How to Prove Algorithms Linearisable

Gerhard Schellhorn[1], Heike Wehrheim[2], and John Derrick[3]

[1] Universität Augsburg, Institut für Informatik, 86135 Augsburg, Germany
schellhorn@informatik.uni-augsburg.de

[2] Universität Paderborn, Institut für Informatik, 33098 Paderborn, Germany
wehrheim@uni-paderborn.de

[3] Department of Computing, University of Sheffield, Sheffield, UK
J.Derrick@dcs.shef.ac.uk

Abstract. Linearisability is the standard correctness criterion for concurrent data structures. In this paper, we present a *sound* and *complete* proof technique for linearisability based on backward simulations. We exemplify this technique by a linearisability proof of the queue algorithm presented in Herlihy and Wing's landmark paper. Except for the manual proof by them, none of the many other current approaches to checking linearisability has successfully treated this intricate example. Our approach is grounded on *complete mechanisation*: the proof obligations for the queue are verified using the interactive prover KIV, and so is the general soundness and completeness result for our proof technique.

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 hashtables which allow for concurrent access by many processes at the same time. 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, or only lock small parts of the structure. This inevitably leads to race conditions. Indeed, the designers of such algorithms do not aim at race-free but at *linearisable* algorithms. Linearisability [14] requires that fine-grained implementations of access operations (e.g., insertion or removal of an element) appear as though they take effect "instantaneously at some point in time" [14], thereby achieving the same effect as an atomic operation.

Recently, a number of new approaches to proving linearisability have appeared, some supported by theorem provers (like our own), some automatic based on user-annotated algorithms and some manual (see Section 7). Looking at these 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.

In this paper, we propose a proof technique which can be used to prove linearisability of *every* linearisable algorithm: Our method is *sound and complete* for linearisability.

P. Madhusudan and S.A. Seshia (Eds.): CAV 2012, LNCS 7358, pp. 243–259, 2012.

The approach is based on *backward simulations* - a technique borrowed from data refinement. More precisely, we show that a fine-grained implementation is linearisable 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 linearisability is not new; however, current refinement-based approaches (e.g. [9]) are based on both backward *and* forward simulations. We exemplify our approach on the queue implementation of Herlihy and Wing [14]. None of the current other works on linearisability 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). Along with this queue example we also show how to systematically construct the backward simulations needed in the linearisability proofs.

Last but not least we have a complete mechanisation 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 [23]), *and* have verified within KIV that the general soundness and completeness proof of our technique is correct. In summary, this paper thus contains three contributions: (1) the proof of soundness and completeness of backward simulations for linearisability, (2) the linearisability proof for the Herlihy and Wing queue, and (3) the full mechanisation of both the example and the general theory.

The next section gives the algorithms for the example. Section 3 defines linearisability as a specific form of refinement. Section 4 gives our main theorem, that linearisability can always be proven with a backward simulation, and Section 5 derives one for the example, showing that this can be done systematically. Section 6 gives some information on the KIV prover, and sketches how the proof obligations for backward simulation could be verified. Full details of all proofs are online [17]. Section 7 gives related work and Section 8 discusses possible improvements and concludes.

2 Example

The queue of [14], which serves as our running example, is a data structure with two operations: *enqueue* appends new elements (of some type *T*) to the end of the queue and *dequeue* removes elements from the front of the queue. The implementation of the queue uses a shared array *AR* of unbounded length. All slots of the array are initialised with a value *null*, signalling '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. The pseudocode of the queue operations is as follows:

```
E0 enq(lv : T)          D0 deq(): T
E1 /* increment */      D1 lback := back; k := 0; lv := null;
   (k, back) :=         D2 if k < lback goto D3 else goto D1;
      (back,back+1);    D3 (lv, AR[k]) := (AR[k], lv);  /* swap */
E2 /* store */          D4 if lv ≠ null then goto D6 else goto D5;
   AR[k]:= lv;          D5 k := k + 1; goto D2;
E3 return               D6 return(lv)
```

The *enq* operation simply gets a local copy of *back*, increments *back* (these two steps are executed as an atomic "fetch & increment") and then stores the element to be enqueued in the array.

The *deq* operation proceeds in several steps: first, it gets a local copy of *back* and initialises 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 of the code are a loop consecutively visiting the array elements. At every position k visited, the array contents $AR[k]$ is swapped with variable *lv* (i.e., the assignment at D3 is executed in parallel). If the dequeue finds a proper non-*null* element this way ($lv \neq null$), this will be returned, otherwise the search 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.

The complete specification consists of a number of processes $p \in P$, each capable of executing its queue operations on the shared data structure. For the concrete implementation, therefore, these two algorithms can be executed concurrently by any number of processes - where the individual steps (i.e., the statements in locations $E0$ to $D6$) in the operations are taken to be atomic, but crucially can be interleaved. That is, a process may start an *enq* operation (say doing E0 and E1) but then another process may execute its own atomic step (e.g., start a *deq*). Verification that the concrete implementation is somehow correct with respect to abstract, atomic enqueue and dequeue operations is the crux of the problem and linearisability is the proof obligation.

