# Mechanizing a correctness proof for a lock-free concurrent stack

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

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

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

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

**Abstract.** Distributed algorithms are inherently complex to verify. In this paper we show how to verify that a concurrent lock-free implementation of a stack is correct by mechanizing the proof that it is linearizable, linearizability being a correctness notion for concurrent objects. Our approach consists of two parts: the first part is independent of the example and derives proof obligations local for one process which imply linearizabilty. The conditions establish a (special sort of non-atomic) *refinement relationship* between the specification and the concurrent implementation. These are used in the second part to verify the lock-free stack implementation. We use the specification language Z to describe the algorithms and the KIV theorem prover to mechanize the proof.

**Keywords:** Z, refinement, concurrent access, linearizability, non-atomic refinement, theorem proving, KIV.

## 1 Introduction

Locks have been used to control access by concurrent processors to shared objects and data structures. However, performance and other issues have led to the development of *lock-free* algorithms which allow multi-processors access to the data structures in a highly interleaved fashion. Such concurrent algorithms providing access to shared objects (e.g., stacks, queues, etc.) are intrinsically difficult to prove correct, and the down-side of the performance gain from using concurrency is the much harder verification problem: how can one verify that a lock-free algorithm is correct? This paper is concerned with the development of a theory that allows mechanized proof of correctness for lock-free algorithms.

As an example we use the lock-free stack from [**?**] which implements atomic push and pop operations as instructions to read, write and update local variables as well as the stack contents, the individual instructions being devised so that concurrent access can be granted. The only atomic operations are the reading and writing of variables and an atomic *compare-and-swap* (atomically comparing the

values of two variables plus setting a variable). Sequential notions of correctness, such as refinement [**?**,**?**], which rely on strict atomicity being preserved for all operations in an implementation have to be adapted to provide a correctness criteria for such a concurrent non-atomic setting. Linearizability [**?**] is one such criteria which essentially says that any implementation could be viewed as a sequence of higher level operations. Like serializability for database transactions, it permits one to view concurrent operations on objects as though they occur in some sequential order [**?**]:

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

Recent work on formalizing and verifying lock-free algorithms includes [**?**,**?**,**?**,**?**] as well as our own [**?**]. These show correctness by showing that an abstraction (or simulation or refinement) relation exists between the abstract specification and the concurrent implementation [**?**,**?**,**?**,**?**,**?**,**?**]. The proofs are manual or partly supported by theorem provers like, for instance, PVS.

All these papers argue informally that refinement implies the original linearizability criterion of [**?**]. Our work instead gives a formal theory that relates refinement theory and linearizability, which has been fully mechanized using the interactive theorem prover KIV [**?**]. A web presentation of the KIV proofs is available [**?**].

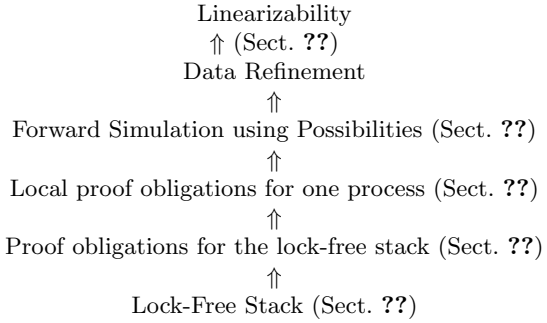


**Fig. 1.** Structure of the linearizability proof for the lock-free stack

Our methodology of proving linearizability consists of two parts: A generic part, shown in the upper half of Fig. **??** which derives proof obligations for one process. These proof obligations are shown to imply linearizability, and their genericity means they should be applicable to other algorithms in addition to the case study presented here. The second part is thus an application specific part (the lower half of Fig. **??**) which instantiates these proof obligations for our particular case study.

We start with the lower half, which extends our work in [**?**], where we used non-atomic refinement to verify linearizability. The specifications given there were expressed as a combination of CSP and Object-Z. We only considered two processes, one doing a *push* operation on the stack, and the other one a *pop* operation. In addition, we restricted the linearization point to always occur at the end of a sequence of operations implementing an atomic abstract operation. This allowed us to prove the correctness of a slightly simplified version of the algorithm given in [**?**].

Returning to this paper, in the next section we specify the stack and its lock-free implementation, given here as a Z specification rather than an integration of CSP and Object-Z (since this allows for simpler proof conditions). (The use of Z also allows the specification of several processes to be very elegantly captured in the specification using Z's notion of *promotion*, although we suppress this aspect here due to space restrictions.)

Section **??** then describes the background and the methodology that we have derived. It results in proof obligations that generalize the ones in [**?**] in two ways. First, we relax the assumption of having just two processes, and verify linearizability for an arbitrary numbers of processes. Second, we allow arbitrary linearization points. The proof obligations are applied on our running example in Section **??**.

The second half of our paper is concerned with the top half of Fig. **??**. In Section **??** we give a formal definition of linearizability and show that it can be viewed as a specific form of data refinement. Section **??** justifies our local proof obligations by constructing a global forward simulation using the possibilities employed in [**?**]. The last section concludes and discusses related work.

