &ret(4, enq,), ret(2, deq, c), ret(1, enq,)\rangle\end{aligned}$$

This is the history corresponding to the above run detailed in the previous section, where, for example, $ret(3, enq,)$ denotes the return of the enqueue operation undertaken by process 3. The return event of enqueue has no output, thus the last argument is empty. The corresponding history of the abstract queue is

$$\begin{aligned}hs = \langle &inv(4, enq, c), ret(4, enq,), inv(2, deq,), ret(2, deq, c),\\ &inv(3, enq, a), ret(3, enq,), inv(1, enq, b), ret(1, enq,)\rangle\end{aligned}$$

Of course, there is no interleaving at the abstract level, thus $ret(1, enq,)$ immediately follows its invocation $inv(1, enq, b)$, that is histories of the abstract specification (like this one) will always be *sequential* (definition coming up). These histories are in a sense an abstraction of the concrete run (with just invocations and returns), but not as abstract as the initial specification which did not even differentiate between invocation and return. We will use them to compare our abstract and concrete runs in a fashion detailed below. First, some basic definitions.

We use predicates $inv?(e)$ and $ret?(e)$ to check whether an event $e \in Event$ is an invocation or a return, and we let $Ret!$ be the set of return events. We let $e.p \in P$ be the process executing the event e and $e.i \in I$ the index of the abstract operation to which the event belongs.

Not all sequences of events are correct histories of a data type. Thus we need the idea of a legal (concurrent) history: a legal history consists of matching pairs of invocation and return events plus some pending invocations, where an operation has started but not yet finished[1]

Definition 3.1 (*Legal histories*). Let $h : \operatorname{seq} Event$ be a sequence of events. Two positions m, n in h form a *matching pair*, denoted $mp(m, n, h)$ if

$$\begin{aligned}&0 < m < n \leq \#h \wedge inv?(h(m)) \wedge ret?(h(m)) \wedge h(m).p = h(n).p \wedge h(m).i = h(n).i \wedge\\ &\forall\, k : \mathbb{N} \bullet m < k < n \Rightarrow h(k).p \neq h(m).p\end{aligned}$$

[1]The definition of legal histories in [Herlihy and Wing 1990] (there called well-defined) requires all projections of the history to one process to be sequential. We prefer our equivalent definition, since many proofs must reason about the properties of matching pairs and pending invokes in legal histories.

A position $n : 1..\#h$ in h is a *pending invocation*, denoted $pi(n,h)$, if

$$inv?(h(n)) \wedge \forall\, m \bullet n < m \leq \#h \Rightarrow h(m).p \neq h(n).p$$

The set of all pending invokes $h(n)$ with $pi(n,h)$ is denoted as $pi(h)$. h is *legal*, denoted $legal(h)$, if

$$\begin{array}{l} \forall\, n : 1..\#h \bullet \textbf{if } inv?(h(n)) \textbf{ then } pi(n,h) \vee \exists\, m : 1..\#h \bullet mp(n,m,h) \\ \qquad\qquad\qquad\qquad\qquad\;\; \textbf{else } \exists\, m : 1..\#h \bullet mp(m,n,h) \end{array}$$

□

Both of the above given histories are legal. In the history h we, for instance, have matching pairs 1 and 5, $mp(1,5,h)$, and no pending invocations (all invocations are followed by a matching return).

Later, we will formally define how to construct histories out of a data type. For the moment note that history hs as well as all other histories of the abstract data type are *sequential*. This is formalized in the following predicate:

$$sequ(h) == \exists\, n \bullet \#h = 2n \wedge \forall\, m : 1..n \bullet mp(m+m-1, m+m)$$

A (complete[2]) sequential history is a sequence of matching pairs. For a legal history, function $complete(h)$ defined as

$$\begin{array}{l} complete(\langle\rangle) = \langle\rangle \\ complete(\langle e\rangle \frown h) = \textbf{if } pi(1, \langle e\rangle \frown h) \textbf{ then } complete(h) \textbf{ else } \langle e\rangle \frown complete(h) \end{array}$$

removes all pending invocations from a legal history h. To determine, whether a concurrent history h is linearizable it is compared with abstract sequential histories, to see whether one can be found, that matches. First of all, h might need to be extended by some (but not necessarily all) returns $h_0 \in \operatorname{seq} Ret!$ that match the pending invocations. This is the case when h contains operations where the effect has already taken place, though they have not returned. An example for this is the operation $D3$ in the dequeue: once the swap occurs on a non-empty array element, the effect of the operation has taken place although it has not returned yet. This results in a history $h \frown h_0$ which is now compared to a sequential history hs according to two conditions:

L1. when projected onto processes, $complete(h \frown h_0)$ and hs have to be equivalent, and
L2. the ordering of operation executions in h needs to be preserved in hs.

Here, two operations are *ordered* if the second one starts after the first one has returned. We rephrase this in a more formal way using a bijection f (denoted $\rightarrowtail\!\!\!\!\twoheadrightarrow$ using Z notation) between h and hs so that it can be used in our theorem prover:

Definition 3.2 (*Linearizable histories*). Given two histories h, hs, we define h to be in *lin*-relation with hs, denoted $lin(h, hs)$, if

$$\begin{array}{l} \exists\, f : 1..\#h \rightarrowtail\!\!\!\!\twoheadrightarrow 1..\#hs \bullet \\ \qquad legal(h) \wedge sequ(hs) \wedge \forall\, n : 1..\#h \bullet h(n) = hs(f(n)) \\ \quad \wedge \forall\, m, n : 1..\#h \bullet mp(m,n,h) \Rightarrow f(n) = f(m) + 1 \\ \quad \wedge \forall\, m, n, m', n' : 1..\#h \bullet mp(m,n,h) \wedge n < m' \wedge mp(m',n',h) \Rightarrow f(n) < f(m') \end{array}$$

A (concrete) history h is *linearizable* with respect to some sequential (abstract) history hs, denoted $linearizable(h, hs)$, if

$$\exists\, h_0 : \operatorname{seq} Ret! \bullet legal(h \frown h_0) \wedge lin(complete(h \frown h_0), hs)$$

[2][Herlihy and Wing 1990] allow a final pending invocation in sequential histories. For the definition of linearizability only complete sequential histories are needed.

A concrete data type CDT is *linearizable* with respect to an abstract data type ADT if every history of CDT is linearizable with respect to some history of ADT. □

Example 1: The history h from our running example is linearizable with respect to hs. □

Example 2: An example of a non-linearizable concrete history is

$$h_1 = \langle inv(1, enq, a), ret(1, enq,), inv(2, enq, b), inv(3, deq,), ret(3, deq, b)\rangle$$

For it to be linearizable we would have to find a sequential history hs_1 which keeps the ordering of operations (condition L2), i.e., in which the enqueue of a comes before the other two operations, and in which the dequeue still returns a b (condition L1). Neither the placement of the enqueue of b before nor after the dequeue would give us a valid abstract history. □

3.2. Data refinement and simulations

The central aim of work on linearizability is to derive proof methods to verify that a particular concurrent algorithm is linearizable. As we have seen, linearizability is a global condition defined over histories, so we are particularly interested in proof obligations that are *local* (in some sense, e.g., in terms of verifying conditions on an operation per operation basis) as opposed to global so that we can avoid reasoning over all histories.

Indeed, in our previous work on proving linearizability [Derrick et al. 2008; 2011a] we have derived proof obligations that examine one step at a time instead of full runs, using the ideas of *data refinement* between the two data types, specifically the obligations are all based on the idea of showing *simulations* between the data types. That is, we have exploited the fact that simulations are step-local obligations that can verify the global refinement property. The essence of our proof method is thus to derive *local proof obligations* such that local proof obligations between ADT and CDT imply CDT to be a certain sort of refinement of ADT which in turn can be shown to imply CDT is linearizable wrt ADT.

Using these methods we have been able to verify some of the standard benchmark algorithms for work in this area. For example, in [Derrick et al. 2008] we show how to verify a lock-free implementation of a stack. However, the proof obligations were not *complete*, that is not every case of linearizability could be verified using the methodology, and to cover more complex or subtle algorithms, the proof obligations had to be adapted to tackle all new aspects. For example, the lazy set implementation [Heller et al. 2005] tackled in [Vafeiadis et al. 2006] and [Colvin et al. 2006] has linearization points set by processes other than the one executing the operation, and we needed to use more general simulation conditions to treat this aspect.

For all of our proof obligations we have a mechanized proof of *soundness*, i.e., we have formally shown that the validity of the proof obligations implies linearizability. Here, we are now also interested in *completeness* of proof obligations: we aim to derive a proof technique which is general enough to show linearizability for *every* linearizable data type and algorithm. Of course, soundness needs to be kept.

The technique which we develop next is based on the idea of data refinement and simulations as was the original method. Essentially, we show that given an abstract and a concrete data type ADT and CDT (representing an abstract view and concurrent implementation, respectively) we can construct a particular simulation relation between ADT and CDT if and only if CDT is linearizable with respect to ADT. This opens up a way of proving linearizability for arbitrarily complex algorithms. Of course, there is no hope in keeping full locality — and simulations give us the best sense of locality that we can preserve — since we now are deriving a method general enough for all linearizable algorithms, even ones where an operation is linearized by that of another process.

Because our completeness proof and proof obligations for linearizability are based on simulations, we briefly discuss data refinement and the simulation methodology before we explain the approach further.

Simulations are used to show a refinement relationship between data structures. The purpose of data refinement [de Roever and Engelhardt 1998; Derrick and Boiten 2001] is to support a formal model-based design by giving a correctness criterion for changes made on the data representation or the operations of a data type. A data refinement relationship between two data types A and C guarantees *substitutability* of one data type by another: when invoking the same operations on C instead of on A, the outcome (i.e., observation) is one which has to be possible for A as well. The concept of "outcome" is formalized by a specific additional *finalization* operation, which determines what is observable at the end of every program. Normally, finalizations return the outputs that have been observed during execution of the program, and are thus not stated explicitly. However, in our case, we will use finalizations which return histories, i.e., $\mathit{Fin} \subseteq \mathit{State} \times \mathit{History}$, as the histories are the observations we are interested in. In its basic form we thus have a definition of data refinement as follows, where it is assumed that we have exactly one concrete operation for every abstract operation.

Definition 3.3 (*Data Type*). A data type $D = (\mathit{State}, \mathit{Obs}, \mathit{Init}, (\mathit{Op}_i)_{i \in I}, \mathit{Fin})$ consists of a set State of states, a set Obs of observations[3], an initialization relation $\mathit{Init} \subseteq \mathit{Obs} \times \mathit{State}$, a set of operations $\mathit{Op}_i \subseteq \mathit{State} \times \mathit{State}$ for $i \in I$, and a finalization relation $\mathit{Fin} \subseteq \mathit{State} \times \mathit{Obs}$. A *program* Prg is a sequence of operations, and running a program $\mathit{Prg} = j_1 \dots j_n$ creates the execution (a relation $\subseteq \mathit{Obs} \times \mathit{Obs}$)

$$\mathit{Prg}(C) \mathrel{\widehat{=}} \mathit{Init} \fatsemi \mathit{Op}_{j_1} \fatsemi \dots \fatsemi \mathit{COp}_{j_n} \fatsemi \mathit{CFin} \qquad \square$$

In the following we set $\mathit{Op} \mathrel{\widehat{=}} \bigcup_{i \in I} \mathit{Op}_i$ and define Op^* to be the reflexive, transitive closure of this relation.

Definition 3.4 (*Data refinement*). Let $A = (\mathit{AState}, \mathit{Obs}, \mathit{AInit}, (\mathit{AOp}_i)_{i \in I}, \mathit{AFin})$ and $C = (\mathit{CState}, \mathit{Obs}, \mathit{CInit}, (\mathit{COp}_i)_{i \in I}, \mathit{CFin})$ be two data types with the same observations. Then C is a *data refinement* of A, normally denoted $A \sqsubseteq C$, if for all programs Prg, $\mathit{Prg}(C) \subseteq \mathit{Prg}(A)$ holds. $\square$

The definition assumes that initialization, operations and finalization are all relations (and $\fatsemi$ is relational composition), the operations relate two states of the data types, and finalizations relate states of data types with some observation set used for comparison. For a more detailed account of data refinement see [Derrick and Boiten 2001].

In our setting, however, things are a bit different. For one thing, we have one abstract operation implemented by a whole number of concrete operations: e.g., an atomic enqueue is split into several smaller operations in the concrete implementation, and the function *abs* gives us the correspondence. We thus have a 1-to-n relationship here between abstract and concrete operations.

Conversely, we might also have the case that one concrete step in the implementation linearizes several operations (an n-to-1 relationship), as we shall see below for instance the Herlihy-Wing queue.

As a consequence, we cannot have the same program (Prg in the definition above) execute on concrete and abstract data type, but need a very relaxed form of refinement in which the effect of a concrete program in C can be achieved by some *arbitrary* program in A. Note that this is different from weak data refinement as given by [Derrick et al. 1997].

[3]Observations are often called global states. To avoid confusion with global and local states of threads we use the term observation here.