Our proof of linearisability proceeds by showing that the concurrent implementation is a backward simulation of an atomic abstract specification of the queue, i.e. that every step of the implementation can be simulated by the abstract specification in a backward fashion. To this end, we phrase both abstract specification and implementation in terms of *data types*. A data type consists of a state *State* (set of variables) and operations on the state $Op \subseteq State \times State$ (e.g. enqueue or dequeue, or operations like $D1, D2, \ldots$). In addition, an initialisation operation $Init : State$ specifies constraints on the initial state and a finalisation operation $Fin \subseteq State \times F$ relates states to global result values from some set F. Intuitively, *Fin* fixes those parts of the state that we are interested in and want to observe when comparing the data types. A data type is written as

$$(State, Init, (Op_{p,i})_{p \in P, i \in I}, Fin)$$

Note that we have incorporated processes in here. We take a relational view on operations, and use $\fatsemi$ for composition of relations. Primed variables in operations refer to the after state. Sequences of operations are written as Op^*.

For the abstract queue (omitting *Fin* for the moment), we for instance have $A = (AState, AInit, Enq_{p \in P}, Deq_{p \in P})$ given by

$$AState \mathrel{\widehat{=}} [q : \mathrm{seq}\, T] \qquad Enq_p(x? : T) \mathrel{\widehat{=}} [q' = q \frown \langle x? \rangle]$$
$$AInit \mathrel{\widehat{=}} [q = \langle\,\rangle] \qquad Deq_p(x! : T) \mathrel{\widehat{=}} [x! = first(q) \wedge q' = rest(q)]$$

Here, the variable $x?$ is an input to and $x!$ an output of the operation.

The data type $C = (CState, CInit, (COp_{p,j})_{p \in P, j \in J})$ for the concurrent implementation is more complex. The state consists of the two global variables *back* : $\mathbf{N}$ and

$AR : \mathbf{N} \to T$ to represent the array with elements of type T. Additionally, the local variables of processes are part of the state, e.g. $lback : P \to \mathbf{N}$ represents the values of the local variable *lback* for all the processes. Finally, $pc : P \to \{N, E1, E2, E3, D1, \ldots, D6\}$ defines a program counter for all processes, $pc(p) = N$ means that process p is currently running no operation. Initial states $CInit : CState$ have an empty array, $back = 0$ and $pc(p) = N$ for all $p \in P$.

The operations of this concrete data type are made up of the steps of the algorithm: every line in the algorithm becomes one operation[1]. We thus for instance have an operation called $enq1_p$ (line E1 in the enqueue of process p) which is specified as

$$enq1_p \mathrel{\widehat{=}} [pc(p) = E1 \wedge pc'(p) = E2 \wedge k'(p) = back \wedge back' = back + 1]$$

Here, we use the convention of not mentioning variables of the state which remain unchanged. In a similar way we can define operations for all other steps of the algorithm.

Our goal for the next section is to show that all concurrent runs of the algorithm given here faithfully implement queue operations. E.g., a concrete run might start with the sequence $enq0_3 \fatsemi enq1_3 \fatsemi enq0_1 \fatsemi deq0_2$. Does this represent a possible implementation of an abstract run? Formally, we have to prove linearisability and we will do so by showing that the concurrent implementation is (a particular type of) refinement of the abstract atomic specification.

3 Linearisability and Refinement

Linearisability is defined by comparing *histories* created by the atomic queue operations and those created by the concurrent implementation. Histories are sequences of *inv*oke and *ret*urn events of particular operations (out of some index set I) by particular processes $p \in P$ with certain input or output values. For example, a possible history of our queue implementation is

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

In this history, process 3 first *invokes* an enqueue operation with argument a. Next, process 1 invokes an enqueue for element b. While these two processes are running, process 2 starts a dequeue, and process 4 invokes an enqueue of c. At the end, first process 3 *returns* from its enqueue and finally process 4. These histories are thus abstracting the algorithm into just its start and end given by the *invokes* and *returns* of the operations. In a *legal* history, a return event of process p from operation i is always preceded by a *matching* (i.e., corresponding) invoke event with the same p and i, while an invoke event may or may not be followed by a matching return. In the latter case, the operation has not yet finished, and the invoke is a member of the set $pi(h)$ of *pending invokes* of history h. For the given history $pi(h) = \{inv(1, enq, b), inv(2, deq,)\}$. In the following, *Event* denotes the set of all events, and we write *Ret* for the set of all return events.