## 2 The Stack and its Lock-free Implementation

Our example stack and its lock-free implementation is based on that given in [**?**]. Initially the stack is described as a sequence of elements of some given type $T$ together with two operations *push* and *pop*. *push* pushes its input $v?$ on the stack, and *pop* removes one element that is returned in $v!$. When the stack is empty, *pop* leaves it and returns the special value *empty* (assumed to be not in $T$). The specification is given in Z [**?**,**?**], with which we assume the reader is familiar. The abstract stack specification is defined by:

*AS*

$$stack : \operatorname{seq} T$$

*AInit*

$$AS'$$

$$stack' = \langle\,\rangle$$

*push*

$\Delta AS$

$v? : T$

---

$stack' = \langle v? \rangle \frown stack$

*pop*

$\Delta AS; \quad v! : T \cup \{empty\}$

---

$stack = \langle\,\rangle \Rightarrow$
$\quad v! = empty \land stack' = stack$
$stack \neq \langle\,\rangle \Rightarrow$
$\quad v! = head\ stack \land stack' = tail\ stack$

The lock-free implementation uses a linked list of nodes which can be accessed by a number of processes. We use a free type to model linked lists so that a node is either *null* or consists of an element plus a further node (its successor in the list), we also define functions for extracting values from the nodes in a list:

$$Node ::= node\langle\langle T \times Node \rangle\rangle \mid null$$

$first : Node \nrightarrow T$

---

$\forall\, t : T; n : Node \bullet$
$\quad first\ node(t, n) = t$

$second : Node \nrightarrow Node$

---

$\forall\, t : T; n : Node \bullet$
$\quad second\ node(t, n) = n$

We will furthermore use a function $collect : Node \to \mathrm{seq}\, T$ later on, which collects all values of nodes reachable from some initial node in a list.

Informally a node consists of a value $val : T$ and a pointer *next*, which either points to the next node in the list or is empty and then has value *null*. A variable *head* is used to keep track of the current head of the list. Operations *push* and *pop* are split into several smaller operations, making new nodes, swapping pointers etc.. There is one operation atomically carrying out a comparison of values and an assignment: $CAS(mem, exp, new)$ (compare-and-swap) compares *mem* to *exp*; if this succeeds (i.e. *mem* equals *exp*), *mem* is set to *new* and *CAS* returns *true*, otherwise the *CAS* fails, leaves *mem* unchanged and returns *false*. In pseudo-code the push and pop operations that one process executes are given as follows (here, `head` refers to the head of the linked list):

```
push(v : T):
1  n:= new(Node);
2  n.val := v;
3  repeat
4     ss:= head;
5     n.next := ss;
6  until CAS(head,ss,n)
```

```
pop(): T:
1   repeat
2      ss:= head;
3      if ss = null then
4         return empty;
5      ssn := ss.next;
6      lv := ss.val
7   until CAS(head,ss,ssn);
8   return lv
```

Thus the *push* operation first creates a new node with the value to be pushed onto the stack. It then repeatedly sets a local variable *ss* to *head* and the pointer of the new node to *ss*. This ends once the final *CAS* detects that *head* (still) equals *ss* upon which *head* is set to the new node *n*. Note that the *CAS* in *push*

does not necessarily succeed: in case of a concurrent pop, *head* might have been changed in between. The *pop* is similar: it memorizes the head it started with in *ss*, then determines the remaining list and the output value. If *head* is still equal to *ss* in the end, the *pop* takes effect and the output value is returned.

Previously we used CSP to describe the orderings of the individual operations as given in the pseudo-code above. However, it is sufficiently simple that we use a program counter of the form either a number (1) or an $O$ or $U$ (for pop or push) followed by a number and give the ordering as part of the Z operations. Thus we get the following values for program counters:

$$PC ::= 1 \mid O2 \mid O3 \mid O5 \mid O6 \mid O7 \mid O8 \mid U4 \mid U5 \mid U6$$

We use a global state $GS$, that consists just of the head of the stack. This state is shared by all processes executing the algorithms of *push* and *pop* above:

*GS*

$$head : Node$$

*GSInit*

$$GS'$$

$$head' = null$$

Every process then possesses its own local state space (where local variables like `ss` and `n` reside) and the global shared data structure, i.e., the linked list characterised by its current head. The local state $LS$ of a particular process consists of the variables given in the pseudo-code above together with a program counter *pc*:

*LS*

$$sso, ssu, ssn, n : Node$$
$$pc : PC$$
$$lv : T \cup \{empty\}$$

*LSInit*

$$LS'$$

$$pc' = 1$$

To distinguish their use in *pop* and *push* we have appended a $u$, $o$, respectively, to variables *ss*. Each line in the pseudo-code is turned into a Z operation, where the numbering is according to line numbers in the pseudo-code.

First the operations are described in terms of their effect for one process, i.e. in terms of the state $GS$ and one state $LS$. We classify the operations into invoking operations (`INVOP`, the first operation in a *push* or *pop*) and return operations (`RETOP`). For instance *psh*2 is the invoking operation of *push*, constructing a new node.

*psh*2 [INVOP]

$$\Xi GS$$
$$\Delta LS$$
$$v? : T$$

$$pc = 1 \land pc' = U4$$
$$n' = node(v?, null)$$

*psh*4

$$\Xi GS$$
$$\Delta LS$$

$$pc = U4 \land pc' = U5$$
$$ssu' = head$$

*psh*5

$$\Xi GS$$
$$\Delta LS$$

$$pc = U5 \land pc' = U6$$
$$n' = node(first\ n, ssu)$$

$CAS_t psh$

$\Delta GS$ [RETOP]
$\Delta LS$

$pc = U6 \land pc' = 1$
$head = ssu$
$head' = n$

$CAS_f psh$

$\Xi GS$
$\Delta LS$

$pc = U6 \land pc' = U4$
$head \neq ssu$

As usual, the notation $\Xi$ specifies that the respective state space is unchanged, whereas $\Delta$ allows for modifications. However, we adopt the Object-Z (as opposed to Z) convention that all variables not mentioned in the predicate of a schema stay the same. This simply makes the specifications much more readable (otherwise we would have to add a lot of predicates of the form $ssu' = ssu$ etc.) and has no semantic consequence. Note that the only operation within *push* modifying the global data structure is $CAS_t psh$ (the succeeding CAS), which is also the return operation.