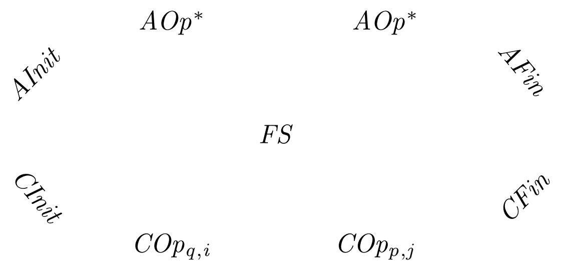


Fig. 1: Forward simulation conditions

Definition 3.5 (*Weak data refinement*). Let $A = (AState, Obs, AInit, (AOp_{p,i})_{p \in P, i \in I}, AFin)$ and $C = (CState, Obs, CInit, (COp_{p,j})_{p \in P, j \in J}, CFin)$ be two data types, each with an explicit finalization.

C is a *weak data refinement of* A if (a) the empty concrete program refines the empty abstract program, and (b) for all programs Prg, $Prg(C) \subseteq AInit \fatsemi AOp^* \fatsemi AFin$. □

It is well known that a refinement relationship can be verified by simulations. Instead of needing to compare *all* behaviors, the beauty of the simulation conditions is that they allow for a step-by-step comparison treating initialization, the operations and finalization separately. Simulations come in two flavors, forward and backward simulation. Together, these are sound and jointly complete as a methodology for verifying refinements [de Roever and Engelhardt 1998; Derrick and Boiten 2001]. In most standard applications of refinement, observation is the input/output of the data type [Woodcock and Davies 1996], however, in the context of verifying linearizability, our notion of observable behavior is the histories. In the next sections we will show that the concrete data type is a weak data refinement of the abstract data type using a finalization operation extracting histories. Here, we first of all define a notion of forward and backward simulation appropriate for weak data refinement.

Figure 1 shows the proof obligations for a forward simulation. As with all simulations, we need to find an abstraction (or retrieve) relation FS relating states of the concrete and the abstract data type, and show that initialization, finalization and operations of the concrete data type can be forward simulated by the abstract data type (in the sense that the diagrams commute).

Definition 3.6 (*Forward simulation*). Let $A = (AState, Obs, AInit, (AOp_{p,i})_{p \in P, i \in I}, AFin)$ and $C = (CState, Obs, CInit, (COp_{p,j})_{p \in P, j \in J}, CFin)$ be two data types with finalization. A relation $FS : AState \leftrightarrow CState$ is a *forward simulation* from A to C if the following three conditions hold:

— Initialization: $CInit \subseteq AInit \fatsemi FS$,
— Finalization: $FS \fatsemi CFin \subseteq AFin$,
— Correctness: $\forall\, p \in P, j \in J \bullet FS \fatsemi COp_{p,j} \subseteq AOp^* \fatsemi FS$. □

In the correctness condition we see that now one operation of the concrete data type can be *matched* by a (possibly empty) sequence of arbitrary abstract operations. We will use the term "match" in the following to mean the (sequence of) abstract operations for some concrete operation during simulation, both for forward and backward. We sometimes also say that a concrete operation is *mapped* to some abstract operation, or a sequence of abstract operations.

In Figure 2 we show the diagram for backward simulations, in which simulation by the abstract data type is carried out in a backward fashion (in the sense that the diagrams commute).

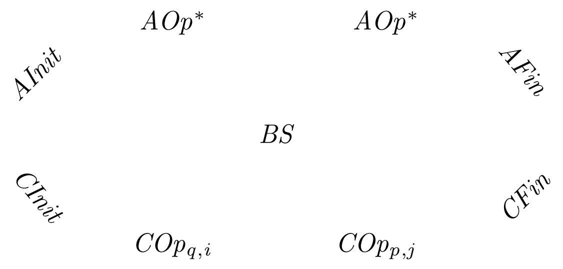


Fig. 2: Backward simulation conditions

Definition 3.7 (*Backward simulation*). Let $A = (AState, Obs, AInit, (AOp_{p,i})_{p \in P, i \in I}, AFin)$ and $C = (CState, Obs, CInit, (COp_{p,j})_{p \in P, j \in J}, CFin, GState)$ be two data types with the same observations. A relation $BS : AState \leftrightarrow CState$ is a *backward simulation* from C to A if the following three conditions hold:

— Initialization: $CInit \mathbin{⨾} BS \subseteq AInit$,
— Finalization: $CFin \subseteq BS \mathbin{⨾} AFin$,
— Correctness: $\forall\, p \in P, j \in J \bullet COp_{p,j} \mathbin{⨾} BS \subseteq BS \mathbin{⨾} AOp^*$. □

These two definitions give us a very general form of simulations which is needed for weak data refinement.

4. A SOUND AND COMPLETE PROOF TECHNIQUE

The starting point for our linearizability proofs is the following. Let $ADT = (AState, AInit, (AOp_{p,i})_{p \in P, i \in I})$ and $CDT = (CState, CInit, (COp_{p,j})_{p \in P, j \in J})$ be two data types, where sets I and J are used to index the abstract and concrete operations, and P is a set of process identifiers. The function $abs : J \rightarrow I$ defines the correspondence between abstract and concrete operations, and is assumed to be total and surjective, and an n-to-1 mapping between concrete and abstract operations. Abstract operations $AOp_{p,i}$ have input and output denoted $in? : IN_i$ and $out! : OUT_i$, concrete operations $COp_{p,j}$ either have an input $in? : IN_{abs(j)}$ (invoking operations), or an output $out? : OUT_{abs(j)}$ (returning operations) or no input and output at all.

Our objective now is to show that we can always prove linearizability via a backward simulation, i.e., CDT is linearizable with respect to ADT *if and only if* there is a backward simulation between two specifically constructed data types, called $HBDT$ and $HCDT$. These data types first of all serve as a theoretical vehicle for proving the completeness result; however, they can also be used in linearizability proofs for case studies as we will see for the Herlihy-Wing queue.

In the following we will construct in total three data types from ADT and CDT, called $HADT$, $HBDT$ and $HCDT$. $HADT$ and $HBDT$ are constructed out of ADT and $HCDT$ out of CDT. We will speak of the data types ADT and $HADT$ as residing on level A, $HBDT$ to constitute level B, and finally CDT and $HCDT$ to lay on level C. This idea of levels originates from the usual way of drawing refinement relationships, where the abstract specification is drawn on top of the concrete specification. In our case, level B constitutes some intermediate level.

Each of the new data types will, in addition to the state we had so far, have histories – which the definition of linearizability is based on – in their state. Adding the history as an auxiliary variable does not change the runs of the data types. It enables to add a finalization schema that compares the collected histories, implying that $HADT$, $HBDT$ and $HCDT$ are instances of data types as defined in Def. 3.3.

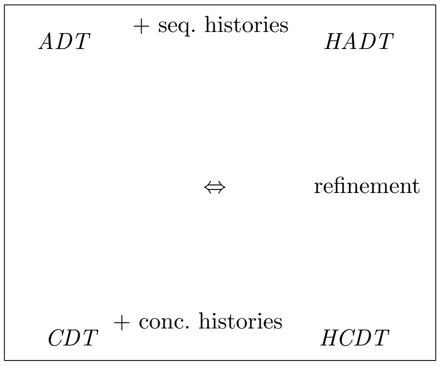


Fig. 3: Previous approach

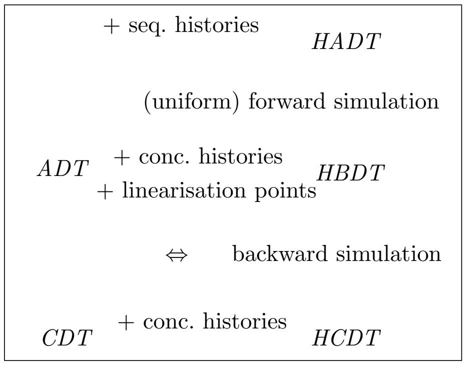


Fig. 4: New approach

4.1. Constructing our data types

Figure 3 shows our approach as detailed in previous papers. From a given ADT and CDT we first of all constructed data types $HADT$ and $HCDT$ which enhanced them with histories and a generalized finalization operation returning histories. For these, we showed that CDT is linearizable to ADT if and only if a refinement from $HADT$ to $HCDT$ exists. This is the main result we proved in [Derrick et al. 2011a]. Proving refinement usually involved forward *and* backward simulations.

Here, we are now interested in proving completeness of *backward* simulation alone for linearizability. To this end, we define the new intermediate specification $HBDT$ (we can think of these as forming three *levels*, and thus $HBDT$ sits in between level ADT and CDT, see Figure 4), which adds concurrency and a notion of linearization point to ADT. It, however, does not yet resemble the concurrent implementation. Just like $HADT$, $HBDT$ is constructed from the abstract data type ADT only. Level B splits the operations of ADT into invocations, returns and an explicit *linearization operations*, i.e., operations where the effect on the data structure happens. The main result of this paper is now that CDT is linearizable to ADT if and only if there is a backward simulation between $HBDT$ and $HCDT$. For the proof of this fact we need a further property of these three levels, namely that there *always* is a forward simulation between $HADT$ and $HBDT$, i.e., even if CDT is not linearizable. The existence of this forward simulation justifies that we can safely work with $HBDT$ in our completeness theorem.

The result implies that *all* linearizability proofs can in principle be done with *backward simulation*. However, it does not directly provide us with a technique for constructing these backward simulations for concrete examples. We next explain all levels and the forward and backward simulations in detail.

Abstract level. For our first level A we extend ADT with histories and finalization giving us a new data type $HADT = (\mathit{HAState}, \mathit{History}, \mathit{History} \times \mathit{HAInit}, (\mathit{HAOp}_{p,i})_{p\in P, i\in I}, \mathit{HAFin})$. Basically, we extend the local state of ADT with a new variable storing the current history of a run. States are thus of type (as, hs) where as is a state of ADT and hs a sequential history.

$$
\begin{array}{l}
\mathit{HAState} \mathrel{\widehat{=}} \mathit{AState} \land [hs : \operatorname{seq} \mathit{Event}] \\
\mathit{Obs} \mathrel{\widehat{=}} \mathit{History} \\
\mathit{HAInit} \mathrel{\widehat{=}} \mathit{AInit} \land [hs' : \operatorname{seq} \mathit{Event} \mid hs' = \langle\,\rangle] \\
\mathit{HAOp}_{p,i} \mathrel{\widehat{=}} \exists\, in? : \mathit{IN}_i, out? : \mathit{OUT}_i \bullet \\
\qquad\qquad \mathit{AOp}_{p,i} \land [hs, hs' : \operatorname{seq} \mathit{Event} \mid hs' = hs \frown \langle \mathit{inv}(p, i, in?), \mathit{ret}(p, i, out!)\rangle] \\
\mathit{HAFin} \mathrel{\widehat{=}} \mathit{HAState} \land [H : \operatorname{seq} \mathit{Event} \mid \mathit{linearizable}(H, hs)]
\end{array}
$$

As in all following definitions, initialization ignores the input history: it is defined to relate any history to the initial states specified by $HAInit \subseteq History \times Astate$. Operations immediately add invocations and returns to the history, thus level A is sequential. Furthermore note that the finalization does not return the sequential history itself but nondeterministically returns a *concurrent* history which is linearizable into the sequential history. Thus the finalization here is consistent with the finalization of a concrete data type returning a concurrent history iff the concurrent history is linearizable wrt. the sequential history.

For future reference in simulations we define a predicate $linval(hs, as)$, that characterizes possible final values as after running the operations of a sequential history hs by

$$linval(hs, as) \mathrel{\widehat{=}} (hs, as) \in HAOp^*(\!|\ HAInit\ |\!)$$

The definition makes use of the Z operator $R(\!|\ S\ |\!)$ returning the relational image of set S under relation R.

Intermediate level. In the intermediate level B we use a variable R which takes sets of return events. The state of data type $HBDT$ is defined to be of type (as, h, R), where as is a state of ADT, h a history and R a set of returns. This set R resembles the return events used in the linearizability definition to extend the current history $(\exists\, h_0 : \mathrm{seq}\, Ret! \bullet \ldots)$. These are the returns which have not happened so far but for which the effect of the operation has already taken place. Since we have an explicit formalization of linearization point in $HBDT$ (see next), it is easy to determine when such a return event has to be in the set R.

In addition, we divide the abstract operations into invocation, return and linearization operations. Invocation of an operation from a process p can only occur if there is no pending invocation of the same process in the history, returns can only occur when there is a pending invocation. The linearization operation, called Lin, is used to make the notion of "an operation taking effect" explicit. Thus the new operation Lin adds an appropriate return event to R. So, invocations and returns just extend the histories, whereas linearization changes both the local state $AState$ of the data structure and adds an event to R.

