Electronic Communications of the EASST
Volume 53 (2012)

Proceedings of the
12th International Workshop on
Automated Verification of Critical Systems
(AVoCS 2012)

Proving Linearizability of Multiset with Local Proof Obligations

Oleg Travkin, Heike Wehrheim and Gerhard Schellhorn

15 pages

Guest Editors: Gerald Lüttgen, Stephan Merz
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Proving Linearizability of Multiset with Local Proof Obligations

Oleg Travkin[1], **Heike Wehrheim**[1] **and Gerhard Schellhorn**[2]

[1]Universität Paderborn, Institut für Informatik,
33098 Paderborn, Germany
[oleg82|wehrheim]@uni-paderborn.de
[2]Universität Augsburg, Institut für Informatik,
86135 Augsburg, Germany
[schellhorn]@informatik.uni-augsburg.de

Abstract: Linearizability is a key correctness criterion for concurrent software. In our previous work, we introduced local proof obligations, which, by showing a refinement between an abstract specification and its implementation, imply linearizability of the implementation. The refinement is shown via a thread local backward simulation, which reduces the complexity of a backward simulation to an execution of two symbolic threads. In this paper, we present a correctness proof by applying those proof obligations to a lock-based implementation of a multiset. It is interesting for two reasons: First, one of its operations inserts two elements non-atomically. To show that it linearizes, we have to find one point, where the multiset is changed instantaneously, which is a counter-intuitive task. Second, another operation has non-fixed linearization points, i.e. the linearization points cannot be statically fixed, because the operation's linearization may depend on other processes' execution. This is a typical case to use backward simulation, where we could apply our thread local variant of it. All proofs were mechanized in the theorem prover KIV.

Keywords: linearizability, refinement, multiset, concurrency, verification, KIV

1 Introduction

With an increasing number of cores per CPU, data structures like lists or sets can be used more and more concurrently. To avoid bottlenecks in a system, such data structures must be designed for maximizing throughput. This is done by applying fine-grained synchronisation schemes or avoiding the use of locks at all and using hardware primitives for synchronisation instead. Due to the fine granularity of synchronisation and interleaving of instructions, verifying such data structures to be correct is hard.

Reasoning about correctness of concurrent data structures also means reasoning about their linearizability [HW90]. Linearizability is a safety property. Data structure implementations are said to be linearizable, if for each concurrent execution there exists a sequential execution producing the same result. Operations of linearizable data structures seem to take effect instantaneously at some point in time between their invocation and response. This point is referred to as the linearization point (LP) or sometimes also as the commit point. The LP does not necessarily have to be a fixed location. Sometimes, the LP is even outside the performed operation, i.e. an operation is linearized by another concurrently running operation.

Proving a data structure to be linearizable usually involves showing a refinement relation between an abstract specification of a data structure and its actual implementation. An abstract specification contains atomic operations only. To show that an implementation is linearizable w.r.t. its abstract specification, one has to prove that the implemented operations non-atomically refine the operations from the abstract specification.

Several different approaches exist in order to prove linearizability, such as shape abstraction [ARR+07], rely-guarantee reasoning and separation logic combined [VP07] and our own simulation-based approach [DSW07]. In our approach, we formally defined a general theory relating refinement theory and linearizability following the notion of Herlihy and Wing [HW90]. From our general theory, we could derive proof obligations that are thread local and, once proven for a concrete data structure implementation, imply linearizability of the implementation. In our ongoing work, we aim to determine the class of algorithms and data structures, to which these local proof obligations can be applied.

As a case study for this paper, we selected a multiset implementation proposed by Elmas et al., which they verified linearizable [EQS+10]. In contrast to Elmas et al., our aim is to understand why it is linearizable. It is an interesting case study for several reasons: (1) Its operation *InsertPair* adds two elements to the multiset non-atomically. Thus, finding an abstraction *function* for a refinement proof may be challenging for such an operation as Elmas et al. already pointed out. It seems to be counter-intuitive in thinking of it to have exactly one location for the LP, since the point of element insertion is implemented by two instructions. (2) It includes an operation with an LP that is not statically fixed. In such cases usually a full backward simulation is required. We could avoid this by using our theory extension for potential linearization points [DSW11b]. (3) Another point to mention is that the multiset is blocking, i.e. critical variables are protected by locks. However, the implementation contains a writing instruction to a potentially locked variable. This property forced us to slightly modify our local proof obligation theory.