The first step in our proof technique is to add the history created by the algorithms to the data types: We construct *history enhanced* data types collecting histories. The enhancement we define is not specific to the queue example but applies to all concrete

[1] In the KIV specification of the algorithm we split IF-statements into a true and false case.

and abstract data types for which we want to show linearisability. The history enhanced concrete data type $HC = (HCState, HCInit, (HCOp_{p,j})_{p\in P,j\in J}, HCFin)$ gets $HCState \mathrel{\widehat{=}} CState \wedge [h : Event^*]$. As in the example above, the invoking steps of operations op of processes p (like $enq0_p$ and $deq0_p$ for the queue corresponding to lines $E0$ and $D0$) add an event $inv(p, op, a)$ (where a is the input value of op) to the history. Similarly the returning operations add return events, all others leave the history unchanged. Now we can also define a meaningful finalisation operation: $HCFin \subseteq HCState \times F$ for $F = Event^*$ extracts the collected history by defining $HCFin((cs, h), H)$ iff $H = h$.

On the abstract data type we perform a slightly different form of enhancement which is also motivated by our objective of wanting to prove linearisability. Informally, linearisability means that all histories created by the implementation could also be produced by working with an abstract atomic queue. As Herlihy and Wing formulate it: we want the concurrent implementation to "provide the illusion that each operation ... takes effect instantaneously at some point between its invocation and return". This point in time is usually called the *linearisation point*. The formal definition given in [14], however, is based on comparing concurrent and sequential histories (where the latter are sequences of matching invocation and return pairs). Already [14] note that this definition is not suitable for proofs, therefore like most related work (see Section 7) we prefer an alternative definition that directly formalises the idea of a linearisation point. In the enhancement of the abstract data type $HA = (HAState, HAInit, \{Inv_{p,i}, Lin_{p,i}, Ret_{p,i}\}_{p\in P,i\in I}\}, HAFin)$ we thus add histories *plus* we also split operations in three: an invocation, a linearisation point and a return.

$$
\begin{aligned}
HAState &\mathrel{\widehat{=}} AState \wedge [h : Event^*, R : \mathbb{P}\, Ret] \\
HAInit &\mathrel{\widehat{=}} AInit \wedge [h = \langle\,\rangle \wedge R = \varnothing] \\
Inv_{p,i}(in? : In) &\mathrel{\widehat{=}} [(\neg\, \exists\, i', in' \bullet inv(p, i', in') \in pi(h)) \wedge \\
&\qquad as' = as \wedge R' = R \wedge h' = h \frown \langle inv(p, i, in?)\rangle] \\
Lin_{p,i} &\mathrel{\widehat{=}} [\exists\, in, out \bullet inv(p, i, in) \in pi(h) \wedge (\neg\, \exists\, out_2 \bullet ret(p, i, out_2) \in R) \wedge \\
&\qquad AOp_{p,i}(in, as, as', out) \wedge h' = h \wedge R' = R \cup \{ret(p, i, out)\}] \\
Ret_{p,i}(out! : Out) &\mathrel{\widehat{=}} [ret(p, i, out!) \in R \wedge h' = h \frown \langle ret(p, i, out!)\rangle \wedge \\
&\qquad R' = R \setminus \{ret(p, i, out!)\} \wedge as' = as] \\
HAFin &\mathrel{\widehat{=}} [H : Event^* \mid H = h]
\end{aligned}
$$

As we see here we do not only add a variable h collecting histories but also a variable R, a set of return events. The role of R is to collect return events for those operations which have already taken effect, i.e., which are past the linearisation point but have not yet returned. Abstract execution of operations now consists of three steps: the invocation operation $Inv_{p,i}$ just adds an invoke event to the history, the linearisation operation $Lin_{p,i}$ changes the state according to the original definition of the operation in A, adds a return event to R (now the effect has taken place) and keeps the history. The return operation $Ret_{p,i}$ adds the return event to the history and – now that it is present in h – has to remove it from R. Finalisation again gives the current history.

Note that HA is concurrent in the sense that operations of processes are interleaved. However, the "effect" operation Lin is still atomic and thus faithfully reflects the original abstract data type. These two abstractions HA and HC can thus be compared.

Definition 1. *The reachable states of HA are called* possibilities. *Writing HAOp for the union of all operations of HA, we define*

$$Poss(as, h, R) \mathrel{\widehat{=}} (HAInit \fatsemi HAOp^*)(as, h, R)$$

Our definition of possibilities is essentially the same as the one in Herlihy and Wing's paper [14], p. 486f. The notation given there is $(as, P, R) \in Poss(h)$, with a redundant set P, that contains those invokes in $pi(h)$ with no matching return in R. Axioms S, I, C and R correspond one-to-one to our operations *HAInit*, *Inv*, *Lin* and *Ret*: the premise and conclusion are the pre- and post-state of the operations (our side conditions guarantee legal histories in conclusions, left implicit in [14]).