This gives data type $HBDT = (HBState, History, History \times HBInit, (Lin_{p,i})_{p\in P, i\in I} \cup (Inv_{p,i})_{p\in P, i\in I} \cup (Ret_{p,i})_{p\in P, i\in I}, HBFin)$ defined by

$$\begin{array}{rcl}
HBState & \widehat{=} & AState \wedge [h : \mathrm{seq}\, Event, R : \mathbb{P}\, Ret] \\
Obs & \widehat{=} & History \\
HBInit & \widehat{=} & AInit \wedge [h = \langle\,\rangle \wedge R = \varnothing] \\
Inv_{p,i} & \widehat{=} & [(\neg\, \exists\, i', in' \bullet inv(p, i', in') \in pi(h)) \wedge \\
 & & \exists\, in? : IN_i \bullet\ \ as' = as \wedge R' = R \wedge h' = h \frown \langle inv(p, i, in?)\rangle] \\
Lin_{p,i} & \widehat{=} & [\exists\, in, out \bullet inv(p, i, in) \in pi(h) \wedge (\neg\, \exists\, out_2 \bullet ret(p, i, out_2) \in R) \wedge \\
 & & \ AOp_{p,i}(in, as, as', out) \wedge h' = h \wedge R' = R \cup \{ret(p, i, out)\}] \\
Ret_{p,i} & \widehat{=} & [\exists\, out! : Out_i \bullet ret(p, i, out!) \in R \wedge h' = h \frown \langle ret(p, i, out!)\rangle \wedge \\
 & & \qquad\qquad\qquad R' = R \setminus \{ret(p, i, out!)\} \wedge as' = as] \\
HBFin & \widehat{=} & HBState \wedge [H : \mathrm{seq}\, Event \mid H = h]
\end{array}$$

Again the initialization relation ignores the initial history. We define $HBOp$ to be the union relation $\bigcup_{p\in P, i\in I}(Lin_{p,i} \cup Ret_{p,i} \cup Inv_{p,i})$ of all operations of $HBDT$, $Lin \mathrel{\widehat{=}} \bigcup_{p\in P, i\in I} Lin_{p,i}$ is all linearization operations, and similarly for Ret and Inv.

Note that level B is concurrent, i.e., produces concurrent histories. Note also that level B is only derived from level A, and uses the same index set I for operations.

Example: As an example we give a run of $HBDT$ which resembles the concrete run given in Section 2. Note that there is more than one such run. Recall that the state is $\langle as, h, R\rangle$ and as is the queue.

$$
\begin{array}{ll}
\multicolumn{2}{l}{\langle\langle\,\rangle, \langle\,\rangle, \varnothing\rangle} \\
\xrightarrow{Inv_{3,enq}(a)} & \langle\langle\,\rangle, \langle inv(3, enq, a)\rangle, \varnothing\rangle \\
\xrightarrow{Inv_{1,enq}(b)} & \langle\langle\,\rangle, \langle inv(3, enq, a), inv(1, enq, b)\rangle, \varnothing\rangle \\
\xrightarrow{Inv_{2,deq}()} & \langle\langle\,\rangle, \langle inv(3, enq, a), inv(1, enq, b), inv(2, deq,)\rangle, \varnothing\rangle \\
\xrightarrow{Inv_{4,enq}(c)} & \langle\langle\,\rangle, \langle inv(3, enq, a), inv(1, enq, b), inv(2, deq,), inv(4, enq, c)\rangle, \varnothing\rangle \\
\xrightarrow{Lin_{4,enq}} & \langle\langle c\rangle, \langle inv(3, enq, a), inv(1, enq, b), inv(2, deq,), inv(4, enq, c)\rangle, \{ret(4, enq,)\}\rangle \\
\xrightarrow{Lin_{2,deq}} & \langle\langle\,\rangle, \langle\ldots\rangle, \{ret(4, enq,), ret(2, deq, c)\}\rangle \\
\xrightarrow{Ret_{4,enq}()} & \langle\langle\,\rangle, \langle\ldots, ret(4, enq,)\rangle, \{ret(2, deq, c)\}\rangle \\
\xrightarrow{Lin_{3,enq}} & \langle\langle a\rangle, \langle\ldots, ret(4, enq,)\rangle, \{ret(2, deq, c), ret(3, enq,)\}\rangle \\
\xrightarrow{Ret_{2,deq}(c)} & \langle\langle a\rangle, \langle\ldots, ret(4, enq,), ret(2, deq, c)\rangle, \{ret(3, enq,)\}\rangle \\
\xrightarrow{Ret_{3,enq}()} & \langle\langle a\rangle, \langle\ldots, ret(4, enq,), ret(2, deq, c), ret(3, enq,)\rangle, \varnothing\rangle \\
\xrightarrow{Lin_{1,enq}} & \langle\langle a, b\rangle, \langle\ldots, ret(4, enq,), ret(2, deq, c), ret(3, enq,)\rangle, \{ret(1, enq,)\}\rangle \\
\xrightarrow{Ret_{1,enq}()} & \langle\langle a, b\rangle, \langle\ldots, ret(4, enq,), ret(2, deq, c), ret(3, enq,), ret(1, enq,)\rangle, \varnothing\rangle
\end{array}
$$

□

Concrete level. In a third step we now construct level $HCDT$ consisting of the concrete operations in the concurrent implementation extended with histories. This is similar to the extension $HADT$ which we have already defined for ADT. One difference here is that the finalization returns the history itself. In order to determine when to extend histories by *inv*'s or *ret*'s we classify all operations of the concrete data type into invocation, return or other operations. Therefore our final data type is defined as $HCDT = (HCState, History, History \times HCInit, (HCOp_{p,j})_{p \in P, j \in J}, HCFin)$ where

$$
\begin{array}{l}
HCState \mathrel{\widehat=} CState \wedge [h : \mathrm{seq}\, Event] \\
Obs \mathrel{\widehat=} History \\
HCInit \mathrel{\widehat=} CInit \wedge [h' : \mathrm{seq}\, Event \mid h' = \langle\,\rangle] \\
HCFin \mathrel{\widehat=} HCState \wedge [H : \mathrm{seq}\, Event \mid H = h]
\end{array}
$$

and the operations are defined by

$$
HCOp_{p,j} \mathrel{\widehat=}
\begin{cases}
\exists\, in? : IN_{abs(j)} \bullet COp_{p,j} \wedge [h, h' : \mathrm{seq}\, Event \mid h' = h \frown \langle inv(p, abs(j), in?)\rangle] \\
\qquad\qquad \text{iff } j \text{ is an invoke operation} \\
\exists\, out! : OUT_{abs(j)} \bullet COp_{p,j} \wedge [h, h' : \mathrm{seq}\, Event \mid h' = h \frown \langle ret(p, abs(j), out!)\rangle] \\
\qquad\qquad \text{iff } j \text{ is a return operation} \\
COp_{p,j} \wedge [h, h' : \mathrm{seq}\, Event \mid h' = h] \\
\qquad\qquad \text{otherwise}
\end{cases}
$$

All operations of the embedded type now work on the concrete state plus the history. Embedding an operation $COp_{p,j}$ that invokes an algorithm and has input $in?$ gives an operation $HCOp_{p,j}$ that adds a corresponding invoke event to the history, and similarly for returning operations. All others leave the history unchanged. Again, as expected, level C is concurrent.

Example: For the Herlihy-Wing queue we have invocation operations $enq0_p$ and $deq0_p$ and return operations $enq3_p$ and $deq6_p$ for all $p \in P$. This gives the following examples of enhanced concrete operations:

$$
\begin{array}{l}
HCEnq0_p \mathrel{\widehat=} \exists\, lv? \bullet enq0_p \wedge [h, h' : \mathrm{seq}\, Event \mid h' = h \frown \langle inv(p, enq, lv?)\rangle] \\
HCEnq1_p \mathrel{\widehat=} enq1_p \wedge [h, h' : \mathrm{seq}\, Event \mid h' = h\rangle] \\
HCDeq6_p \mathrel{\widehat=} \exists\, el! \bullet deq6_p \wedge [h, h' : \mathrm{seq}\, Event \mid h' = h \frown \langle ret(p, enq, el!)\rangle]
\end{array}
$$

□

4.2. Soundness and Completeness Results

These definitions help us to establish the main result of the paper, namely that backward simulations are sound and complete for showing linearizability. So, for the constructed data types of levels A, B and C we will now show that

(1) there always exists a forward simulation from $HADT$ to $HBDT$ (by showing that $Inv_{p,i}$ and $Ret_{p,i}$ in $HBDT$ are matched by empty steps of $HADT$ and $Lin_{p,i}$ in $HBDT$ by $HAOp_{p,i}$ in $HADT$), and – the main result – that
(2) a backward simulation from $HCDT$ to $HBDT$ exists if and only if CDT is linearizable with respect to ADT.

Central to both proofs will be the notion of *possibilities.* Possibilities have already been introduced in [Herlihy and Wing 1990] as an alternative way of defining linearizability. A possibility is a triple consisting of a history h, a set of return events R and a state as. Intuitively, $Poss(as, h, R)$ means that it is possible to reach abstract state as when executing the history h assuming that all returns in R have taken place, i.e., these operations have already taken effect and have changed the state. In [Derrick et al. 2011a] there is a rule-based definition of possibilities, which exactly matches the way we construct the state of level B. Thus we directly use it here.

Definition 4.1. A *possibility* is a reachable state of the B-level. Thus we define $Poss(as, h, R)$ by the following (remember from above $HBOp$ is the union of all operations in B) :

$$Poss(as, h, R) \mathrel{\widehat=} (as, h, R) \in HBOp^*(\!|\ HBInit\ |\!) \qquad \square$$

Note that the histories occurring in possibilities are concurrent. We get the following property of possibilities.

PROPOSITION 4.2. *Possibilities are prefix-closed: If $Poss(as, h_0 \frown h, R)$ for some histories h_0, h, set of returns R and abstract state as, then there are as_0 and R_0, such that $Poss(as_0, h_0, R_0)$ and $HBOp^*((as_0, h_0, R_0), (as, h_0 \frown h, R))$.*

Proof: Simple induction over the number of operation applications necessary to reach the final state $(as, h_0 \frown h, R)$, since every operation adds at most one event and we start with the empty history. Remark: This lemma is already known from [Herlihy and Wing 1990], p. 487. □

Possibilities and linearizability are equivalent in the following sense: Whenever we have a possibility (as, h, R), then we can arrange the return events in R into some arbitrary order, append them to h and by this get a linearizable history. Conversely, if a concurrent history can be extended (with those returns for which the effect of the operation has already taken place) and is then linearizable, we also have a possibility for this history.

To formally state this close connection, we need a definition to relate return event sets and histories. We let $setof(h)$ stand for the set of events occurring in h. We then define $R = setof(h)$ to be true iff h is a (duplicate free) sequence that contains the same set of events as R. The following theorem just formalizes one direction, namely that linearizability implies possibilities. (Recall that $complete(h \frown h_0, hs)$ removes all pending invocations from $(h \frown h_0, hs)$, and lin is the linearizable predicate.)

THEOREM 4.3. *Let h be a history and as an abstract state. Then the following holds.*

$$\exists\, h_0, hs \bullet lin(complete(h \frown h_0), hs) \wedge linval(hs, as) \wedge h_0 \subseteq seq\,Ret! \\ \Rightarrow Poss(as, h, setof(h_0))$$

This theorem is very similar to Theorem 10 of [Herlihy and Wing 1990]; we have given a proof in [Derrick et al. 2011a]. □

With these definitions and results at hand, we can formulate and prove our first result about the existence of a forward simulation between levels A and B.

THEOREM 4.4. *Let HADT and HBDT be the data types of levels A and B as defined in Section 4.1. Then there exists a forward simulation from HADT to HBDT.*

Proof: We construct a forward simulation relation FS from $HADT$ to $HBDT$. Note that states of $HADT$ are of the form (as, hs) and those of $HBDT$ of type (bs, h, R). We first define the forward simulation relation we will use as the following:

$$FS = \{((as, hs), (bs, h, R)) \mid as = bs \wedge Poss(bs, h, R) \wedge linval(hs, as) \wedge \\ \forall\, h_0 \bullet R = setof(h_0) \Rightarrow lin(complete(h \frown h_0, hs)\}$$

remembering that the predicate $linval(hs, as)$ characterizes possible final values as after running the operations of a sequential history hs.
We need to show that this is indeed a forward simulation.

Initialization. Straightforward as $HBInit(bs, h, R)$ implies $AInit(bs)$ and the definition of $HBDT$ immediately gives us $Poss(bs, h, R)$. The rest then follows by the fact that $lin(\langle\rangle, \langle\rangle)$ holds.
Finalization. Straightforward: by definition of FS we know that for all $((as, hs), (bs, h, R)) \in FS$ the history h is linearizable into hs.
Correctness. Let $((as, hs), (bs, h, R)) \in FS$. We split the correctness proof into three cases covering invoke, return and linearization operations.

(1) Assume the concrete operation in level B to be executed is an *invoke.* Hence $(bs, h, R) \xrightarrow{Inv_{p,i}} (bs, h \frown \langle inv(p, i, in?)\rangle, R)$ for some input $in?$. For the level A we choose an empty ($skip$) step. Thus we need to show that $((as, hs), (bs, h \frown \langle inv(p, i, in?)\rangle, R)) \in FS$:
— (i) $as = bs$ still holds,
— (ii) $Poss(bs, h \frown \langle inv(p, i, in?)\rangle, R)$ holds by definition of $Poss$,
— (iii) $\forall\, h_0 \bullet R = setof(h_0 \Rightarrow lin(complete(h \frown \langle inv(p, i, in?)\rangle \frown h_0, hs))$ holds since *complete* is removing the (new) pending invoke,
— (iv) $linval(hs, as)$ is still true.