We used the interactive theorem prover KIV [RSSB98] to formalize the implementation, the abstract specification as well as to mechanize all our linearizability proofs. In Section 2, we introduce the multiset implementation, the encoding of the implementation and its abstract specification. Section 3 presents the abstraction function that we used for the refinement proofs. We give a brief overview of our local proof obligations in Section 4, followed by an explanation of the invariants used for the proofs in Section 5. In Section 6, we discuss why we modified our local proof obligations. Finally, Section 7 concludes.

2 Multiset as a Case Study

As a case study for our local theory, we chose an implementation of a multiset data structure of integers from Elmas et al. [EQS+10]. The parts of the implementation we considered in our proof are only the three operations presented in Figure 1 as Elmas et al. published it. A delete operation or an insert operation for single elements were not considered to limit the proof effort on the one hand and to gain results that can be compared on the other.

The multiset is implemented as an array containing elements of type *Slot*. A slot encodes its value by the integer attribute *elt*. A second attribute *stt* of type *Status* encodes the state of insertion. A slot may be in *empty*, *reserved* or *full* state. An integer *i* is supposed to be in the set,

```
enum Status = { empty,reserved,full };
record Slot { elt: int, stt: Status };
var M: array[0..N-1] of Slot

LookUp(x:int) returns(r:bool)
var i: int;
L1 for (i := 0; i < N; i++) {
L2  lock(M[i]);
L3  if (M[i].elt==x && M[i].stt==full) {
L4   unlock(M[i]);
L5   r := true; return;
L6   } else unlock(M[i]);
L7 } //i++ evaluation
L8 r := false; return;

atomic FindSlot(x:int) returns (r:int)
1 if (forall 0<=i<N. M[i].stt != empty) {
2  r := N;
3 } else {
4  assume (0<=r<N && M[r].stt==empty);
5  M[r].stt := reserved;
6 }
```

```
InsertPair(x:int, y:int) returns {r:bool}
 var i,j: int;
 I1 i := FindSlot(x);
 I2 if (i == N) {
 I3  r := false;
 I4  return;}
 I5 j := FindSlot(y);
 I6 if (j == N) {
 I7  M[i].stt := empty;
 I8  r := false;
 I9  return; }

 I10 M[i].elt := x;
 I11 M[j].elt := y;

 I12 lock(M[i]);
 I13 lock(M[j]);
 I14 M[i].stt := full;
 I15 M[j].stt := full;
 I16 unlock(M[i]);
 I17 unlock(M[j]);
 I18 r := true; return;
```

Figure 1: Multiset operations *LookUp*, *FindSlot* and *InsertPair* from [EQS$^+$10]

if there is a slot s with $s.elt = i \wedge s.stt = full$.

The *LookUp* operation tests whether some integer value is contained in the set. To do so, it traverses the array and tests (at location $L3$) each element whether it is equal to the parameter value and in *full* state. A slot is locked before the test is performed and released afterwards to prevent other threads from modifying the slot while it is tested.

Elements are inserted pairwise in the *InsertPair* operation. *InsertPair* calls the *FindSlot* operation to reserve empty slots. Actually, *FindSlot* is implemented similarly to *LookUp* by means of an array traversal, but here we rely on the algorithm by Elmas et al. [EQS$^+$10] for simplicity and assume it to be atomic. The *assume* statement in *FindSlot* blocks execution until the succeeding statement becomes true. Atomicity of *FindSlot* ensures that two concurrent *FindSlot* calls always reserve different slots. If *FindSlot* cannot find an empty slot to reserve, it returns with N as error value[1]. In this case, *InsertPair* returns with value *false* either at location $I4$ or $I9$. If the second *FindSlot* call fails to reserve an empty slot, the slot reserved during the first call is released at location $I7$. If two slots i,j could be reserved, the new values are assigned at $I10$ and $I11$. To accomplish insertion, both slots need to be in *full* state. Surrounded by lock statements, both slots get the *full* state assigned in $I14$ and $I15$. By unlocking the elements, changes are made visible to other threads.