Lynch [18], Sec. 13.1.2, gives a similar definition of linearisability using the "canonical wait-free automaton for atomic objects". States of this automaton are essentially (as, P, R) (P is called inv-buffer), traces of the IO automaton correspond to our history. Theorems 9 and 10 of [14] state that possibilities are equivalent to linearisability:

Theorem 1. *An implementation data type C is linearisable with respect to some abstract data type A if and only if for every history h created by C there exists a possibility* $Poss(as, h, R)$.

This theorem seems to be universally accepted, and informal arguments for its validity appear in many papers. However, to relate the results given here to the original definition of linearisability, we have mechanised the proof in KIV. The proof is rather complex. It shows that a forward simulation exists between the type *HA* given here and the abstract data type we have used in [7] for the original linearisability definition. This provided a hint that backward simulation could be a complete proof procedure for the definition given here.

The theorem gives us the option to prove linearisability by showing the existence of possibilities, which can be viewed as a form of a refinement, given next.

Definition 2. *Let* $A = (AState, AInit, (AOp_{p,i})_{p \in P, i \in I}, AFin)$ *and* $C = (CState, CInit, (COp_{p,j})_{p \in P, j \in J}, CFin)$ *be abstract and concrete data types respectively.*
A program *is a sequence of operation (indices)* $Prg = j_1 \ldots j_n$*; and running a program on the data type C gives the execution*

$$Prg(C) \mathrel{\widehat{=}} CInit \fatsemi COp_{j_1} \fatsemi \ldots \fatsemi COp_{j_n} \fatsemi CFin$$

C is a data refinement of *A, denoted*[2] $C \sqsubseteq A$*, if for all programs Prg,* $Prg(C) \subseteq AInit \fatsemi AOp^* \fatsemi AFin$*. An empty concrete program must refine the empty abstract program.*

Note that this is a very weak form of refinement as it assumes that the effect of a particular program in C can be achieved with some *arbitrary* program (AOp^*) in A. This is crucial for our approach since for complex linearisable algorithms – such as the one we consider in this paper – one concrete step in the implementation may correspond to the execution of several linearisation steps in the abstract data type. This type of refinement applied on the enhanced data types coincides with linearisability.

[2] Note that the literature on refinement usually writes the notation the other way round.

Theorem 2. *$HC \sqsubseteq HA$ iff C linearisable wrt. A.*

Proof: The concrete histories h are the values returned by finalisation of HC. Refinement implies that there is an abstract run which also produces h. This run reaches a state (as, h, R) in HA before finalisation, i.e. $Poss(as, H, R)$ holds, and linearisability follows by Theorem 1. On the other hand, if linearisability holds, and h is a concrete history, then there is a possibility $Poss(as, h, R)$ by Theorem 1, so refinement holds, since finalisation will give h. □

4 Proving Linearisability with Backward Simulation

Data refinement is the process of adding implementation detail to an initial abstract algorithm, and standard results show that forward and backward simulations are sound and jointly complete for verifying refinements (see [6] for an overview).

The fact that linearisability can be expressed in terms of refinement also underlies the work of Doherty, Groves et al. [9,12]. However, their work as well as many others assume that linearisability needs *both* backward and forward simulation to be complete (and, e.g., [9] uses both). Here, we show that in fact backward simulation alone is already complete for proving linearisability.

Definition 3 (Backward simulation). *Let $A = (AState, AInit, (AOp_{p,i})_{p\in P, i\in I}, AFin)$ and $C = (CState, CInit, (COp_{p,j})_{p\in P, j\in J}, CFin)$ be two data types. A relation $BS \subseteq CState \times AState$ is a* backward simulation *from C to A, denoted $C \preceq_{BS} A$, if the following conditions hold:*

- *Initialisation: $CInit \fatsemi BS \subseteq AInit$,*
- *Finalisation: $CFin \subseteq BS \fatsemi AFin$,*
- *Correctness: $\forall p \in P, j \in J \bullet COp_{p,j} \fatsemi BS \subseteq BS \fatsemi AOp^*$.*

The correctness condition is weaker than usual (to match the weak data refinement) in that it only requires a concrete operation to be matched by an arbitrary sequence of abstract operations. Note that BS is often called *abstraction relation*: given a concrete state one has to define what the possible corresponding abstract states are.

The main result of this paper is that backward simulations are sufficient and we can avoid forward simulations entirely when verifying linearisability. The proof relies on the following two observations.

Proposition 1. *([14], p. 487) 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 $HAOp^*((as_0, h_0, R_0), (as, h_0 \frown h, R))$.*

Proof: Simple induction over the number of operation executions 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. □

Proposition 2. *The reachable states (as, h, R) of HA satisfy an invariant called $retsforpis(h, R)$ (returns for pending invokes only) which says that all return events in R have a process p with a corresponding invoke event in $pi(h)$.*

Again the proof is by induction on the number of operation executions. This now lets us formulate and prove our main theorem which shows that backward simulation is sound and complete for linearisability.