(2) Assume the concrete operation in level B to be executed is a *return.* Hence $(bs, h, R) \xrightarrow{Ret_{p,i}} (bs', h', R')$ where (bs', h', R') is $(bs, h \frown \langle ret(p, i, out!)\rangle, R \setminus \{ret(p, i, out!)\})$ for some output $out!$. For the level A we again choose an empty ($skip$) step. Thus we need to show that $((as, hs), (bs, h \frown \langle ret(p, i, out!)\rangle, R \setminus \{ret(p, i, out!)\})) \in FS$:
— (i) $as = bs$ still holds,
— (ii) $Poss(bs, h \frown \langle ret(p, i, out!)\rangle, R \setminus \{ret(p, i, out!)\})$ by definition of $Poss$,
— (iii) We need to show $\forall\, h'_0 \bullet R' = setof(h'_0) \Rightarrow lin(complete(h' \frown h'_0, hs))$. We take an arbitrary h'_0. From this we construct a sequence $h_0 \mathrel{\widehat=} \langle ret(p, i, out!)\rangle \frown h'_0$. For this we have $R = setof(h_0)$, hence $lin(complete(h \frown h_0, hs))$. The fact that $h \frown h_0 = h' \frown h'_0$ gives us the desired result.
— (iv) $linval(hs, as)$ is still true.

(3) Assume the concrete operation in level B to be executed is a *linearization* step. Hence $(bs, h, R) \xrightarrow{Lin_{p,i}} (bs', h', R')$ where for bs' we know that $AOp_{p,i}(in?, bs, bs', out!)$ and $inv(p, i, in?) \in pi(h)$, $h' = h$ and $R' = R \cup$

$\{ret(p,i,out!)\}$. Now we choose $AOp_{p,i}$ as corresponding abstract step and this brings the abstract state to $as' \mathrel{\widehat{=}} bs'$ and $hs' \mathrel{\widehat{=}} hs \frown \langle inv(p,i,in?), ret(p,i,out!)\rangle$. Again we get $as' = bs', Poss(bs', h', R')$ and $linval(hs', as')$ by definition. The interesting part is again the requirement about lin.

Since we have taken a Lin-step, $inv(p,i,in) \in pi(h)$ must hold, so $h = h_1 \frown \langle inv(p,i,in)\rangle \frown h_2$ for some h_1 and h_2. Also $AOp_i(in, as, as', out)$ and $R' = R \cup ret(p,i,out)$ must be true to have $Poss(as', h', R')$.

We set $hs' \mathrel{\widehat{=}} hs \frown \langle inv(p,i,in), ret(p,i,out)\rangle$, and have to prove $lin(complete(h' \frown h'_0, hs'))$ for every h'_0 that satisfies $R' = setof(h'_0)$. The latter implies $h'_0 = h'_1 \frown \langle ret(p,i,out)\rangle \frown h'_2$, since $ret(p,i,out) \in h'_0$, and we can choose $h_0 \mathrel{\widehat{=}} h'_1 \frown h'_2$ in the induction hypothesis. Since h_0 satisfies $R = setof(h_0)$, we get $lin(complete(h \frown h_0), hs)$ and $linval(hs, as)$.

Now, $complete(h \frown h_0) = complete(h_1 \frown h_2 \frown h'_1 \frown h'_2)$ since $inv(p,i,in)$ is a pending invoke in $h \frown h_0$, that is dropped by the complete-function. The induction hypothesis therefore gives an order-preserving bijection between $complete(h_1 \frown h_2 \frown h'_1 \frown h'_2)$ and hs. This bijection can be extended to one between
$complete(h' \frown h'_0) = complete(h_1 \frown \langle inv(p,i,in)\rangle \frown h_2 \frown h'_1 \frown \langle ret(p,i,out)\rangle \frown h'_2)$
and hs' as required to prove lin. This is done by adding a mapping between the two additional events $inv(p,i,in)$ and $ret(p,i,out)$ (the technical details are quite complex, since *positions* must be mapped, which change by one for events between the invoke and the return, and by two for events in h'_2). The new mapping is order preserving, since no operation is started in $h' \frown h'_0$ after the return $ret(p,i,out)$.

□

As a corollary we now get the other connection between possibilities and linearizability (a strengthened version of Theorem 9 in [Herlihy and Wing 1990]):

COROLLARY 4.5. *Let h be a history, R a set of return events and as an abstract state. Then*

$$Poss(as, h, R) \Rightarrow$$
$$\exists\, hs \bullet \forall\, h_0 \bullet R = setof(h_0) \Rightarrow lin(complete(h \frown h_0), hs) \wedge linval(hs, as)$$

Proof: Assuming $Poss(as, h, R)$, the forward simulation asserts the existence of (hs, as') with $FS((hs, as'), (h, as, R))$. Expanding the definition of FS gives the desired conclusion. □

Our second result is the one that links the intermediate to the concrete level. Here, we can show that linearizability is equivalent to the existence of a backward simulation. This is our soundness and completeness result for linearizability.

THEOREM 4.6. *Let $HBDT$ and $HCDT$ be the data types of levels B and C as defined above, and let ADT and CDT be the original data types we started with. Then CDT is linearizable with respect to ADT if and only if there is a backward simulation between $HCDT$ and $HBDT$.*

Proof: We have to prove two parts: (a) *Soundness:* the existence of an arbitrary BS being a backward simulation between $HBDT$ and $HCDT$ implies that CDT linearizable wrt. ADT, and (b) *Completeness:* CDT linearizable wrt. ADT implies that there is a backward simulation BS (which we will give).

$$\begin{tikzcd}
\exists(as, h_B, R) \arrow[r, phantom, "HBOp^*"] \arrow[d, phantom, "BS"] & (as', h'_B, R') \arrow[d, phantom, "BS"] \\
(cs, h_C) \arrow[r, phantom, "HCOp_{p,j}"] & (cs', h'_C)
\end{tikzcd}$$

Fig. 5: Correctness rule for backward simulation

Part (a): Assume there is a backward simulation BS between $HCDT$ and $HBDT$, and assume an arbitrary run of $HCDT$ is given, that starts with (cs_0, h_0) such that cs_0 is initial and $h_0 = \langle\rangle$, goes through (cs_i, h_i) for $0 \le i \le n$ and ends in (cs_n, h_n). We have to prove that h_n is a linearizable history. The finalization condition ensures that there is a state $(as_n h_n, R_n)$ of $HBDT$ with the same h_n. Now the correctness condition of backward simulation (see Fig. 5) guarantees that $(as_{n-1}, h_{n-1}, R_{n-1})$ can be found with $BS((cs_{n-1}, h_{n-1}), (as_{n-1}, h_{n-1}, R_{n-1}))$ and $HBOp^*((as_{n-1}, h_{n-1}, R_{n-1}), (as_n, h_n, R_n))$. Iterating the construction (see Fig. 6) gives states (as_i, h_i, R_i), all linked via BS (and therefore all having the correct h_i because of finalization) such that $HBOp^*((as_i, h_i, R_i), (as_{i+1}, h_{i+1}, R_{i+1})$. Altogether we have $HBOp^*((as_0, h_0, R_0), (as_n, h_n, R_n)$. Since $HCInit(cs_0, h_0)$ holds, the initialization condition implies $HBInit(as_0, h_0, R_0)$. Together this implies that (as_n, h_n, R_n) is a reachable state of $HBDT$, i.e., $Poss(as_n, h_n, R_n)$. Finally, Corollary 4.5 gives the desired linearizability of h_n.

Part (b): assume CDT is linearizable wrt. ADT. We prove that BS defined by

$$BS = \{((cs, h_C), (as, h_B, R)) \mid Poss(as, h_B, R) \land h_B = h_C \land (h_B = \langle\,\rangle \Rightarrow AInit(as))\}$$

is always a backward simulation.

Initialization. if $HCInit(cs, h_C)$ and $BS((cs, h_C), (as, h_B, R))$ then we know that $h_B = h_C$ and thus empty, and furthermore $AInit(as)$ and $R = \varnothing$ which implies $HBInit(as, h_B, R)$.

Correctness. requires that given a concrete transition $(cs, h_C) \xrightarrow{HCOp_{p,i}} (cs', h'_C)$ and an abstract state (as', h'_B, R') we need to find a state (as, h_B, R) such that $((cs, h_C), (as, h_B, R)) \in BS$ and $(as, h_B, R) \xrightarrow{HBOp^*} (as', h'_B, R')$ (see Figure 5). Essentially this follows from Proposition 4.2: We know that h_C is a prefix of h'_C (histories are only extended by operations), thus by prefix-closedness of possibilities there must be some possibility $Poss(as, h_C, R)$ such that $HBOp^*((as, h_C, R), (as', h'_C, R'))$. If h_C is moreover empty, as has to be an initial state since possibilities with empty histories always contain an initial state. Thus we get $BS((cs, h_C), (as, h_C, R))$.

Finalization. If $HCFin((cs, h_C), h_C)$ then by linearizability of CDT wrt. ADT we get the following: $\exists\, h_S$ such that $\exists\, h_0 \in \mathrm{seq}\, Ret! \bullet lin(complete(h_C \frown h_0), h_S) \land \exists\, as \bullet linval(h_S, as)$. By Theorem 4.3 we get $Poss(as, h_C, setof(h_0))$. Hence this has to be a state of $HBDT$. Furthermore, if h_C (and thus h_B) is empty, we have $AInit(as)$ as the initial state is the only one which belongs to a possibility with an empty history, and thus $BS((cs, h_C), (bs, h_C, setof(h_0))$. □

A comparison of Theorem 4.6 to general completeness results, which imply that backward simulations and history variables are jointly complete for data refinement is given in Section

$$\begin{tikzcd}
(as_0, h_0, R_0) \arrow[r, phantom, "HBOp^*"] & \dots\ (as_{n-1}, h_{n-1}, R_{n-1}) \arrow[r, phantom, "HBOp^*"] & (as_n, h_n, R_n) \\
(cs_0, h_0) \arrow[u, phantom, "BS"] \arrow[r, phantom, "HCOp_{p,j_1}"] & \dots\ (cs_{n-1}, h_{n-1}) \arrow[u, phantom, "BS"] \arrow[r, phantom, "HCOp_{p,j_n}"] & (cs_n, h_n) \arrow[u, phantom, "BS"]
\end{tikzcd}$$

Fig. 6: Composing commuting bw. sim. diagrams

8 on related work. In summary, Theorem 4.6 shows, that for linearizability, the only history variable ever needed is the history needed to define linearizability (i.e., possibilities) itself.

The backward simulation BS of the completeness proof is in some sense a *maximal* backward simulation, since every other backward simulation B will satisfy

$$B((cs, h_B), (as, h_C, R)) \rightarrow BS((cs, h_B), (as, h_C, R))$$

for all reachable states (cs, h) as the soundness proof shows. BS is also *completely* independent of the concrete state cs. In applications we will of course make use of cs to define *smaller* simulations: relating fewer states (but not too few!) makes the proof of the main correctness condition for backward simulation easier, since fewer commuting diagrams have to be completed in the correctness condition.

Since any backward simulation must always keep the concrete and the abstract history identical, the "matching" sequence $HBOp^*$ used in the commuting diagrams of Fig. 5 can be specialized. For an invoking operation the sequence must have the form $Lin^* \mathbin{⨾} Inv \mathbin{⨾} Lin^*$, (recall, that Lin is the union of all linearization operations $Lin_{p,i}$; similarly for Inv and Ret), since only sequences of that type will update the history by adding one invoke event.

For a returning operation the matching sequence $HBOp^*$ must be $Lin^* \mathbin{⨾} Ret \mathbin{⨾} Lin^*$. All other operations can only correspond to a sequence Lin^* of linearization points.

In the case study we will map only internal operations (i.e., ones that do not change the history) to Lin-steps. This seems to be possible in general, when invoke and return operations only copy input/output to/from local buffers, and when all algorithms have at least one step between invoke and return. For the maximal backward simulation it is possible to show that it executes all linearization points *as early as possible*, i.e., directly with invoking operations. This can be shown based on the following observation.

PROPOSITION 4.7. *For processes $p, q \in P$, $p \neq q$, and arbitrary operation indices i, j, the operations of HBDT satisfy*

$$Lin_{p,i} \mathbin{⨾} Inv_{q,j} = Inv_{q,j} \mathbin{⨾} Lin_{p,i} \text{ and } Lin_{p,i} \mathbin{⨾} Ret_{q,j} = Ret_{q,j} \mathbin{⨾} Lin_{p,i}$$

Linearization steps commute with both invoke and return steps of other processes.