Similarly, we model *pop*. The operation *pop*2 which reads *head* at line 2 is duplicated, since it is called as an invoking operation (when $pc = 1$) as well as when the loop is iterated ($pc = O2$).

$pop2inv$

$\Xi GS$ [INVOP]
$\Delta LS$

$pc = 1 \land pc' = O3$
$sso' = head$

$pop2$

$\Xi GS$
$\Delta LS$

$pc = O2 \land pc' = O3$
$sso' = head$

$pop3t$

$\Xi GS$ [RETOP]
$\Delta LS$
$v! : T \cup \{empty\}$

$pc = O3 \land pc' = 1$
$sso = null \land v! = empty$

$pop3f$

$\Xi GS$
$\Delta LS$

$pc = O3 \land pc' = O5$
$sso \neq null$

$pop5$

$\Xi GS$
$\Delta LS$

$pc = O5 \land pc' = O6$
$ssn' = second\ sso$

$pop6$

$\Xi GS$
$\Delta LS$

$pc = O6 \land pc' = O7$
$lv' = first\ sso$

$CAS_t pop$

$\Delta GS$
$\Delta LS$

$pc = O7 \land pc' = O8$
$head = sso$
$head' = ssn$

$CAS_f pop$

$\Xi GS$
$\Delta LS$

$pc = O7 \land pc' = O2$
$head \neq sso$

$pop8$

$\Xi GS$ [RETOP]
$\Delta LS$
$v! : T \cup \{empty\}$

$pc = O8 \land pc' = 1$
$v! = lv$

Here, we find two return operations: a *pop* can return with output *empty* (operation *pop*3*t*) or with the effect of one node actually removed from the front of the linked list and its value returned as an output (operation *pop*8).

All these operations are defined for the local state. For the full scenario, it is assumed that each operation is executed by processes $p \in P$, then working on $GS$ and a local state $LS = lsf(p)$ returned by a function $lsf$ that stores all local states. A formal definition of this full scenario could be given in terms of promotion (see [**?**] or [**?**]), for reasons of space we only give a simple relational definition which also adds histories in Section **??** (histories are needed for linearizabilty).

We now have an abstract model of the stack, where *push* and *pop* occurs atomically, and an implementation with several processes operating concurrently on the linked list implementing the stack. The objective then is to show that this lock-free implementation is linearizable with respect to the initial abstract model. Technically, this is done showing a particular form of refinement relation between abstract and concrete specification.

## 3 The Refinement Methodology

To show that linearizability holds we take the local view of one process and define forward simulation conditions that show that the concrete implementation is a non-atomic refinement [**?**,**?**] of the abstract stack. The purpose of this type of non-atomic refinement is to show that the concrete system (with many small steps) resembles the abstract system with a smaller number of "larger" steps. We will argue informally, that the proof obligations are sufficient to guarantee linearizability, a formal justification will be given in Sections **??** and **??**.

In a standard (atomic) refinement, an abstract operation is implemented by one concrete operation, and to verify the correctness of such a transformation forward (and backward) simulations are used to describe how the abstract and concrete specifications proceed in a step by step fashion. The simulations are given in terms of an *abstraction relation* $R$ relating states of the concrete and abstract specification, and then one proves that from related states, every step of a concrete operation can be mimicked by a corresponding abstract operation leading to related states again. In the case of non-atomic refinement one abstract operation is now implemented by a number of concrete operations. To adapt the simulation conditions one requires that concrete steps are either matched by an abstract step or by no operation at all. One scenario of this kind is depicted in Figure **??**.

**Fig. 2.** Linearization in the middle

In the diagram, the upper level shows states and transitions of the abstract specification, the lower those of the concrete system. Dashed lines depict the abstraction relation $R$. (The labellings will be explained later). Starting from a pair of related states, a number of steps in the concrete system might abstractly

have no effect on the state, thus the abstraction relation $R$ stays at the same abstract state while changing the concrete. However, some concrete transitions (in this case the one in the middle) match a corresponding abstract operation, and the usual condition in simulations tells us that an abstract operation needs to be executed such that the after-states are again related. Matching with an abstract step has to take place whenever the current concrete operation has a visible effect, and in our setting such visible effects are the linearization points. Details of how the non-atomic simulation conditions are derived and examples of their use are given in [**?**,**?**].

For our purposes we tailor the simulation conditions to our particular application of concurrent algorithms. First of all, since our concrete specification is partitioned into a global state $GS$ together with local states $LS$ for each process, we will describe the abstraction relation $R$ as sets of elements $(as, gs, ls) \in R$ which we also write as $R(as, gs, ls)$. $R(as, gs, ls)$ then means that an abstract state $as$ is related to a global concrete state $gs$ and *one* local state $ls$ [1].