Theorem 3. *Let C, A be a concrete and an abstract data type, and HC, HA their history enhancements as defined above. Then HC* $\preceq_{BS}$ *HA iff C linearisable wrt. A.*

Proof: The easy direction from left to right just combines soundness of backward simulation and Theorem 2. For the other direction, assume C is linearisable wrt. A. We define a relation BS as

$$BS((cs,h),(as,H,R)) \mathrel{\widehat{=}} h = H \wedge Poss(as,H,R) \\ \wedge (H = \langle\rangle \Rightarrow AInit(as))$$

and prove the three proof obligations which show backward simulation.

- For *Initialisation*, we must prove that $HCInit(cs,h)$ implies $HAInit(as,H,R)$ when BS holds. Since $h = \langle\rangle$, we have $H = h = \langle\rangle$ and $ASInit(as)$. It remains to show that $R = \varnothing$. This follows from Proposition 2, since $pi(\langle\rangle) = \varnothing$.
- *Finalisation* requires to find an abstract (as,H,R) for every (cs,h), such that BS holds. Since C is linearisable, there is a state (as,h,R) with $Poss(as,h,R)$. If h is nonempty, this state is already sufficient. Otherwise, state (as,h,R) was reached from an initial state $(as_0,h_0,R_0) \in HAInit$ with $h_0 = \langle\rangle$, $R_0 = \varnothing$ and $ASInit(as)$. Therefore we can choose $(as,H,R) := (as_0,h_0,R_0)$.
- For *Correctness*, assume that both $BS((cs',H'),(as',H',R'))$ and $HCOp_{p,j}((cs,H),(cs',H'))$ hold. We have to find (as,H,R) with $BS((cs,H),(as,H,R))$ and $HAOp^*((as,H,R),(as',H',R'))$. Now for all concrete operations H is a prefix of H': either $H' = H$ or $H' = H \frown \langle e\rangle$ for invoking and returning operations that add an event e. Prefix closedness of possibilities (Prop. 1) gives a reachable state (as,H,R) for the prefix H of H' with $HAOp^*((as,H,R),(as',H',R'))$. Again, if $H \neq \langle\rangle$, this state already satisfies BS. Otherwise, like for finalisation, we have to choose the initial state. □

The theorem gives a backward simulation which matches an invoke operation $COp_{p,j}$ to a sequence $Lin^* \fatsemi Inv_{p,abs(j)} \fatsemi Lin^*$, where $Lin = \bigcup Lin_{p,i}$. Similarly, return operations match a sequence of linearisation steps with one return in the middle (since only such sequences add the right event to H). Other steps are matched to an empty sequence of abstract steps.

Theorem 3 specialises general completeness results, which imply that backward simulations and history variables are jointly complete for data refinement (these can be adapted to our formalism from [1], or more directly from [19], Theorem 5.6). However, all general completeness proofs add history variables, which record the full history of *all concrete states*. Theorem 3 shows, that for linearisability, the only history variable ever needed is the history needed to define linearisability (i.e. possibilities) itself.

5 Backward Simulation for the Case Study

The theory given in the last section ensures that any linearisable algorithm can be verified using a backward simulation BS. However, it does not tell us how to find such a

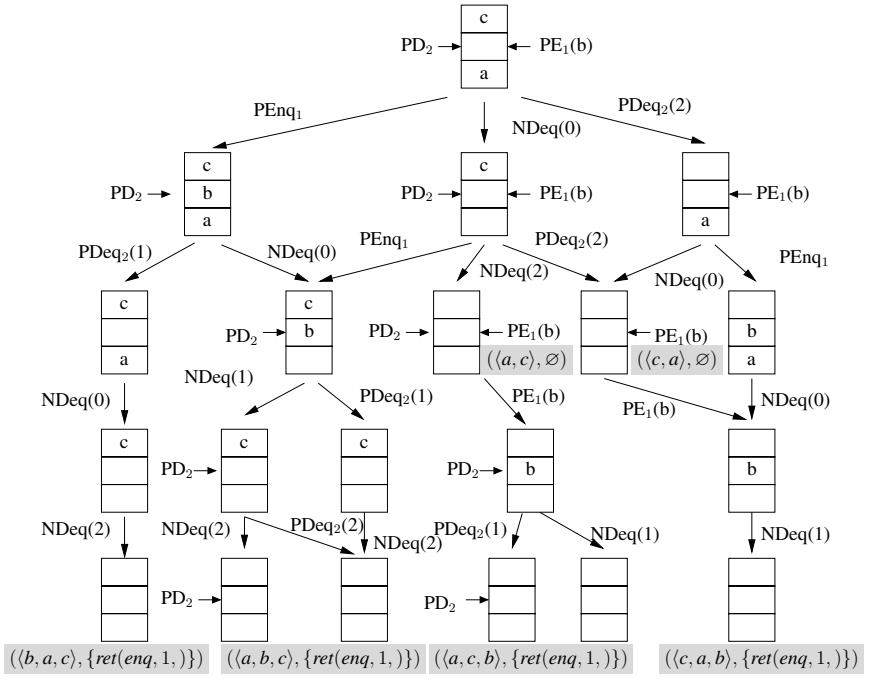

Fig. 1. Observation tree for an example state *cs*

relation between concrete states (cs, h) and abstract states (as, H, R). 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 *HA*. As a first observation, the finalisation condition requires $h = H$ and thus we can 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 is that for finding backward simulations one has to analyse the *observations made by future behaviours*.