Therefore we can strengthen the completeness part of theorem 4.6 above to:

THEOREM 4.8. *If CDT is linearizable with respect to ADT, then it is possible to find a backward simulation between HBDT and HCDT such that in the correctness condition*

$$HCOp_{p,j} \mathbin{⨾} BS \subseteq BS \mathbin{⨾} HBOp^*$$

the abstract operation sequence $HBOp^*$ can be strengthened to:

— *$Ret_{p,i}$, when the concrete operation is a return operation and $abs(j) = i$*
— *$Inv_{p,i} \mathbin{⨾} (skip \cup (Lin_{p,i} \mathbin{⨾} Lin^*))$ for invoke operations*
— *skip for other operations*

Proof:

The backward simulation is the same as in the completeness proof above.

— Internal steps were already matched to skip in the original proof.
— For return steps, the possibility (as, h, R) that was found in the proof above is connected to (as', h', R') via a sequence $Lin^* \fatsemi Ret_{p,i} \fatsemi Lin^*$, where p is the process executing the concrete return for operation i. The second sequence cannot contain any linearization steps for p, since process p is not running an operation $(h' = h \frown \langle ret(p, i, out)\rangle$, so p has no pending invoke). Therefore all these steps can be commuted with the return step, giving $(Lin^* \fatsemi Ret_{p,i})((as, h, R), (as', h', R'))$. The backward simulation can therefore choose the state (as_0, h_0, R_0) right before $Ret_{p,i}$, and match the concrete return step to a return step only.
— The sequence of abstract operations matching a concrete step is $Lin^* \fatsemi Inv_{p,i} \fatsemi Lin^*$. A similar argument to the above would move all Lin-steps *after* the invoke, which does not help. Moving all *Lin*-steps before the invoke is possible only, if the sequence does not contain a linearization step for p itself. If it does, the best we can do is to leave a sequence $Lin_{p,i} \fatsemi Lin^*$ after the invoke, and just as for the return, choose the possibility right before $Inv_{p,i}$. □

The maximal simulation that is used by Theorems 4.6 therefore has (by virtue of matching particular concrete to abstract operations) linearization points as early as possible, where they are matched directly with invoking steps on the concrete level.

The simulation we construct for our case study will however, *delay* possible linearization points as far as possible to the end of running an operation. This has the advantage of making the backward simulation smaller (than the most general one used in the proof) and therefore makes the correctness conditions easier to verify.

In summary the theorems of this section give us a sound and complete proof technique for showing linearizability. The key part of a linearizability proof thus lies in finding an appropriate backward simulation (and showing it to actually be one). Next, we will see how this works for the Herlihy-Wing queue, and along this example we give some general guidelines for finding backward simulations.

5. VERIFICATION OF THE CASE STUDY

The theory given in the last section ensures that any linearizable algorithm can be verified using a backward simulation BS between $HBDT$ and $HCDT$. In addition, if we can find a backward simulation between the data types, we know linearizability holds. However, the actual backward simulation BS used in the completeness result is, as we commented, maximal in some sense. Thus it is not always practical to use this backward simulation for a specific verification. In this section we show how to use the insights gained from what the backward simulation must be to derive a smaller relation between concrete states (cs, h) and abstract states (as, H, R) that can verify the Herlihy-Wing queue linearizable.

As the abstract state in our case study consists of the queue variable q only, we also write (q, H, R) for the state of $HBDT$. As a first observation, the finalization condition requires $h = H$ and thus we can always split BS into the part relating state spaces and that of relating histories.

$$BS((cs, h), (q, H, R)) \mathrel{\widehat{=}} B(cs, q, R) \wedge h = H$$

The key insight we now need for this case study is that for finding the backward simulation one has to analyze the *observations made by future behaviors*. We believe this to be a strategy which is useful for other examples as well.

To explain this idea, assume we want to define the set of states (as, h, R) related via BS to some state (cs, h). The smaller we make this set, the fewer states we have to consider in the proof of the correctness condition (every (as, h, R) will have to be considered as an instance of (as', h'_B, R') in Figure 5). Now the future runs from (cs, h) will give us some information, which states we cannot avoid to include. To understand this consider a run from $(cs_0, h_0) := (cs, h)$ to some state (cs_n, h_n). Assume we know already that BS must map (cs_n, h_n) to (as_n, h_n, R_n). Candidates for (cs_n, h_n) are quiescent states, where no process is running. For these it is clear that set R must be empty, and the correct abstract state is usually obvious. E.g., when cs_n has no running processes and an empty array, it is obvious that the corresponding abstract state related must have an empty queue and an empty set R.

Then it is necessary to define at least one witness (as, h, R) related via BS to the current state (cs, h), that shows that the state (as_n, h_n, R_n) is possible, i.e. for which $HBOp^*((as_0, h_0, R_0), (as_n, h_n, R_n))$ holds. Otherwise the diagram shown in Figure 6 will not commute.

To demonstrate the idea for our example consider the state cs (which corresponds to the example run explained in Section 2 up to step 7) shown at the top of Figure 7. The state shows a situation where the array contains two elements, $AR(0) = a$ and $AR(2) = c$. Furthermore process 1 is running an enqueue operation that tries to enqueue the element b at position 1, which has reached $pcf(1) = E2$, but then has been preempted. We call such an operation with $pcf(1) = E2$ a *pending enqueue*, and write $PE_1(b)$ in the figure to indicate it. Note that the "gap" in the array is due to this enqueue: it has increased the global *back* pointer before the enqueue of c, but has not executed statement E2 yet. In addition there is a *pending dequeue* of process 2 (PD_2) currently looking at position 1 as well. Such a dequeue operation has already initialized its *lback* ($pc \neq D1$), but has not yet successfully retrieved an element ($pc \neq D6$, and if $pc = D4$, then still $lv = null$). The history so far is

$$h = \langle inv(3, enq, a), inv(1, enq, b), inv(2, deq,), inv(4, enq, c), ret(3, enq,), ret(4, enq,)\rangle$$

To define B we now have to find out what possible abstract queue states this concrete state could correspond to. For this we look at observations made about this state when proceeding with executions on it. The *observation tree* shows all future executions from this state when new *observers* are started. An observer gives us information about the elements in the data structure, most often by extracting data from it. For our queue, the observers are dequeue operations. Processes currently running (like the enqueue) might or might not be continued.

First, consider the leftmost branch. It describes the following steps: (1) the pending enqueue of process 1 runs to completion ($PEnq_1$), then (2) the pending dequeue runs to completion and returns the element in position 1 which is b ($PDeq_2(1)$), (3) a new dequeue is started (of whatever process), runs to completion and returns the element stored in position 0 which is a ($NDeq(0)$), and (4) another new dequeue starts, completes and returns the element in position 2 which is c. Hence from the point of view of these dequeues the queue content has been $\langle b, a, c\rangle$. Note that we do not start any new enqueues, we just *observe* the existing state of the queue.

The rightmost branch executes pending operations in a different order (first the dequeue and then the enqueue) and again runs two observing dequeues. Here, we see that the queue is $\langle c, a, b\rangle$. Different future executions thus give different orderings of queue elements. Still, the order is not arbitrary: for instance $\langle b, c, a\rangle$ is impossible. We see that it is not only the current state of the array which determines the queue content, but also the pending enqueues and dequeues, and their current position into the array. Hence we cannot define our backward simulation B as a *function* from concrete to abstract state since this would contradict one or the other run. In summary, the backward simulation we look for must relate the current state cs to *any* queue that is possible in a future observation.

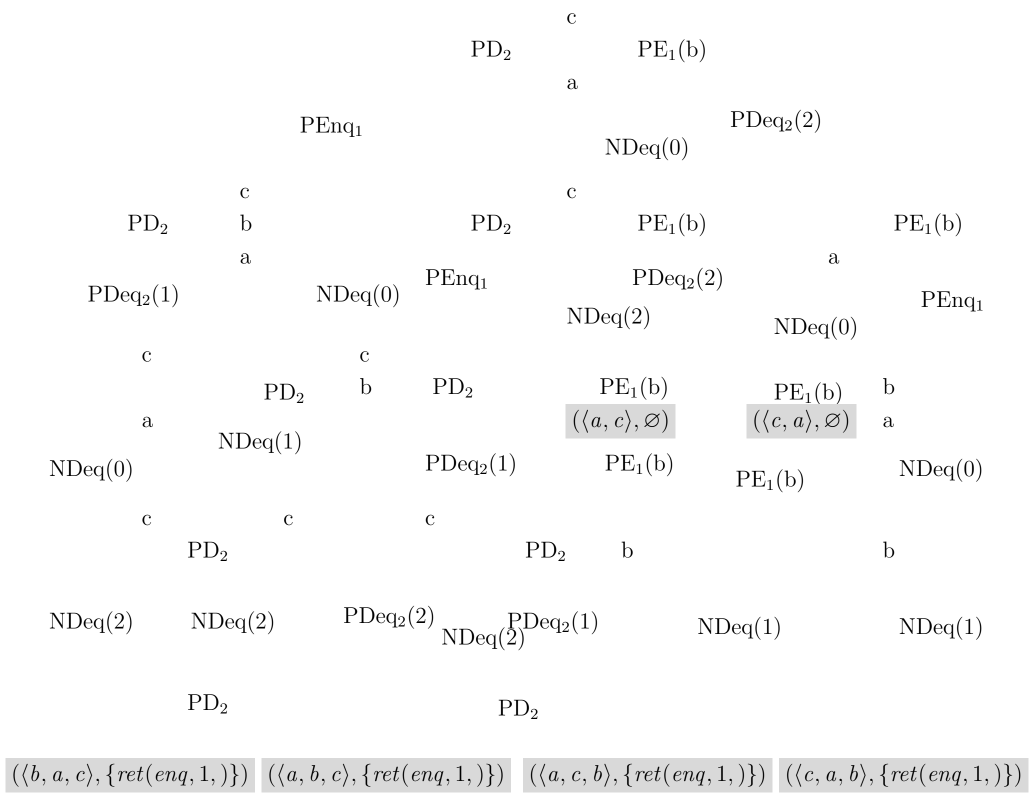


Fig. 7: Observation tree for an example state *cs*

We still have to determine the R-components B relates concrete states to. Recall that R collects linearization points. Again, general advice on finding a backward simulation is to *defer decisions as far as possible to the future* (this observation is not specific to linearizability or concurrency, see [Banach and Schellhorn 2010]). For our case, we delay any linearization point that still can be executed to the future, i.e. we do *not* add it to set R. This is possible for pending dequeues. These can linearize at the time they swap the element: they have a *definite* linearization point in the sense that we can attach it to line $D3$ when they swap a non-null element. However, enqueue operations cannot linearize in the future, since they would put the element in the wrong place in the queue. We find that enqueue already *potentially* linearizes when it executes $E1$, but only if the future run considered executes the operation to the end. In other runs, linearization will happen when the element is actually inserted at line $E2$.

These considerations now help us towards defining B. We write $NDeq(n)(cs, cs')$ to mean that a new (observer) dequeue is started, returns the element in array position n and brings the concrete state from cs to cs':

$NDeq(n)$

$\Delta CState$

$eb(AR, 0, n)$
$AR(n) \neq null$
$AR'(n) = null$

where eb ("empty between") is defined as

$$eb(AR, m, n) \mathrel{\widehat{=}} \forall\, k \bullet m \leq k < n \Rightarrow AR(k) = null$$

Similarly, we write $PDeq_p(n)(cs, cs')$ to say the same for an already running (pending) dequeue of process p:

$$\begin{array}{l}
\underline{\quad PDeq_p(n) \quad} \\
\Delta\, CState \\
\hline
\quad (pcf(p) \in \{D4, D5\} \wedge kf(p) + 1 \leq n \wedge eb(AR, kf(p) + 1, n)) \\
\vee\, (pcf(p) \in \{D2, D3\} \wedge kf(p) \leq n \wedge eb(AR, kf(p), n)) \\
n < lbackf(p) \\
lvf(p) = null \wedge lvf'(p) = AR(n) \\
AR(n) \neq null \wedge AR'(n) = null \\
pcf'(p) = N \\
\hline
\end{array}$$

and finally $PEnq_p(cs, cs')$ for the completion of a pending enqueue:

$$\begin{array}{l}
\underline{\quad PEnq_p \quad} \\
\Delta\, CState \\
\hline
pcf(p) = E2 \wedge pcf'(p) = N \\
AR'(kf(p)) = lvf(p) \\
\hline
\end{array}$$

The actual definition of B *recursively* follows the paths of the tree and has to consider four cases:

— The array is empty. Then the queue is empty as well and the set R consists of return events for those processes which have definitely achieved their effect (denoted $outs(cs)$). In our case, these are all the enqueues after their store (at $E3$), and the dequeues after the non-$null$ swap (at $D6$ or at $D4$, when $lv \neq null$).

— An observing dequeue (newly started) returns the element in position n of the array. All elements below n must be $null$. The corresponding abstract queue thus has $AR(n)$ as its first element. The rest of the queue (and of B) is defined by recursion.

— A pending dequeue finishes and returns the element in position n of the array. Thus again one of the corresponding abstract queues has $AR(n)$ as first element. The rest of the queue (and B) is defined by recursion.

— A pending enqueue finishes and the corresponding return event is already in R. Then the effect on the abstract queue has already taken place, i.e., $ret(p, enq,) \in R$. B is defined by recursion using the same queue q, but removing the return event from R.