2.1 Abstract Multiset

We start with formalizing the operations of the abstract data structure as well as its initialization. A multiset can contain several instances of an element. A function $count : \mathbb{Z} \times mset \to \mathbb{N}$ is used to count the instances of an element contained in an *mset*. Initialisation of the set s is done by setting it to an empty set, which we model by the predicate *ASInit*. We define the abstract

[1] We use N, not -1 as in [EQS$^+$10], just to avoid the extra type $\mathbb{N} \cup \{-1\}$ in KIV.

multiset as

$$\begin{aligned}
ASInit(s) &\Leftrightarrow s = \varnothing \\
LookUp(n, s, s', result) &\Leftrightarrow s' = s \wedge (result \Leftrightarrow n \in s) \\
InsertPair((x,y), s, s', result) &\Leftrightarrow \textbf{if } result \textbf{ then } s' = s \cup \{x, y\} \textbf{ else } s' = s
\end{aligned}$$

where *LookUp* and *InsertPair* model legal operation executions of the abstract specification. The first parameter is the operations input, i.e. a single integer value in case of a *LookUp* and a tuple of two integer values in case of *InsertPair*. Two parameters represent the multiset before (s) and after (s') operation execution completes. A boolean parameter *result* models the return value.

In case of *LookUp* the multiset remains unchanged: $s = s'$. The result is *true* iff parameter n is contained in the set s. Since the *InsertPair* operation may fail, we have two possible outcomes to consider. A successful execution ($result = true$) adds values x and y to the set s. In a failing execution the multiset remains unchanged.

2.2 Concrete Multiset

Since we want to show a refinement, we need a concrete specification besides the abstract one. Therefore, we go on with formalizing the multiset implementation and an arbitrary number of processes, which may execute operations *LookUp* or *InsertPair*, as a concrete state. For brevity, we use the Z notation to introduce the concrete implementation.

To keep our specification modular we divide the concrete state *CS* into a local and global part. The global state *GS* is represented by an array of slots *M*, where *N* is the array length. Each *Slot* stores a value of a multiset element in its attribute *elt* of type integer $\mathbb{Z}$. The insertion status is encoded by the attribute *stt*. An additional third attribute *lock* distinguishes between locked and unlocked slots. If the lock attribute *lock* stores p, then the *Slot* is locked by process p. Otherwise it stores *none*. Initially, the array is filled with *empty* unlocked slots holding 0, which we model by a predicate *GSInit*.

$$\begin{array}{|l}
\hline \textit{Slot} \\
elt : \mathbb{Z} \\
stt : \{empty, reserved, full\} \\
lock : (PId \cup \{none\}) \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\hline \textit{LS} \\
pc : PC \\
li, ini, inj : \mathbb{N} \\
lx, inx, iny : \mathbb{Z} \\
lr, inr : \mathbb{B} \\
pid : PId \\
\hline
\end{array}$$

$$\begin{array}{|l}
\hline \textit{CS} \\
GS \\
lsf : PId \rightarrow LS \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\hline \textit{GS} \\
M : [0..N\text{-}1] \rightarrow Slot \\
\hline
\end{array}$$

$$\begin{array}{|l}
\hline \textit{CSInit} \\
CS \\
\hline
GSInit \\
\forall p : PId \bullet\ lsf(p).pc = Idle \wedge lsf(p).pid = p \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\hline \textit{GSInit} \\
GS \\
\hline
\forall m : \mathbb{N} \bullet m < N \Rightarrow M[m] = (0, empty, none) \\
\hline
\end{array}$$

The local state *LS* is used to model process execution. Therefore, we need a program location variable *pc* of type *PC*. We use dot notation for modelling access to type attributes, e.g. *ls.pc* for the *pc* attribute. To distinguish between different processes, we also need a process identifier *pid* of type *PId*. Furthermore, the local state models all local variables of both operations. Local variables are prefixed with *l* for *LookUp* (resp. *in* for *InsertPair*). There are three input variables *lx*, *inx* and *iny* of type $\mathbb{Z}$ and two boolean output variables *lr* and *inr*. We use *li* in *LookUp* for array traversal. In *InsertPair*, *ini* and *inj* are used to store the index of a reserved slot. The latter three are of type $\mathbb{N}$, since an array index is never negative.