To explain the approach consider the example state (the history for this state was given at the start of Section 3) *cs* shown at the top of Figure 1. The state shows a situation where the array has been filled with 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 $pc(1) = E2$, but then has been preempted. We call such an operation with $pc(1) = E2$ 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 initialised its *lback* ($pc \neq D1$), but has not yet successfully retrieved an element ($pc \neq D6$, and if $pc = D4$ then still $lv = null$).

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 contents 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 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.

We still have to determine the R-components B relates concrete states to. Recall that R collects linearisation 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 linearisability or concurrency, see [3]). For our case, we delay any linearisation 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 linearise at the time they swap the element: they have a *definite* linearisation point in the sense that we can attach it to line $D3$ when they swap a non-null element. However, enqueue operations cannot linearise in the future, since they would put the element in the wrong place in the queue. We find, that enqueue already *potentially* linearises when it executes $E1$, but only if the future run considered executes the operation to the end. In other runs, linearisation will happen when the element is actually inserted at line $E2$.

These considerations now help us towards defining B. We write $NDeq(n)(cs, cs')$[3] 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'. Similarly, we write $PDeq_p(n)(cs, cs')$ to say the same for an already running (pending) dequeue of process p, and finally $PEnq_p(cs, cs')$ for the completion of a pending enqueue. 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 $lv \neq null$).

[3] The web presentation [17] gives a formal definition.

- 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 1 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

KIV [23] 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 analyse 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 [17]. The completeness proof follows the proof given in Section 4, the difficult part is Theorem 1.

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.

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

$$(HCOp_{p,j} \fatsemi BS)((cs,H),(q',R',H')) \Rightarrow$$
$$\exists\, q,R \bullet BS((cs,H),(q,R,H)) \wedge HAOp^*((q,R,H),(q',R',H'))$$

A suitable sequence of abstract operations $HAOp^*$ 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 and $enq2_p$ the sequence is the linearisation step $Lin_{deq,p}$ resp. $Lin_{enq,p}$. 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. The choice of $HAOp^*$ simplifies the proof obligation to

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

The simplicity of the changes to the observation tree is then reflected by the simplicity of the proofs: they all are proven 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 $PN \in \{PDeq_q(n), PEnq_q, NDeq(n)\}$. The resulting goals can be closed immediately with the induction hypothesis by noting that $COP_{p,j}$ and PN always commute. The only exception is $deq3t_p$ which needs an auxiliary lemma that $PEnq_q \fatsemi deq3f_p$ commutes with every 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 difficult case is $enq1_p$ which adds a new pending enqueue process, and has to deal with a potential linearisation point. To see what happens, consider the example shown in Fig. 2. 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 linearisation point has not been executed. For this case simulation is trivial, choosing the empty sequence as $HAOp^*$. The difficult cases have $ret(1,enq,) \in R'$. As the last result with $q' = \langle a,c,b\rangle$ shows, the element

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

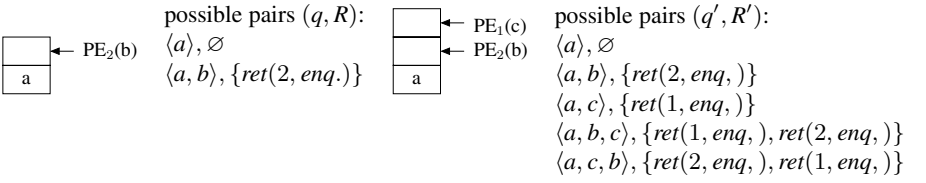

Fig. 2. Results of B before and after executing $enq1_2$

c may be observed to be *not* the last element of the queue. This demonstrates that one linearisation step with c is not sufficient on the abstract level. Instead the right choice for (q, R) is $(\langle a\rangle, \varnothing)$, and both linearisation steps $Lin_{1,enq} \fatsemi Lin_{2,enq}$ are necessary as $HAOp^*$. This exploits the fact that the potential linearisation 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 only consists of elements that pending enqueues will add in the future. Adding $\langle c\rangle \frown q_2$ corresponds to a sequence of abstract linearisation steps $LinE_{p,\underline{r}} := Lin_{p,enq}; Lin_{r_1,enq} \fatsemi \ldots \fatsemi Lin_{r_n,enq}$. For the last result of the example, $\underline{r} = \langle 2\rangle$ and $q_2 = \langle a\rangle$. Therefore we strengthen the proof obligation for $enq1_p$ to

$$(enq1_p \fatsemi B)(cs, q', R') \wedge ret(p, enq,) \in R' \Rightarrow$$
$$\exists\, q, q_2, R, \underline{r} \bullet B(cs, q, R) \wedge LinE_{p,\underline{r}}((q, R, H), (q', R', H))$$

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 $enq1_p \fatsemi PEnq_p$ commutes with all 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. We finally complete the proof of $enq1_p$ by showing that the observable queues for an empty array consist of some (or none) of the elements of pending enqueues (in any order).