Putting into one definition (and taking as abstract state as the queue state q) we get

$$\begin{aligned}
B(cs, q, R) := \quad & (\forall\, i : \mathbb{N} \bullet AR(i) = null) \wedge q = \langle\rangle \wedge R = outs(cs)) \\
& \vee (\exists\, q', n \bullet q = \langle AR(n) \rangle \frown q' \wedge (NDeq(n) \fatsemi B)(cs, q', R)) \\
& \vee (\exists\, q', p, n \bullet q = \langle AR(n) \rangle \frown q' \wedge (PDeq_p(n) \fatsemi B)(cs, q', R)) \\
& \vee (\exists\, p \bullet ret(enq, p,) \in R \wedge (PEnq_p \fatsemi B)(cs, q, R \setminus \{ret(enq, p,)\}))
\end{aligned}$$

Applying this technique to our example state cs in the root of Figure 7 gives a total of six pairs (q, R) with $B(cs, q, R)$. These are written with shaded background at those nodes of the tree where the array is empty.

Note that the definition of B is well-founded: $PEnq$ removes a pending enqueue process (and adds one element to the array), $PDeq$ and $NDeq$ each remove an array element. The corresponding well-founded order $<_B$ plays a central role in the correctness proofs of the next section.

6. VERIFICATION WITH KIV

The previous section has discussed the verification of the Herlihy-Wing queue, specifically it defined the backward simulation B we will use. In this section we discuss the mechanization of the proof obligations with KIV. KIV [Reif et al. 1998] is an interactive verifier, based on structured algebraic specifications using higher-order logic (simply typed lambda-calculus). Crucial features of KIV used in the proofs here are the following.

— Proofs in KIV are explicit proof trees of sequent calculus which are graphically displayed and can be manipulated by pruning branches, or by replaying parts of proofs after changes. This is of invaluable help to analyze and efficiently recover from failed proof attempts due to incorrect theorems, which is typically the main effort when doing a case study like the one here.
— KIV implements correctness management: lemmas can be freely used before being proved. This allows to focus on difficult theorems first, which are subject to corrections. Changing a lemma invalidates those proofs only that actually used it.
— KIV uses heuristics (e.g., for quantifier instantiation and induction) together with conditional higher-order rewrite rules to automate proofs. The rules are compiled into functional code, which runs very efficiently even for a large number of rules: the case study here uses around 2000 rules, 1500 of these were inherited from KIV's standard library of data types.

KIV was used to verify the completeness result for backward simulation as well as to prove the resulting proof obligations for the queue case study. A web presentation of all specifications and proofs can be found online [KIV 2011]. The completeness proof follows the proof given in Section 4, in terms of mechanization the difficult part is Theorem 4.4.

The correctness of the queue implementation is proved by instantiating the backward simulation relation B with the concrete operations of the Herlihy-Wing queue which were sketched in Section 2. This results in proof obligations that are instances of the backward simulation as given in Definition 3.7.

The interesting proof obligations for the case study are the correctness conditions for each operation. These can be written as[4]

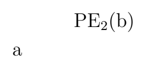
$$\begin{aligned}&(HCOp_{p,j} \mathbin{⨾} BS)((cs,H),(q',H',R')) \Rightarrow\\ &\qquad \exists\, q,R \bullet BS((cs,H),(q,H,R)) \wedge HBOp^*((q,H,R),(q',H',R'))\end{aligned}$$

Whilst the previous section defined the simulation we will use, it did not determine the specific operation(s) that $HBOp^*$ should be instantiated to. Thus the question now is which $HBOp^*$ to use here. A suitable sequence of abstract operations $HBOp^*$ that fixes q and R is easy to determine in most cases: for invoking and returning operations it is just the corresponding abstract invoke and return. For all other operations, except $enq1_p$, $enq2_p$ and $deq3t_p$ (the case of $deq3$, where the swap is with a non-*null* element), the sequence is empty. These correspond to cases where the observation tree for the current state cs is not changed by the operation. For $deq3t_p$ and $enq2_p$ the sequence is the linearization step $Lin_{p,deq}$ resp. $Lin_{p,enq}$. These two operations reduce the observation tree to one of its branches. The only difficult case is when $COp_{p,j}$ is $enq1_p$ which is explained below. With (q',R') denoting the result state of the chosen sequence $HBOp^*$, all other proof obligation simplify to

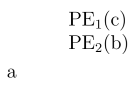
$$(COp_{p,j} \mathbin{⨾} B)(cs,q',R') \Rightarrow B(cs,q,R)$$

The simplicity of the changes to the observation tree is then reflected by the simplicity of the proofs: they all are by well-founded induction over $<_B$, followed by a case split over the definition of B. This gives a trivial base case and three recursive cases for each operation

[4]For easier readability, we leave out the invariants of the two data types.



possible pairs (q, R):
$\langle a\rangle, \varnothing$
$\langle a, b\rangle, \{ret(2, enq.)\}$



possible pairs (q', R'):
$\langle a\rangle, \varnothing$
$\langle a, b\rangle, \{ret(2, enq,)\}$
$\langle a, c\rangle, \{ret(1, enq,)\}$
$\langle a, b, c\rangle, \{ret(1, enq,), ret(2, enq,)\}$
$\langle a, c, b\rangle, \{ret(2, enq,), ret(1, enq,)\}$

Fig. 8: Results of B before and after executing $enq1_2$

$PN \in \{PDeq_q(n), PEnq_q, NDeq(n)\}$. The resulting goals can be closed immediately with the induction hypothesis by noting that $COp_{p,j}$ and each of the operations in PN always commute. The only exception is $deq3f_p$ which needs an auxiliary lemma that $PEnq_q \mathbin{⨟} deq3f_p$ commutes with every operation in PN. This case crucially relies on the obvious invariant that there may be no more than one pending enqueue process for each array element.

The only difficult case is $enq1_p$ which adds a new pending enqueue process, and has to deal with a potential linearization point. To see what happens, consider the example shown in Fig. 8. It shows a situation on the left where an element a is in the array and process 2 is pending with element b, together with the possible observations (q, R) returned by the simulation B. Process 1 then executes $enq1_1(cs, cs')$ and becomes pending too with element c. This is shown on the right, together with the possible pairs (q', R') such that $B(cs', q', R')$.

The pairs (q, R) before the operation are exactly the subset of those pairs (q', R') where $ret(1, enq,) \notin R'$, i.e., the potential linearization point has not been executed. For this case simulation is trivial, choosing the empty sequence as $HBOp^*$. The difficult cases have $ret(1, enq,) \in R'$. As the last result with $q' = \langle a, c, b\rangle$ shows, the element c may be observed to be *not* the last element of the queue. This demonstrates that one linearization step with c is not sufficient on the abstract level. Instead the right choice for (q, R) is $(\langle a\rangle, \varnothing)$, and it is necessary to instantiate $HBOp^*$ to the sequential composition of both linearization steps, i.e. take $HBOp^*$ to be $Lin_{1,enq} \mathbin{⨟} Lin_{2,enq}$. This exploits the fact that the potential linearization of process 2 *may not have been executed*, and can still be executed after the one for process 1.

In general, the element c enqueued by some process p may be observed in any place behind the current elements of the array: we have $q' = q \frown \langle c\rangle \frown q_2$, where q_2 consists of elements only that pending enqueues will add in the future.

Defining $LinE_{\underline{r}} := Lin_{r_1,enq} \mathbin{⨟} \ldots \mathbin{⨟} Lin_{r_n,enq}$ to denote the sequence of linearization steps for a sequence $\underline{r} = \langle r_1, \ldots r_n\rangle$ of processes, the proof obligation for the case with $ret(1, enq,) \in R'$ for $enq1_p$ is strengthened to

$$\begin{aligned}&(enq1_p \mathbin{⨟} B)(cs, q', R') \wedge ret(p, enq,) \in R' \Rightarrow\\ &\qquad \exists\, q, q_2, R, \underline{r} \bullet B(cs, q, R) \wedge LinE_{\langle p\rangle \frown \underline{r}}((q, H, R), (q', H, R'))\end{aligned}$$

The abstract steps first linearize the enqueue of process p, and then may execute linearization steps for some other pending enqueue processes in $\underline{r}$. Again the proof follows the standard well-founded induction scheme over $<_B$. The difficult case occurs when unfolding B executes $PEnq_p$ for the same process p. This case requires another induction to prove that $enq1_p \mathbin{⨟} PEnq_p$ commutes with all operations PN. This works except for a new dequeue process that removes the element just added by process p, which can only happen for an empty array. In this case we finally have to prove the implication

$$\begin{aligned}&(\forall\, i \bullet AR(i) = empty) \wedge B(cs, q, R) \Rightarrow\\ &\qquad \exists\, \underline{r} \bullet B(\langle\,\rangle, outs(cs), cs) \wedge LinE_{\underline{r}}((\langle\,\rangle, H, outs(cs)), (q, H, R))\end{aligned}$$

It states that the observable queues for an empty array consist of some (or none) of the elements of pending enqueues (in any order). Although this fact is obvious, an attempt to prove the lemma with the standard induction scheme fails: after one *PEnq* has been executed, the array is nonempty again, so the induction hypothesis is not applicable. The necessary generalization is

$$\begin{array}{l} B(cs, q \frown \langle a\rangle, R) \Rightarrow \\ \qquad \exists\, p \bullet \quad ret(p, enq,) \in R \wedge a = lvf(p) \wedge pcf(p) = E2 \\ \qquad\qquad \wedge\; B(cs, q, R \setminus \{ret(p, enq,)\}) \\ \qquad \vee\; \exists\, n \bullet n < back \wedge AR(n) = a \wedge B(cs\{AR(n) \mapsto null\}, q, R) \end{array}$$

It says, that the last element a of any nonempty queue $q \frown \langle a\rangle$ returned by B for state cs is either an element that some pending enqueue wants to add or some array element $AR(n)$. In the first case the corresponding return event is in R, and B may also return q, by not executing this pending enqueue. In the other case, B will return (q, R), when calling it with the modified state $cs\{AR(n) \mapsto null\}$, where the array element has been removed. The proof of the generalized lemma is again by standard induction over $<_B$, the original lemma follows from the special case, where the array is empty.

This closes the last of our backward simulation proofs for operation $enq1_p$, implying linearizability of the Herlihy-Wing queue by our main Theorem 4.6.

7. THREAD-LOCAL PROOF OBLIGATIONS

Verification of the queue example is particularly difficult, since it is neither possible to define an abstraction function, nor to fix a specific instruction of the code as the linearization point of enqueue. However, it is easier to verify linearizability in many standard algorithms since, e.g., they allow one to define an abstraction function which simplifies the proof obligations.

For algorithms falling into simpler classes, using the general backward simulation in the verification is unnecessary complex, it is easier to specialize the proof obligations to the class first. Typically the specialization will exploit symmetries to derive *thread-local* proof obligations.

Linearizability as a notion is inherently global, however, for simpler algorithms one aims to find local more compositional proof obligations. In a sense the use of simulations is one aspect of locality — since they allow an operation-by-operation comparison. The other type of locality we aim for is *thread-local* by which we mean we can just consider one thread of the program — that is, the effect of *one process at a time*.

In [Derrick et al. 2011b] we have discussed thread-local proof obligations for such a class, and this section will explain the ideas necessary to prove that these conditions imply the global backward simulation in this paper (and therefore imply linearizability).

For thread-local conditions, the state cs is split into a global state gs and local states $lsf(p)$ for each process p (so $cs \mathrel{\widehat{=}} (gs, lsf)$). In our example gs consists of the queue q and the $back$ pointer, and $lsf(p)$ returns the tuple of values $kf(p), lback(p), lvf(p)$ and $pcf(p)$.

In many cases (but not in the one considered here), it is then possible to specialize the abstract states as, that must be related to (cs, h) to a single state, using an abstraction function abs that depends on the global state gs only (ignoring both the history h and the local states). Thus we will be able to write $abs(gs) = as$ as the abs is independent of h and lsf. The abstraction function is typically defined on states that satisfy an invariant $inv(gs, lsf)$[5]. If such an abstraction function abs exists, the backward simulation can be written as

$$BS((gs, lsf, h), (as, H, R)) \mathrel{\widehat{=}} inv(gs, lsf) \wedge abs(gs) = as \wedge h = H \wedge Rets(h, gs, lsf, R)$$

[5]formally, an invariant $ginv(gs)$ with only the global state as argument would be sufficient. To maintain such an invariant, however, it is usually necessary to have restrictions on the local states too.

using an auxiliary relation $Rets$, which determines the set of return events R for linearized processes.

Now whether an operation of process p has linearized or not, and what the return value will be often depends on the local state $lsf(p)$ of process p only: a standard case is, that the linearization point of an algorithm is a specific instruction, so just the local program counter is relevant. These are the cases when thread-locality is achieved as the behavior is not dependent on other processes. In this case $Rets$ can be defined as

$$Rets(h, gs, lsf, R) \mathrel{\widehat{=}} R = \bigcup_p ret_p(lsf(p))$$

where ret_p returns an empty set when the algorithm is before the linearization point (as determined by the program counter of p), and a set of one return event if it is after the linearization point.