The second adaption concerns the processes themselves. Processes can have three different states: they can be idle ($IDLE$), have already invoked the implementation of an abstract operation possibly with some input $in$ ($IN(in)$) or they have already produced some output $out$ for this operation ($OUT(out)$). This gives rise to the following types:

$$STATUS ::= IDLE \mid IN\langle\!\langle T \cup \{empty\}\rangle\!\rangle \mid OUT\langle\!\langle T \cup \{empty\}\rangle\!\rangle$$

To verify the simulation conditions, we require that a function $status : LS \rightarrow STATUS$ has been defined. We furthermore require the concrete set of operations that implement one abstract operation `AOP` to be split into three classes: *invocations* `INVOP`, *returns* `RETOP` and other operations (here denoted `COP`), as is the case in our stack example above. The formal KIV specification has concrete operations $\{COp_j\}_{j \in J}$ with $J$ partitioned into $IJ$, $RJ$ and $CJ$, abstract operations $\{AOp_i\}_{i \in I}$, and a mapping $abs : J \rightarrow I$ to define which concrete operation implements which abstract operation. For better readability we drop indices in the proof obligations.

**Fig. 3.** Linearization on invocation or on return

The forward simulation conditions consist of six separate clauses: *Init, Invoke, Before Sync, After Sync, Return before sync* and *Return after sync*, which we describe in turn. In addition to the initialisation, these describe a number of different possibilities. First of all we have to say how invocations, returns and other operations behave. Second, we have to describe whether these operations match the abstract one (we call this a *linearization* with $AOP$ in the descriptions

[1] Note that we do not have all local states together in $R$.

below), or whether they should have no observable effect. This second aspect gives us the disjunctions in the conditions below.

First of all, **Init** is the standard initialisation condition, requiring that initial states of the two specifications are related:

**Init**

$$\exists\, as \in AInit \bullet R(as, gs, ls)$$

Like all following conditions, the formula is implicitly universally quantifies over all its free variables (here: $gs : GS$ and $ls : LS$). **Invoke** places a requirement on the operations marked as `INVOP`, which are the operations that begin a sequence of concrete operations corresponding to one abstract operation. In our example $psh2$ and $pop2inv$ are of type `INVOP`. Invoke requires that if a process is idle and an invoking operation is executed then one of two things happen. Either the concrete after states are linked to the same abstract states as before ($R(as, gs', ls')$, see diagram INV1 to the right of Figure **??**) and the status is moved from $IDLE$ to $IN$, or that linearization has already taken place (diagram INV2 left of Figure **??**), the process moves to its $OUT$ state and we must match this invoke operation with the abstract $AOP$:

**Invoke**

$$\begin{array}{l} R(as, gs, ls) \wedge status(ls) = IDLE \wedge INVOP(gs, ls, gs', ls', in) \\ \quad \Rightarrow \quad (R(as, gs', ls') \wedge (\forall\, lsq : LS \bullet R(as, gs, lsq) \Rightarrow R(as, gs', lsq)) \\ \qquad\quad \wedge status(ls') = IN(in)) \\ \qquad \vee\, (status(ls') = OUT(out) \\ \qquad\quad \wedge \exists\, as' : AS \bullet AOP(in, as, as', out) \wedge R(as', gs', ls') \\ \qquad\qquad\qquad\quad \wedge (\forall\, lsq : LS \bullet R(as, gs, lsq) \Rightarrow R(as', gs', lsq)) \end{array}$$

An additional condition $\forall\, lsq : LS \bullet R(as, gs, lsq) \Rightarrow R(as', gs', lsq)$ is the price to pay for interleaving processes compared to standard non-atomic refinement: the condition prevents that other processes with an unknown, arbitrary state $lsq$ are affected by the local operation.

**Before sync** describes what happens if a concrete $COP$ is executed before we have reached the linearization point. It has two cases: either we have still not linearized after the $COP$, or this is the linearization point (see Figure **??**), in which case we must match with the abstract operation $AOP$:

**Before Sync**

$$\begin{array}{l} R(as, gs, ls) \wedge status(ls) = IN(in) \wedge COP(gs, ls, gs', ls') \\ \quad \Rightarrow \quad (status(ls') = IN(in) \wedge R(as, gs', ls) \\ \qquad\quad \wedge (\forall\, lsq : LS \bullet R(as, gs, lsq) \Rightarrow R(as, gs', lsq))) \\ \qquad \vee\, (status(ls') = OUT(out) \\ \qquad\quad \wedge \exists\, as' : AS \bullet AOP(in, as, as', out) \\ \qquad\qquad\qquad\quad \wedge R(as', gs', ls') \\ \qquad\qquad\qquad\quad \wedge (\forall\, lsq : LS \bullet R(as, gs, lsq) \Rightarrow R(as', gs', lsq))) \end{array}$$

In our example, $psh4, psh5, CAS_f pop, pop3f, pop6, pop5, CAS_t pop$ are all of type **Before Sync**. A dual condition **After Sync** describes the effect of a concrete operation taking place after the linearization point:

**After Sync**

$$\begin{aligned} & R(as, gs, ls) \wedge status(ls) = OUT(out) \wedge COP(gs, ls, gs', ls') \\ & \quad \Rightarrow R(as, gs', ls') \wedge status(ls') = OUT(out) \\ & \qquad \wedge (\forall\, lsq : LS \bullet R(as, gs, lsq) \Rightarrow R(as, gs', lsq)) \end{aligned}$$

In our particular example, we do not have an operation that requires this condition.

The final two conditions describe the effect of the `RETOP` operations and there are two cases. We could either have linearized already (**Return after sync**, diagrams RAS in the figures) in which case the concrete operation should have no observable effect, but the output produced by `RETOP` has to match the current one in status (stored during a previous linearization), or we haven't already linearized, and **Return before sync** then says we must now linearize with the abstract operation $AOP$.