7 Related Work

Our work gives a general and practically applicable method for proving linearisability. It should be contrasted with other methods of proving linearisability which fall into several classes.

First, there is work on model checking linearisability, e.g. [5] for checking a specific algorithm or [4] for a general strategy. These approaches are very good at finding counter examples when linearisability is violated. However, these methods only check short sequences of (usually two or three) operations by exploring all possibilities of linearisation points, so they do not give a full proof. They also do not yield any explanation of why a certain implementation is indeed linearisable.

Work on full proofs has analysed 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 has an abstraction *function*, and all linearisation 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 such algorithms have been developed: [12] uses IO-Automata refinement and interactive proofs with PVS, [27] executes abstract operations as "ghost-code" at the linearisation point, arguing informally that linearisability is implied. Proof obligations for linearisability have also been verified using shape analysis [2].

Our own work in [7] gave step-local forward simulation conditions for this standard case. Conditions were optimised 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 mechanised proofs that these are indeed sufficient to prove linearisability.

A second, slightly more difficult class are algorithms where the linearisation point is non-deterministically one of several instructions, the Michael-Scott queue ([20]) being a typical example. [9] has given a solution using backward and forward simulation, Vafeiadis [27] 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 linearisation point in the code. Instead steps of *other* processes linearise. The standard example for this class is Heller et al's "lazy" implementation of sets [13]. There, the *contains* algorithm that checks for membership in the set has no definitive linearisation point. Based on the idea that linearisation of such operations can happen at any time during its execution, [28] develops the currently most advanced automated proof strategy for linearisability in the Cave tool.

Our work in [8] gives thread-local, step-local conditions for this class, and verifies Heller et al's lazy set. Mechanised proofs that these conditions can be derived from the general theory given here are available on the Web too [16].

All these three classes, where mechanised proofs have been attempted, had an abstraction function, so different possibilities for one concrete state could only differ in the possible linearisation 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. One of the practically relevant examples is the elimination queue [21], 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 exactly the same proof strategy as shown here.

For this most complex class only pencil-and-paper approaches exist to proving linearisability, so our proof of the Herlihy-Wing queue is the first that mechanises 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. Even for simpler classes many proof approaches so far have resorted to global arguments about the past, either informally e.g. [13], [20], [29], using explicit traces [22] or with temporal past operators [11].

Herlihy and Wing's own proof in [14] 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 normalised derivations to ensure that it is possible to switch from one (q, R) to another (q', R') in the middle of the proof.

8 Conclusion and Future Work

In this paper, we have presented a sound and complete proof technique for linearisability 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 linearisable 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 linearisability. Both the linearisability proof for the queue and the general soundness and completeness proof for our technique have been mechanised 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 [15] 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 linearisability, though it has advantages for proving the liveness property of lock-freedom [24] and has been used to verify the hard case-study of Hazard pointers [25]. 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 (e.g. RGSep [28] or HLRG [11]) helps to avoid disjointness predicates between (private) portions of the heap, and gives heap-local reasoning.

Finally, there is a recent trend to generalise linearisability to general refinement of concurrent objects [10], [26], where the abstract level is not required to execute one abstract operation. We have not yet studied these theoretically interesting generalisations, since they are not needed for our examples. This – as well as techniques for optimising our approach with respect to proof effort – is left for future work.

References

1. Abadi, M., Lamport, L.: The existence of refinement mappings. Theoretical Computer Science 2, 253–284 (1991)
2. Amit, D., Rinetzky, N., Reps, T.W., Sagiv, M., Yahav, E.: Comparison Under Abstraction for Verifying Linearizability. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 477–490. Springer, Heidelberg (2007)
3. Banach, R., Schellhorn, G.: Atomic Actions, and their Refinements to Isolated Protocols. FAC 22(1), 33–61 (2010)
4. Burckhardt, S., Dern, C., Musuvathi, M., Tan, R.: Line-up: a complete and automatic linearizability checker. In: Proceedings of PLDI, pp. 330–340. ACM (2010)