A standard example where such a definition is possible is Treiber's stack [Treiber 1986], where linearization happens with the CAS (compare and swap) instruction that does the main modification of the data structure.

Our queue example allows such a definition for dequeue (but not for enqueue!). There, $ret_p(lsf(p))$ would return $\{ret(p, deq, lvf(p))\}$, when $pcf(p) \in \{D4, D6\} \wedge lvf(p) \neq null$, otherwise returning the empty set.

Note, that when both an abstraction function and a ret_p-function can be defined for all operations, BS becomes a (partial) function. And when a simulation relation is in fact a function (whether partial or total) the distinction between forward and backward simulation disappears and thus either set of proof obligations could be used (usually forward simulations are easier to verify).

More complex examples have operations with *potential* linearization points, like the *enqueue* operation of our running queue example. What makes this example particularly difficult is that linearization of *enqueue* modifies the abstract queue value that the concrete state represents. A much more common case of potential linearization points is when the abstract operation has no effect on the abstract state. Either the operation is one that always reads from the abstract data structure — the lookup operation of the Heller et al's [Heller et al. 2005] lazy set data structure, which is discussed in detail in [Derrick et al. 2011b] is one case — or the specific run just reads from the data structure. Michael and Scott's queue [Michael and Scott 1996] provides one example, where a potential LP is needed for the attempt to dequeue from an empty queue (which leaves the empty queue unchanged!).

For reading operations the potential linearization value often depends on the global state. Again, Heller et al's [Heller et al. 2005] lazy set is an example: the value returned by lookup is stored in the global heap.

Given potential and definite linearization points which depend on local state only, the natural definition of $Rets$ becomes

$$Rets(h, gs, lsf, R) \mathrel{\widehat{=}} \bigcup_p dret_p(gs, lsf(p), gs) \subseteq R \subseteq \bigcup_p pret_p(gs, lsf(p), gs)$$

where $dret_p(gs, lsf(p))$ gives the single return event for operation p, when it has definitely linearized. Similarly, $pret_p(lsf(p))$ gives the return event for process p if it has potentially or definitely linearized.

In [Derrick et al. 2011b] we have expressed the information computed by $dret_p$ and $pret_p$ with the help of a status function.

— $status(gs, lsf(p)) = IN(in)$ means that the operation has received input in and definitely not linearized, implying $dret_p(lsf(p)) = pret_p(lsf(p)) = \varnothing$.

— $status(gs, lsf(p)) = INOUT(in, out)$ is reserved for an operation that has potentially linearized and will return out, if the linearization becomes definitive. This implies $dret_p(lsf(p)) = \varnothing$ and $pret_p(lsf(p)) = \{ret(p, runs(pcf(p)), out)\}$, where $runs(pcf(p))$

is the index of the running operation that can be determined from the program counter $pcf(p)$.

— $status(gs, lsf(p)) = OUT(out)$ is reserved for operations that have definitely linearized, implying $dret_p(lsf(p)) = pret_p(lsf(p)) = \{ret(p, runs(pcf(p)), out)\}$.

With these definitions the symmetry between all processes can be exploited to reduce the backward simulation condition to a proof obligation that mentions the local states of two processes only. In the following we assume lsp and lsp' are the local state of one algorithm before and after executing (i.e. $lsf(p)$ and $lsf'(p)$ if the process is p), and lsq is the local state of another process q.

The (main) resulting thread-local proof obligation is then

$$\begin{aligned}
&\forall\, gs, gs' : lsp, lsp, lsq.\\
&LInv(gs, lsp) \wedge Linv(gs, lsq) \wedge D(lsp, lsq) \wedge COp_{p,j}(gs, lsp, gs', lsp')\\
&\Rightarrow\\
&\qquad LInv(gs', lsp') \wedge LInv(gs', lsq) \wedge D(lsp', lsq) \wedge\\
&\qquad LP_p(abs(gs), abs(gs')) \fatsemi LP_q(abs(gs'), abs(gs'))
\end{aligned}$$

This local proof obligation requires for each step $COp_{p,j}$ of an algorithm one has to prove a commuting diagram with up to two linearization points for just the two processes p and q considered. The component $LP_p(abs(gs), abs(gs'))$ in the obligation depends on changes of the status function. It abbreviates

— **skip** (i.e. no operation, $abs(gs)$ and $abs(gs')$ must be equal), when process p does not linearize, indicated by the status of p being unchanged $status(gs, lsp) = status(gs', lsp')$.

— $AOp_{p,i}(in, abs(gs), abs(gs'), out)$, if p potentially or definitely linearizes, observed by a status change from $IN(in)$ to $INOUT(in, out)$ or $OUT(out)$.

Again i is the index of the operation that process p runs currently. Of course, $LP_q(abs(gs'), abs(gs'))$ is defined in the same way, but is based on status changes from $status(gs, lsq)$ to $status(gs', lsq)$.

The history is completely absent from the local proof obligation, since the input values of pending invokes (these are needed as inputs for $HBOp^*$) are now available via the status function. Note that when the step of p linearizes another operation q, then the linearization step of q must be one that does not change the abstract $abs(gs')$ (the state is used twice in LP_q). This allows to instantiate $HBOp^*$ in the global proof obligation with the sequence of all processes q that are linearized by the step.

The global invariant $Inv(gs, lsf)$ has been specialized to

$$Inv(gs, lsf) \mathrel{\widehat{=}} \forall\, p : P \bullet (LInv(gs, lsf(p)) \wedge \forall\, q : P \bullet q \neq p \Rightarrow D(lsf(p), lsf(q)))$$

using a local invariant $LInv(gs, ls)$ that mentions one local state only, and a disjointness property $D(lsp, lsq)$. The latter are needed to express disjointness properties of local states (our queue example needs that two enqueue processes p and q never enqueue at the same position, i.e. $kf(p) \neq kf(q)$ as a disjointness property).

The derivation of the local proof obligation above (and variants, see e.g. [Travkin et al. 2012; Tofan et al. 2014]) could easily be mechanized (the effort being a few days of work). They are sufficient for many case studies, which exhibit a certain symmetry between the processes.

8. RELATED WORK

This paper uses a refinement-based approach to linearizability that defines operations on an abstract data structure that are different from the data structures used by the concrete algorithms. Other approaches (in particular the ones based on model checking, shape analysis or reduction as discussed below) ignore the data refinement aspect. They show that the concrete algorithms are equivalent to an abstract level that executes each algorithm atomically as one step, thus relegating the data refinement aspect to informal arguments.

Our extension of enhancing a data type with a history and finalization parallels the construction in [Woodcock and Davies 1996] that adds sequences of inputs and outputs to embed Z refinement into relational refinement, or the extraction of (action) traces for I/O automata. However, since we are in a concurrent setting, the events of a history (or the actions of an I/O automaton) additionally have to record which process executed which operation.

Earlier work on I/O automata already recognized that linearizability can be defined as an instance of refinement [Lynch 1996] (the book uses the term "atomic objects" for linearizability). Colvin, Groves et al [Doherty et al. 2004; Colvin et al. 2005] gave the first mechanized proofs for linearizability using forward and backward simulations for I/O automata.

Completeness results. Our work is, in essence, a completeness result showing that backwards simulations are all that are needed for a particular type of refinement (here one that will give us linearizability). A number of completeness results for different refinement settings exist, each defining an intermediate layer between the abstract and the concrete layer. For data refinement, [He Jifeng et al. 1986] proves, that forward and backward simulations are complete, using an intermediate layer, where states are sets of abstract states (powerset construction). The completeness result for refinement in a TLA-like setting given in [Abadi and Lamport. 1991] adds history and prophecy variables (specializing forward and backward simulations) to an intermediate layer. A similar result for I/O automata (which is closer to our setting) is proved in [Lynch and Vaandrager 1995], see their Theorem 5.6. The states of the intermediate level defined in both completeness proofs record the full history of *all concrete states.*

Finally, the completeness proofs of [Hesselink 2008] for the TLA setting and [Schellhorn 2008], [Schellhorn 2009] for a setting of ASMs also define intermediate layers, based on the concrete state. Both define a deterministic intermediate layer, that stores additional information ("eternity variables" and "choice functions") in the initial state to predict nondeterminism of the concrete layer.

Our completeness theorem differs from all of these completeness results, in giving a specialized intermediate level suitable for linearizability proofs. The intermediate level is closely related to possibilities as defined in [Herlihy and Wing 1990]. The definition given there is a rule-based definition of $Poss(as, P, R) \in Poss(H)$ where H is a concurrent history, P is the set of pending invokes of H, that have not linearized, and R is the set of return events for operations that have linearized. Our definition recasts the 4 rules given in [Herlihy and Wing 1990] (axioms S, I, C, R) as operations *HBInit*, *Inv* and *Lin* and *Ret* of *HBDT*, dropping the redundant set P. Our intermediate level is also related to the "canonical wait-free automaton for atomic objects" from [Lynch 1996], which has states that correspond to triples (as, P, R) (each process is in one of the states "before LP","after LP" "idle", corresponding to being in P, in R or in none of the two). However, as this automaton lacks H, it is not possible to avoid using both forward and backward simulations to verify linearizability.

Approaches to verifying linearizability. Since our work gives a general, applicable method for proving linearizability, it should be contrasted with other methods of proving linearizability.

First, there is work on model checking linearizability, e.g. [Cerný et al. 2010] for checking a specific algorithm or [Burckhardt et al. 2010] for a general strategy. These approaches are very good at finding counter examples when linearizability is violated. However, these methods only check short sequences of (usually two or three) operations by exploring all possibilities of linearization points, so these approaches do not give a full proof. They also do not yield any explanation of why a certain implementation is indeed linearizable.

Work on full proofs has analyzed several classes of increasing complexity, where figuring out simulations (in particular thread-local ones, that exploit the symmetry of all processes executing the same operations) becomes increasingly difficult.

The simplest standard class of algorithms can be verified in a refinement context with an abstraction *function*, and all linearization points can be fixed to be specific instructions of the code of an algorithm (often atomic compare-and-swap (CAS) instructions are candidates). A variety of approaches for verifying such algorithms have been developed: [Groves and Colvin 2007] uses I/O refinement and interactive proofs with PVS, [Vafeiadis 2007] executes abstract operations as "ghost-code" at the linearization point, arguing informally that linearizability is implied. Proof obligations for linearizability have also been verified using shape analysis [Amit et al. 2007].

Our own work in [Derrick et al. 2011a] gave step-local forward simulation conditions for this standard case. Conditions were optimized for the case where reasoning about any number of processes can be reduced to thread-local reasoning about one process and its environment abstracted to one other process. It mechanized proofs that these are indeed sufficient to prove linearizability.

A second, slightly more difficult class are algorithms where the linearization point is nondeterministically one of several instructions, the Michael-Scott queue ([Michael and Scott 1996]) being a typical example. [Doherty et al. 2004] has given a solution using backward and forward simulation, Vafeiadis [Vafeiadis 2007] uses a prophecy variable as additional ghost code. Our work here shows that backward simulation alone is sufficient.

A third, even more difficult class are algorithms that use observer operations that do not modify the abstract data structure. Such algorithms often have no definite linearization point in the code. Instead steps of *other* processes linearize. The standard example for this class is Heller et al's "lazy" implementation of sets [Heller et al. 2005]. There, the *contains* operation that checks for membership in the set has no definitive linearization point. Based on the idea that linearization of such operations can happen at any time during its execution, [Vafeiadis 2010] develops the currently most advanced automated proof strategy for linearizability in the Cave tool.

As an alternative approach, Lipton's reduction [Lipton 1975] can be used to verify examples of the first three classes. Manual proofs for the first two classes are ([Groves and Colvin 2009], [Groves 2008], [Jonsson 2012]), the only mechanized work we are aware of is [Elmas et al. 2010], which proves an example of the third class.

Our work in [Derrick et al. 2011b] gives thread-local, step-local conditions for this class, and verifies Heller et al's lazy set. Mechanized proofs that these conditions can be derived from the general theory given here are available on the Web too [KIV 2010]. In [Travkin et al. 2012] the example of [Elmas et al. 2010] has been verified with these proof obligations too.

All these three classes, where mechanized proofs have been attempted, had an abstraction function, so different possibilities for one concrete state could only differ in the possible linearization points that have been executed (our set R of return events). However, the Herlihy-Wing queue is just the simplest example that falls outside of these classes. We have chosen it here since it is easy to explain, not because it is practically relevant. Other example include atomic snapshot registers [Afek et al. 1993] and the multiset given in [Elmas et al. 2010], when a delete operation is added. We could recently verify this example using an embedding of the backward simulation technique into RGITL [Schellhorn et al. 2014] (a publication is currently in preparation).

One of the most important, practically relevant examples is the elimination queue [Moir et al. 2005], which to our knowledge is currently the most efficient lock-free queue implementation. This example has some striking similarities to the case study considered here. Verifying this case study is future work, however it seems clear that it can be verified using the same proof strategy as shown here.

For this most complex class only pencil-and-paper approaches existed before our work, so our proof of the Herlihy-Wing queue is the first that mechanizes such a proof (and even a full proof, not just the verification of proof obligations justified on paper) for this algorithm. Our proof is step-local in considering stepwise simulation. Recent work in [Henzinger et al. 2013] has verified proof obligations for the linearizability of the Herlihy-Wing queue automatically. However the proof obligations given are specific for queues, and the algorithm verified has to be transformed heavily before verification.