**Return before sync**

$$\begin{aligned} & R(as, gs, ls) \wedge status(ls) = IN(in) \wedge RETOP(gs, ls, gs', ls', out) \\ & \quad \Rightarrow \exists\, as' : AS \bullet AOP(in, as, as', out) \wedge R(as', gs', ls') \\ & \qquad\qquad \wedge status(ls') = IDLE \\ & \qquad\qquad \wedge (\forall\, lsq : LS : R(as, gs, lsq) \Rightarrow R(as', gs', lsq)) \end{aligned}$$

**Return after sync**

$$\begin{aligned} & R(as, gs, ls) \wedge status(ls) = OUT(out) \wedge RETOP(gs, ls, gs', ls', out') \\ & \quad \Rightarrow out' = out \wedge status(ls') = IDLE \wedge R(as, gs', ls') \\ & \qquad \wedge (\forall\, lsq : LS \bullet R(as, gs, lsq) \Rightarrow R(as, gs', lsq)) \end{aligned}$$

In our example, $CAS_t psh$ is of type **Return before sync**, whilst $pop8$ and $pop3t$ are of type **Return after sync**.

## 4 Application to the Stack Example

In this section we describe how the methodology of the previous section can be applied to our running example. This involves mainly working out the correct abstraction relation and the *status* function, and then verifying the conditions that we detailed above.

The representation relation is constructed out of several specific ones, detailing the particular local state at various points during the execution. In summary:

$$R = collect(head) = stack \wedge \bigvee\nolimits_{n \in PC} R^n \wedge pc = n$$

The overall $R$ is the disjunction over individual $R^n$s, for $n$ being some specific value of the program counter, conjoined with the predicate $pc = n$ and – the actual relationship between abstract and concrete state space – a predicate stating that collecting the values from `head` on gives us the stack itself. The individual $R^n$s capture information about the current local state of a process being at $pc = n$. For several $n$s no specific information is needed:

$$R^1 \;\widehat{=}\; AS;\ GS;\ LS \qquad\qquad R^{O2} \mathrel{\widehat{=}} R^1, R^{O3} \mathrel{\widehat{=}} R^1, R^{O8} \mathrel{\widehat{=}} R^1$$

For others, certain predicates on the local state are needed. For instance, $R^{U4}$ states the existence of a locally created node, or $R^{O6}$ the relationship between *sso* and *ssn*. There is no further specific relationship between abstract and concrete state known (except for $collect(head) = stack$, which is true everywhere). Even a previous read of *head* gives us no relationship to, e.g., *sso* since this may have been invalidated by other concurrently running processes.

$$R^{U4} \;\widehat{=}\; LS \mid \exists\, v, no' \bullet\ n = node(v, no')$$

$$R^{U5} \;\widehat{=}\; LS \mid \exists\, v, no' \bullet\ n = node(v, no')$$

$$R^{U6} \;\widehat{=}\; LS \mid \exists\, v \bullet\ n = node(v, ssu)$$

$$R^{O5} \;\widehat{=}\; LS \mid sso \neq null$$

$$R^{O6} \;\widehat{=}\; LS \mid sso \neq null \mid ssn = second\ sso$$

$$R^{O7} \;\widehat{=}\; LS \mid sso \neq null \mid ssn = second\ sso \mid lv = first\ sso$$

Furthermore we need the definition of *status*. The status of a local state essentially depends on the program counter, it determines whether a process is in state *IDLE*, *IN* or *OUT*, thereby determining the linearization points. Local variables $n$, *sso* and *lv* can be used to determine values for input and output.

$$status : LS \to STATUS$$

$$\begin{aligned}
&\forall\, ls \in LS \bullet \\
&\quad (ls.pc = 1 \Rightarrow status(ls) = IDLE) \\
&\quad (ls.pc \in \{U4, U5, U6\} \Rightarrow status(ls) = IN(first\ ls.n)) \\
&\quad (ls.pc = O3 \land ls.sso = null \Rightarrow status(ls) = OUT(empty)) \\
&\quad (ls.pc \in \{O2, O5, O6, O7\} \lor (ls.pc = O3 \land ls.sso \neq null) \\
&\qquad \Rightarrow status(ls) = IN(empty)) \\
&\quad (ls.pc = O8 \Rightarrow status(ls) = OUT(ls.lv))
\end{aligned}$$

For our example, the linearization point of the push algorithm is the last returning instruction $CAS_t psh$ (so all intermediate values of the program counter

within push have status $IN$). For the pop algorithm it is also the last instruction $pop8$ for the case of a nonempty stack. But when $pop2$ finds an empty stack in $head$ and therefore sets $sso'$ to $null$ this must already be the linearization point, since at any later point during the execution the stack might already be nonempty again. There $status$ is $OUT(empty)$ at $pc = O3$ and $sso = null$.

Finally, we look at the verification of one of our conditions. Here, we take a look at condition **Return after sync** and the case where the return operation is $pop8$. Thus, on the left side of our implication we have $R(as, gs, ls) \wedge status(ls) = OUT(out) \wedge pop8(gs, ls, gs', ls', out')$. Since $pop8$ is executed, we have $pc = O8$. Hence $status(ls) = OUT(lv)$. By definition of $pop8$, we furthermore get $gs = gs'$, $out' = lv$, $pc' = 1$. Hence $out' = out$ (first conjunct to be shown), $status(ls') = IDLE$ (follows from $pc' = 1$, second conjunct), and $R(as, gs', ls')$ holds by definition of $R^1$ ($pc' = 1$) and the previous validity of $collect(head) = stack$. Furthermore, from $R(as, gs, lsq)$ we get $R(as, gs', lsq)$ since $gs' = gs$. This can similarly be done for all conditions and all operations, but - as a manual verification is error prone - KIV was used for this purpose, which lead to several small corrections. For example, $R^{U4}$ and $R^{U5}$ originally contained (erroneously) the condition $\exists\, v \bullet n = node(v, null)$. Since the complexity of the linearizability proof is contained in the generic theory, and proof obligations are tailored to interleaved execution of processes, applying them is easy: specification and verification of the stack example needed two days of work.