5. Černý, P., Radhakrishna, A., Zufferey, D., Chaudhuri, S., Alur, R.: Model Checking of Linearizability of Concurrent List Implementations. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 465–479. Springer, Heidelberg (2010)
6. Derrick, J., Boiten, E.: Refinement in Z and Object-Z: Foundations and Advanced Applications. Springer (May 2001)
7. Derrick, J., Schellhorn, G., Wehrheim, H.: Mechanically verified proof obligations for linearizability. ACM Trans. Program. Lang. Syst. 33(1), 4 (2011)
8. Derrick, J., Schellhorn, G., Wehrheim, H.: Verifying Linearisability with Potential Linearisation Points. In: Butler, M., Schulte, W. (eds.) FM 2011. LNCS, vol. 6664, pp. 323–337. Springer, Heidelberg (2011)
9. Doherty, S., Groves, L., Luchangco, V., Moir, M.: Formal Verification of a Practical Lock-Free Queue Algorithm. In: de Frutos-Escrig, D., Núñez, M. (eds.) FORTE 2004. LNCS, vol. 3235, pp. 97–114. Springer, Heidelberg (2004)
10. Filipovic, I., O'Hearn, P.W., Rinetzky, N., Yang, H.: Abstraction for concurrent objects. Theoretical Computer Science 411(51-52), 4379–4398 (2010)
11. Fu, M., Li, Y., Feng, X., Shao, Z., Zhang, Y.: Reasoning about Optimistic Concurrency Using a Program Logic for History. In: Gastin, P., Laroussinie, F. (eds.) CONCUR 2010. LNCS, vol. 6269, pp. 388–402. Springer, Heidelberg (2010)
12. Groves, L., Colvin, R.: Derivation of a scalable lock-free stack algorithm. ENTCS 187, 55–74 (2007)
13. Heller, S., Herlihy, M., Luchangco, V., Moir, M., Scherer III, W.N., Shavit, N.: A Lazy Concurrent List-Based Set Algorithm. In: Anderson, J.H., Prencipe, G., Wattenhofer, R. (eds.) OPODIS 2005. LNCS, vol. 3974, pp. 3–16. Springer, Heidelberg (2006)
14. Herlihy, M., Wing, J.M.: Linearizability: A correctness condition for concurrent objects. ACM TOPLAS 12(3), 463–492 (1990)
15. Jones, C.B.: Specification and design of (parallel) programs. In: Proceedings of IFIP 1983, pp. 321–332. North-Holland (1983)
16. Web presentation of linearizability theory and the lazy set algorithm (2010), `http://www.informatik.uni-augsburg.de/swt/projects/possibilities.html`
17. Web presentation of KIV proofs for this paper (2011), `http://www.informatik.uni-augsburg.de/swt/projects/Herlihy-Wing-queue.html`
18. Lynch, N.: Distributed Algorithms. Morgan Kaufmann Publishers (1996)
19. Lynch, N., Vaandrager, F.: Forward and Backward Simulations – Part I: Untimed systems. Information and Computation 121(2), 214–233 (1995)
20. Michael, M.M., Scott, M.L.: Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In: Proc. 15th ACM Symp. on Principles of Distributed Computing, pp. 267–275 (1996)
21. Moir, M., Nussbaum, D., Shalev, O., Shavit, N.: Using elimination to implement scalable and lock-free fifo queues. In: SPAA, pp. 253–262. ACM (2005)
22. O'Hearn, P.W., Rinetzky, N., Vechev, M.T., Yahav, E., Yorsh, G.: Verifying linearizability with hindsight. In: 29th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC), pp. 85–94 (2010)
23. Reif, W., Schellhorn, G., Stenzel, K., Balser, M.: Structured specifications and interactive proofs with KIV. In: Automated Deduction—A Basis for Applications, Interactive Theorem Proving, vol. II, ch. 1, pp. 13–39. Kluwer (1998)
24. Tofan, B., Bäumler, S., Schellhorn, G., Reif, W.: Temporal Logic Verification of Lock-Freedom. In: Bolduc, C., Desharnais, J., Ktari, B. (eds.) MPC 2010. LNCS, vol. 6120, pp. 377–396. Springer, Heidelberg (2010)

25. Tofan, B., Schellhorn, G., Reif, W.: Formal Verification of a Lock-Free Stack with Hazard Pointers. In: Cerone, A., Pihlajasaari, P. (eds.) ICTAC 2011. LNCS, vol. 6916, pp. 239–255. Springer, Heidelberg (2011)
26. Turon, A., Wand, M.: A separation logic for refining concurrent objects. In: POPL, vol. 46, pp. 247–258. ACM (2011)
27. Vafeiadis, V.: Modular fine-grained concurrency verification. PhD thesis, University of Cambridge (2007)
28. Vafeiadis, V.: Automatically Proving Linearizability. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 450–464. Springer, Heidelberg (2010)
29. Vafeiadis, V., Herlihy, M., Hoare, T., Shapiro, M.: Proving correctness of highly-concurrent linearisable objects. In: PPoPP 2006, pp. 129–136. ACM (2006)