Even for simpler classes many proof approaches so far have resorted to global arguments about the past, either informally e.g. [Heller et al. 2005], [Michael and Scott 1996], [Vafeiadis et al. 2006], using explicit traces [O'Hearn et al. 2010] or with temporal past operators [Fu et al. 2010].

Herlihy and Wing's own proof in [Herlihy and Wing 1990] also uses such global arguments: first, it adds a global, auxiliary variable to the code. The abstraction relation based on this variable is not a simulation. Therefore they have to use global, queue-specific lemmas (Lemmas 11 and 12) about normalized derivations to ensure that it is possible to switch from one (q, R) to another (q', R') in the middle of the proof.

9. CONCLUSIONS

In this paper, we have presented a sound and complete proof technique for linearizability of concurrent data structures. We have exemplified our technique on the Herlihy and Wing queue which is one of the most complex examples of a linearizable algorithm. Except for pen-and-paper proofs no-one has treated this example before, in particular none of the partially or fully automatic approaches to proving linearizability. Both the linearizability proof for the queue and the general soundness and completeness proof for our technique have been mechanized within an interactive prover.

The proof strategy given here is complete, but still not optimal in terms of reduction of proof effort: in particular, we have to encode the algorithms as operations, and just like in Owicki-Gries style proofs we require specific assertions for every particular value of the program counter. Rely-Guarantee reasoning [Jones 1983] can help to reduce the number of necessary assertions and we have already developed an alternative approach based on Temporal Logic that used Relys and Guarantees. That approach can currently handle the standard class of algorithms for linearizability, though it has advantages for proving the liveness property of lock-freedom [Tofan et al. 2010] and has been used to verify the hard case-study of Hazard pointers [Tofan et al. 2011]. Integrating both approaches remains future work.

Our approach is also not fully optimal for heap-based algorithms, where the use of concurrent versions of separation logic with ownership (e.g. RGSep [Vafeiadis 2010] or HLRG [Fu et al. 2010]) helps to avoid disjointness predicates between (private) portions of the heap, and gives heap-local reasoning.

Finally, there is a recent trend to generalize linearizability to general refinement of concurrent objects [Filipovic et al. 2010], [Turon and Wand 2011], where the abstract level is not required to execute atomic abstract operations, or where the return values of operations are references, not values. We have not yet studied these theoretically interesting generalizations, since they are not needed for our examples. This – as well as techniques for optimizing our approach with respect to proof effort – is left for future work.

REFERENCES

Abadi, M. and Lamport., L. 1991. The existence of refinement mappings. *Theoretical Computer Science 2*, 253–284.

Abrial, J.-R. and Cansell, D. 2005. Formal Construction of a Non-blocking Concurrent Queue Algorithm (a Case Study in Atomicity). *Journal of Universal Computer Science 11,* 5, 744–770.

Afek, Y., Attiya, H., Dolev, D., Gafni, E., Merritt, M., and Shavit, N. 1993. Atomic snapshots of shared memory. *J. ACM 40,* 4, 873–890.

Amit, D., Rinetzky, N., Reps, T. W., Sagiv, M., and Yahav, E. 2007. Comparison under abstraction for verifying linearizability. In *CAV*. 477–490.

Banach, R. and Schellhorn, G. 2010. Atomic Actions, and their Refinements to Isolated Protocols. *FAC 22(1)*, 33–61.

Burckhardt, S., C.Dern, Musuvathi, M., and Tan, R. 2010. Line-up: a complete and automatic linearizability checker. In *Proceedings of PLDI*. ACM, 330–340.

Calcagno, C., Parkinson, M., and Vafeiadis, V. 2007. Modular safety checking for fine-grained concurrency. In *SAS 2007*. LNCS Series, vol. 4634. Springer, 233–238.

Cerný, P., Radhakrishna, A., Zufferey, D., Chaudhuri, S., and R.Alur. 2010. Model checking of linearizability of concurrent list implementations. In *CAV*. LNCS 4144, 465–479.

Colvin, R., Doherty, S., and Groves, L. 2005. Verifying concurrent data structures by simulation. *ENTCS 137*, 93–110.

Colvin, R. and Groves, L. 2005. Formal verification of an array-based nonblocking queue. In *ICECCS*. IEEE Computer Society, 507–516.

Colvin, R., Groves, L., Luchangco, V., and Moir, M. 2006. Formal verification of a lazy concurrent list-based set. In *CAV*. LNCS 4144 Series, vol. 4144. Springer, 475–488.

de Roever, W. and Engelhardt, K. 1998. *Data Refinement: Model-Oriented Proof Methods and their Comparison*. Cambridge Tracts in Theoretical Computer Science Series, vol. 47. Cambridge University Press.

Derrick, J. and Boiten, E. 2001. *Refinement in Z and Object-Z: Foundations and Advanced Applications*. Springer. http://www.cs.kent.ac.uk/pubs/2001/1200.

Derrick, J., Boiten, E. A., Bowman, H., and Steen, M. 1997. Weak refinement in z. In *ZUM*, J. P. Bowen, M. G. Hinchey, and D. Till, Eds. Lecture Notes in Computer Science Series, vol. 1212. Springer, 369–388.

Derrick, J., Schellhorn, G., and Wehrheim, H. 2008. Mechanizing a correctness proof for a lock-free concurrent stack. In *FMOODS 2008*. LNCS Series, vol. 5051. Springer, 78–95.

Derrick, J., Schellhorn, G., and Wehrheim, H. 2011a. Mechanically verified proof obligations for linearizability. *ACM Trans. Program. Lang. Syst. 33,* 1, 4.

Derrick, J., Schellhorn, G., and Wehrheim, H. 2011b. Verifying linearisabilty with potential linearisation points. In *Proc. Formal Methods (FM)*. Springer LNCS 6664, 323–337.

Doherty, S., Groves, L., Luchangco, V., and Moir, M. 2004. Formal verification of a practical lock-free queue algorithm. In *FORTE 2004*. LNCS Series, vol. 3235. 97–114.

Doherty, S. and Moir, M. 2009. Nonblocking algorithms and backward simulation. In *DISC*, I. Keidar, Ed. Lecture Notes in Computer Science Series, vol. 5805. Springer, 274–288.

Elmas, T., Qadeer, S., Sezgin, A., Subasi, O., and Tasiran, S. 2010. Simplifying linearizability proofs with reduction and abstraction. In *TACAS*. Springer LNCS 6015, 296–311.

Filipovic, I., O'Hearn, P. W., Rinetzky, N., and Yang, H. 2010. Abstraction for concurrent objects. *Theoretical Computer Science 411,* 51-52, 4379 – 4398.

Fu, M., Y. Li, Y., Feng, X., Shao, Z., and Zhang, Y. 2010. Reasoning about optimistic concurrency using a program logic for history. In *CONCUR*. Springer LNCS 6269, 388–402.

Groves, L. 2008. Trace-based derivation of a lock-free queue algorithm. *ENTCS 201*, 69–98.

Groves, L. and Colvin, R. 2007. Derivation of a scalable lock-free stack algorithm. *ENTCS 187*, 55–74.

Groves, L. and Colvin, R. 2009. Trace-based derivation of a scalable lock-free stack algorithm. *Formal Aspects of Computing (FAC) 21,* 1–2, 187–223.

He Jifeng, Hoare, C. A. R., and Sanders, J. W. 1986. Data refinement refined. In *Proc. ESOP 86*, B. Robinet and R. Wilhelm, Eds. LNCS Series, vol. 213. Springer-Verlag, 187–196.

Heller, S., Herlihy, M., Luchangco, V., Moir, M., III, W. N. S., and Shavit, N. 2005. A lazy concurrent list-based set algorithm. In *OPODIS 2005*. LNCS 3974. 305–309.

Henzinger, T., Sezgin, A., and Vafeiadis, V. 2013. Aspect-oriented linearizability proofs. In *CONCUR 2013: Proc. of Int. Conference on Concurrency Theory*. Springer-Verlag, Berlin, Heidelberg, 242–256.

Herlihy, M. and Wing, J. M. 1990. Linearizability: A correctness condition for concurrent objects. *ACM TOPLAS 12,* 3, 463–492.

Hesselink, W. H. 2007. A criterion for atomicity revisited. *Acta Inf. 44,* 2, 123–151.

Hesselink, W. H. 2008. Universal extensions to simulate specifications. *Information and Computation 206*, 106–128.

JONES, C. B. 1983. Specification and design of (parallel) programs. In *Proceedings of IFIP'83*. North-Holland, 321–332.

JONSSON, B. 2012. Using refinement calculus techniques to prove linearizability. *Formal Aspects of Computing 24,* 4-6, 537–554.

KIV 2010. Web presentation of linearizability theory and the lazy set algorithm. HTTP://WWW.INFORMATIK.UNI-AUGSBURG.DE/SWT/PROJECTS/POSSIBILITIES.HTML.

KIV 2011. Web presentation of KIV proofs for this paper. HTTP://WWW.INFORMATIK.UNI-AUGSBURG.DE/SWT/PROJECTS/HERLIHY-WING-QUEUE.HTML.

LIPTON, R. J. 1975. Reduction: a method of proving properties of parallel programs. *Commun. ACM 18,* 12, 717–721.

LYNCH, N. 1996. *Distributed Algorithms*. Morgan Kaufmann Publishers.

LYNCH, N. AND VAANDRAGER, F. 1995. Forward and Backward Simulations – Part I: Untimed systems. *Information and Computation 121(2)*, 214–233.

MICHAEL, M. M. AND SCOTT, M. L. 1996. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proc. 15th ACM Symp. on Principles of Distributed Computing*. 267–275.

MOIR, M., NUSSBAUM, D., SHALEV, O., AND SHAVIT, N. 2005. Using elimination to implement scalable and lock-free fifo queues. In *SPAA*. ACM, 253–262.

O'HEARN, P. W., RINETZKY, N., VECHEV, M. T., YAHAV, E., AND YORSH, G. 2010. Verifying linearizability with hindsight. In *29th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*. 85–94.

REIF, W., SCHELLHORN, G., STENZEL, K., AND BALSER, M. 1998. Structured specifications and interactive proofs with KIV. In *Automated Deduction—A Basis for Applications*. Vol. II. Kluwer, Chapter 1: Interactive Theorem Proving, 13 – 39.

SCHELLHORN, G. 2008. Completeness of ASM Refinement. *Electron. Notes Theor. Comput. Sci. 214*, 25–49.

SCHELLHORN, G. 2009. Completeness of Fair ASM Refinement. *Science of Computer Programming, Elsevier 76, issue 9*, 756 – 773.

SCHELLHORN, G., TOFAN, B., ERNST, G., PFÄHLER, J., AND REIF, W. 2014. RGITL: A temporal logic framework for compositional reasoning about interleaved programs. *Annals of Mathematics and Artificial Intelligence (AMAI)*, 1 – 44.

SCHELLHORN, G., WEHRHEIM, H., AND DERRICK, J. 2012. How to prove algorithms linearisable. In *CAV*, P. Madhusudan and S. A. Seshia, Eds. Lecture Notes in Computer Science Series, vol. 7358. Springer, 243–259.

TOFAN, B., BÄUMLER, S., SCHELLHORN, G., AND REIF, W. 2010. Temporal logic verification of lock-freedom. In *Proc. MPC 2010*. Springer LNCS 6120. 377–396.

TOFAN, B., SCHELLHORN, G., AND REIF, W. 2011. Formal verification of a lock-free stack with hazard pointers. In *Proc. ICTAC*. Springer LNCS 6916.

TOFAN, B., TRAVKIN, O., SCHELLHORN, G., AND WEHRHEIM, H. 2014. Two approaches for proving linearizability of multiset. *Science of Computer Programming Journal, Elsevier*, to appear.

TRAVKIN, O., WEHRHEIM, H., AND SCHELLHORN, G. 2012. Proving linearizability of multiset with local proof obligations. In *Proceedings of the 12th International Workshop on Automated Verification of Critical Systems (AVoCS)*, G. Lüttgen and S. Merz, Eds. Vol. 53. ECEASST.

TREIBER, R. K. 1986. System programming: Coping with parallelism. Tech. Rep. RJ 5118, IBM Almaden Research Center.

TURON, A. AND WAND, M. 2011. A separation logic for refining concurrent objects. In *POPL*. Vol. 46. ACM, 247–258.

VAFEIADIS, V. 2007. Modular fine-grained concurrency verification. Ph.D. thesis, University of Cambridge.

VAFEIADIS, V. 2010. Automatically proving linearisability. In *CAV*. Vol. LNCS 6174. Springer, 450–464.

VAFEIADIS, V., HERLIHY, M., HOARE, T., AND SHAPIRO, M. 2006. Proving correctness of highly-concurrent linearisable objects. In *PPoPP '06*. ACM, 129–136.

WOODCOCK, J. C. P. AND DAVIES, J. 1996. *Using Z: Specification, Refinement, and Proof.* Prentice Hall.