## 5 Linearizability as Refinement

In this and the following section we show that our local proof obligations are sufficient to guarantee linearizability as defined in Herlihy and Wing's paper [**?**]. The process to do this is as follows: in this section we give two data types $ADT$ and $CDT$ with operations $\{AOp_{p,i}\}_{p\in P, i\in I}$ and $\{COp_{p,j}\}_{p\in P, j\in J}$. These are derived from the local operations $\{COp_j\}_{j\in J}$ and $\{AOp_i\}_{i\in I}$ used in the proof obligations by adding a second index $p \in P$ that indicates the process $p$ executing the operation. For the concrete level, $COp_{p,j}$ now works on a local state $lsf(p)$ given by a function $lsf : P \to LS$ instead of $LS$. We also add histories to the states and operations on both levels, which are lists of invoke and return events. This is necessary, since the formal definition of linearizability is based on a comparison of two histories created by the abstract and concrete level. By placing this criterion in the finalization of $ADT$ and $CDT$, we encode linearizability as a specific case of standard data refinement ([**?**], see [**?**] for the generalization to partial operations) between the two data types (the discrepancy between the sets of indices is bridged by function $abs$, as explained below).

Our proof obligations are then justified in the next section by showing that they imply a forward simulation $FS$ between $ADT$ and $CDT$. This is the most complex step in the verification, since the additional concept of possibilities is needed to define $FS$.

We start by defining the histories $H \in HISTORY$ that are maintained by the two data types. They are lists of invoke and return events $e \in EVENT$.

An event indicates that an invoke or return operation has taken place executed by a certain process with some input (invoke) or output (return). The formal definition is

$$EVENT ::= inv\langle\langle P \times I \times IN \rangle\rangle \mid ret\langle\langle P \times I \times OUT \rangle\rangle$$
$$HISTORY ::= seq(EVENT)$$

Predicates $inv?(e)$ and $ret?(e)$ check an event to be an invoke or a return. $e.p \in P$ is the process executing the event, $e.i$ denotes the index of the abstract operation to which the event belongs. For a history $H$, $\#H$ is the length of the sequence, and $H(n)$ its $n$th element (for $0 \leq n < \#H$). Executing operations adds events to a history. If an invoke operation $INVOP_j$ with input $in$ is executed by process $p$, it adds $inv(p, i, in)$ to the history, where $i = abs(j)$ is the index of the corresponding abstract operation as given by function $abs : J \to I$ of Section **??**. Using a function $lsf : P \to LS$ such that $lsf(p)$ is the local state of process $p$ we therefore define $COp_{p,j}$ to be the operation

$$\begin{aligned} &COp_{p,j}(gs, lsf, H, gs', lsf', H') == \\ &\quad \exists\, ls' \bullet \quad INVOP_j(in, gs, lsf(p), gs, ls') \\ &\qquad\qquad \wedge\, H' = H \frown \langle inv(p, abs(j), in)\rangle \\ &\qquad\qquad \wedge\, lsf' = lsf \oplus \{p \mapsto ls'\} \end{aligned}$$

Similarly, $COp_{p,j}$ for a return operation $RETOP_j$ adds the corresponding return event to the history. Other operations leave the history unchanged.

The histories created by interleaved runs of processes form *legal* histories only: a legal history contains *matching pairs (mp)* of invoke and return events, for operations that have already finished, and *pending invocations (pi)*, where the operation has started (i.e. the invoke is already in the history), but not yet finished. Corresponding formal definitions are

$$\begin{aligned} mp(m, n, H) == \quad & m < n < \#H \wedge H(m).p = H(n).p \wedge H(m).i = H(n).i \\ & \wedge\, \forall\, k \bullet m < k < n \Rightarrow H(k).p \neq H(m).p \end{aligned}$$

$$pi(n, H) == inv?(H(n)) \wedge \forall\, m \bullet n < m < \#H \Rightarrow H(m).p \neq H(n).p$$

$$\begin{aligned} legal(H) == \forall\, n < \#H \bullet \textbf{if } & inv?(H(n)) \textbf{ then } pi(n, H) \vee \exists\, m \bullet mp(n, m, H) \\ & \textbf{else } \exists\, m \bullet mp(m, n, H) \end{aligned}$$

Histories created by abstract operations are *sequential*: each invoke is immediately followed by a matching return. A predicate $seq(HS)$ determines whether $HS$ is a sequential history. Atomically executing $AOp_i$ by process $p$ adds both events to the sequential history. Therefore $AOp_{p,i}$ is defined as

$$\begin{aligned} &AOp_{p,i}(as, HS, as', HS') == \\ &\quad AOp_i(in, as, as', out) \wedge HS' = HS \frown \langle inv(p, i, in), ret(p, i, out)\rangle \end{aligned}$$

Linearizability compares a legal history $H$ and a sequential history $HS$. For pending invokes the effect of the operation may have taken place (this will correspond

to *status* = $OUT$ in our proof obligations). For these operations corresponding returns must be added to $H$. Assuming these returns form a list $H'$, all other pending invokes must be removed. Function *complete* removes pending invokes from $H \frown H'$. Formally we define

$$\begin{array}{l} \mathit{linearizable}(H, HS) == \\ \quad \exists\, H' \subseteq \mathit{ret?} \bullet \mathit{legal}(H \frown H') \wedge \mathit{seq}(HS) \wedge \mathit{lin}(\mathit{complete}(H \frown H'), HS) \end{array}$$

where *lin* requires the existence of a bijection $f$ between the $\mathit{complete}(H \frown H')$ and $HS$ that is order-preserving: for two matching pairs $mp(m, n, H)$ and $mp(m', n', H)$, if the first operation finishes before the second starts (i.e., $n < m'$), the corresponding matching pairs in $HS$ must be in the same order. This gives the following definitions

$$\mathit{lin}(H, HS) == \exists f \bullet \mathit{inj}(f, H, HS) \wedge \mathit{surj}(f, H, HS) \wedge \mathit{presorder}(f, H, HS)$$

$$\begin{array}{l} \mathit{inj}(f, H, HS) == \forall\, m < n < \#H \bullet \quad f(m) \neq f(n) \\ \qquad\qquad\qquad\qquad\qquad\qquad \wedge\, (mp(m, n, H) \Rightarrow f(n) = f(m) + 1) \end{array}$$

$$\mathit{surj}(f, H, HS) == \forall\, m < \#H \bullet f(m) < \#HS \wedge HS(f(m)) = H(m)$$

$$\begin{array}{l} \mathit{presorder}(f, H, HS) == \forall\, m, n, m', n' \bullet \\ \qquad mp(m, n, H) \wedge mp(m', n', H) \wedge n < m' \Rightarrow f(n) < f(m') \end{array}$$

Finally, we define a refinement between two data types $(\mathit{CInit}, \{\mathit{COP}_{p,j}\}, \mathit{CFin})$ and $(\mathit{AInit}, \{\mathit{AOP}_{p,j}\}, \mathit{AFin})$. Both initialization operations are required to set the history to the empty list. $\mathit{AOp}_{p,j}$ is $\mathit{AOp}_{p,abs(j)} \vee \mathit{skip}$ where $abs : J \to I$ maps the index of the concrete operation to the abstract operation it implements. The concrete finalization extracts the collected history and abstract finalization is the linearizability predicate:

$$\begin{array}{l} \mathit{AFin}(as, HS, H') == \mathit{linearizable}(H', HS) \\ \mathit{CFin}(gs, lsf, H, H') == H' = H \end{array}$$

With these finalization operations linearizability becomes equivalent to data refinement for the two data types.

## 6 Possibilities and Forward Simulation

Reasoning with the definition of linearizability as given in the previous section is rather tricky, since it is based on occurrences of events (i.e. positions in the history list). This is the reason why all work we are aware of on linearizability does *not* use this definition, but other definitions, for example, those based on IO automata refinement such as [**?**] argue informally that these imply linearizability. The problem was already noticed by Herlihy and Wing themselves, and they gave an alternative definition based on *possibilities*. Possibilities require a set of operations $\mathit{AOp}_i$ already, so they are less abstract than linearizability which only

uses events. The following rule set defines an inductive predicate $Poss(H, S, as)$[2] where $H$ is a legal history, $S$ is a set of returns that match pending invokes in $H$ (those pending invokes for which the effect has already occurred), and $as$ is an abstract state that can possibly be reached by executing the events in $H$ (hence the term 'possibility').

$$\frac{ASInit(as)}{Poss(\langle\rangle, \varnothing, as)}\ Init \qquad \frac{Poss(H, S, as) \quad \forall\, m.pi(m, H) \Rightarrow H(m).p \neq p}{Poss(H \frown \langle inv(p, i, in)\rangle, S, as)}\ I$$

$$\frac{Poss(H, S, as) \quad \exists\, m.pi(m, H) \wedge H(m) = inv(p, i, in) \quad AOp_i(in, as, as', out)}{Poss(H, S \cup \{ret(p, i, out)\}, as')}\ S$$

$$\frac{Poss(H, S, as) \quad ret(p, i, out) \in S}{Poss(H \frown \langle ret(p, i, out)\rangle, S \setminus \{ret(p, i, out)\}, as)}\ R$$

Rule *Init* describes the possible initial states: No event has been executed, the abstract state is initial, and the set of returns is empty. Rule $I$ allows to add an invoke event for process $p$, provided there is not already a pending one in $H$. Rule $S$ corresponds to linearization points: the effect of the abstract operation takes place (which requires a pending invoke), and the return is added to the set $S$ of returns. The return takes place in the last rule $R$, which adds the return to the history. Compared to the informal definition in [**?**], we had to add some explicit constraints which guarantee that all created histories are legal. Note that the rules for possibilities are a close match for the invoke–linearization point–return structure also present in our proof obligations.

One of the main tasks in formally justifying our proof obligations therefore is a proof that all possibilities are linearizable:

$$Poss(H, S, as) \Rightarrow \exists\, HS \bullet linearizable(H, HS)$$

Essentially this is Theorem 9 of [**?**]. For our own proof within KIV we needed to generalize the theorem to

$$\begin{aligned} &Poss(H, S, as) \Rightarrow \exists\, HS \bullet \forall\, H' \bullet \\ &\qquad eq(H', S) \Rightarrow legal(H \frown H') \wedge lin(complete(H \frown H'), HS) \end{aligned}$$

where $eq(H', S)$ is true, iff $H'$ is a duplicate free list that contains the same events as the set $S$. The proof is inductive over the number of applied rules. The case of the induction step, where rule $S$ is applied is the most complex, since both $complete(H \frown H')$ and $HS$ increase by adding a matching pair. Therefore the bijection between $H$ and $HS$ must change, which creates a large number of subcases to prove that the modified function is bijective and still preserves the

[2] The notation of the original definition is $(v, P, R) \in Poss(H)$. We use a predicate instead of a set, $as$ instead of $v$ for the abstract state and $S$ instead of $R$ to avoid confusion with the simulation relation. The parameter $P$ of the original definition is redundant: $P$ can be shown to be the set of pending invocations of $H$.

order of matching pairs. This proof, and the lemmas consumed, 10 days of the 15 days needed to do the KIV proofs for the whole case study.

Given that the existence of possibilities implies linearizability we are now able to justify our proof obligations. We provide a forward simulation $FS$, such that our proof obligations imply that $COp_{p,j}$ forward simulates $AOp_{p,abs(j)} \vee skip$ (where again, $abs : J \to I$ maps the index of the concrete operation to the one of the corresponding abstract operation). The definition of $FS$ is based on relation $R$ as used in our proof obligations and on possibilities

$$
\begin{array}{l}
FS(as, HS, gs, lsf, H) == \\
\quad (\forall\, p \bullet R(as, gs, lsf(p)) \\
\wedge \quad \{H(n) \bullet pi(n, H)\} \\
\quad = \{inv(p, i, in) \bullet runs(lsf(p)) = i \wedge status(lsf(p)) = IN(in)\} \\
\wedge \exists\, S \bullet \quad S = \{ret(p, i, out) \bullet runs(lsf(p)) = i \wedge status(lsf(p)) = out\} \\
\qquad\quad \wedge Poss(H, S, as) \\
\qquad\quad \wedge \forall\, H' \bullet (eq(H', S) \Rightarrow legal(H \frown H') \wedge lin(complete(H \frown H'), HS)
\end{array}
$$

This formula uses an auxiliary function *runs* defined on local states, that gives the index of the abstract operation, whose implementation is currently running[3]. It is a conjunction of three properties. The first requires that the relation $R$ holds for all local states of processes. The second is an invariant for the concrete data type: the pending inputs in the history correspond exactly to those implementations which run operation $i$ and have not yet passed the linearization point (i.e., status is $IN(in)$). The last conjunct gives the connection to possibilities. The set $S$ consist of those return events where a process $p$ is running operation $i$ and has reached status $OUT(out)$. The last line of the formula should be compared to the proof that possibilities imply linearizability above. There, a possibility implied the existence of a suitable $HS$ with this property. This $HS$ is now the $HS$ constructed by the corresponding abstract run.

The proof that the given formula is indeed a forward simulation is moderately complex. The proof of the main commutativity property for correctness splits into three cases for invoking, return and other operations. Condition **Invoke** is needed for the first case. Internal operations require **Before Sync** for the case where status is $IN(in)$, and **After sync** otherwise. Return operations similarly require **Return before sync** and **Return after sync** respectively.

## 7 Conclusion

In this paper we have considered the verification of correctness of a lock-free concurrent algorithm. In a state-based setting we have followed the approach of using simulations to show that a linearizability condition is met. The correctness proof involved several steps, showing first of all the existence of a particular simulation relation between concrete algorithm and abstract specification and furthermore, via a number of steps, proving that such a simulation relation

[3] In our example, *runs* indicates whether the *pc* of $p$ is within a *push* or *pop* operation.

implies linearizability. All proof steps have been mechanically checked using the theorem prover KIV.

Similar work on showing linearizability has been done by Groves and several co-authors [?,?]. In [?] specifications are written as IO-automata and linearization is shown using forwards and backwards simulation, and these were mechanized in PVS. The mechanization however only included the concrete case study. That simulation guarantees linearizability is argued but not mechanized.

Further work by Groves and Colvin includes [?], where they verify an improved version of an algorithm of Hendler et al. [?] which in turn extends the algorithm of [?] using a new approach based on action systems. This approach, like ours, starts with an abstract level of atomic push and pop operations. The approach uses a different proof technique than ours and their earlier work. Specifically, it works by induction over the number of completed executions of abstract operations contained in a history, and it is based on the approaches of Lipton [?] and Lamport and Schneider [?].

Additional relevant work in state-based formalisms includes [?], where the correctness of a concurrent queue algorithm using Event_B is shown. There, instead of verifying a given implementation, correctness is achieved by construction involving only correct refinement steps.

Another strand of relevant work is that due to Hesselink who has considered the verification of complex non-atomic refinements in the setting of a refinement calculus based notation. For example, in [?] he specifies and verifies (using PVS) a refinement of the lazy caching algorithm, where the model is not linearizable but only sequentially consistent. In [?] he recasts linearizability proofs as a refinement proof between two models which are verified by *gliding simulations* which allow the concrete model to do fewer steps than the abstract one if necessary.

The emphasis of our work was on the derivation of a generic, non-atomic refinement theory for interleaved processes and its mechanization. We have not yet considered the full complexity of the case study as done in [?] which adds memory allocation, modification counts to avoid the ABA problem and extends the algorithm by adding elimination arrays. These extensions remain as further work.

*Acknowledgements.* John Derrick and Heike Wehrheim were supported by a DAAD/British Council exchange grant for this work. The authors like to thank Lindsay Groves for pointing out an erroneous choice of linearization point in an earlier work and for many helpful comments when preparing this paper.