Finally, a full concrete state *CS* consists of the array modelled by GS and a local state function $lsf : PId \rightarrow LS$ that assigns a local state to each process. We initialize the concrete state by initializing the array via *GSInit* and assigning value *Idle* to all program counters *pc* in *CSInit*. A process at location *Idle* models an idle process. Interleaved runs are sequences $(cs_0, cs_1, \ldots cs_n)$ of concrete states starting with an initial state cs_0 that satisfies *CSInit*, such that all pairs (cs_k, cs_{k+1}) are related by one operation.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{InsertPair}0} \\
\hline
GS \\
\Delta LS \\
(x,y)? : \mathbb{Z} \times \mathbb{Z} \\
\hline
pc = Idle \land pc' = I1 \\
inx' = x \land iny' = y \\
\hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{InsertPair}15} \\
\hline
\Delta GS \\
\Delta LS \\
\hline
pc = I15 \land pc' = I16 \\
M' = M \cup \{inj \mapsto (M[inj].elt, full, M[inj].lock)\} \\
\hline
\end{array}$$

We describe a program by specifying one operation for each instruction, using Z notation. Each instruction may modify the local state and the global state. Control flow is encoded using a program counter that is part of the local state. Two examples are the steps *InsertPair0* and *InsertPair15*. The first step *InsertPair0* is the invocation of the operation with the input[2] tuple $(x,y)?$. The step can only be executed, if the current program counter is $pc = Idle$, i.e. not executing another operation. The next program counter by control flow is $pc' = I1$. Local variables inx' and iny' are initialized with the input. By convention, variables not mentioned are unchanged. Similarly, *InsertPair15* updates the program counter and modifies the slot at index *inj* to *full*, corresponding to the instruction in the code at line $I15$. Later, we use $insertPair15_p$ to denote the execution of *InsertPair15* by process *p* with $lsf(p)$ as its local state.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{InsertPair}1} \\
\hline
\Delta GS \\
\Delta LS \\
\hline
pc = I1 \land pc' = I2 \\
\exists n : \mathbb{N} \bullet ini' = n \land \\
\quad \textbf{if } n < N \\
\quad \textbf{then } M[n].stt = empty \land \\
\qquad M' = M \cup \{n \mapsto (M[n].elt, reserved, M[n].lock)\} \\
\quad \textbf{else } n = N \land M' = M \land \forall m : \mathbb{N} \bullet m < N \Rightarrow M[m].stt \neq empty \\
\hline
\end{array}$$

Since we rely on atomicity of *FindSlot*, we had to model its execution as a single instruction.

[2] Input variables in Z are suffixed with “?”, output variables are suffixed with “!”

The figure above shows the encoding of the first *FindSlot* call in line 1 of *InsertPair*, which is basically a decision between "there was a slot that could be reserved" or "there was no empty slot". In the first case $0 \leq n < N$ holds and the status of the slot indexed by n is changed to *reserved*. In the second case $n = N$ holds and the global state remains unchanged.

2.3 Challenges in showing Linearizability

In a simulation based approach for verification of linearizability like ours, we have to find an abstraction function $Abs : CS \to AS$ that maps each concrete state cs of the implementation to an abstract state as. In our case study, cs is basically an array and as is a multiset. Hence, we have to find a function that abstracts an array to a multiset representation. The abstract multiset has only atomic operations and we want to show that the concrete implementation is simulated by the abstract multiset. Therefore, execution of the concrete implementation must be either ignored by the abstract specification (a *skip* step) or both abstract and concrete steps must coincide (*linearization step*) in their execution[3]. In a third case (see Section 4), a process can pass its own LP and by doing that causing another process to pass its LP. A skip step must not change the abstract state. A linearization step changes the abstract state as if the whole operation was executed at once. There must be exactly one linearization step per operation invocation.

In our case study, the abstract multiset has to remain unchanged during an execution of *InsertPair* as long as the LP of the operation is not reached. After it is reached, the state changes immediately to a multiset containing both elements inserted.

One could use a naive abstraction function like:

$$count(x, Abs(M)) = |\, \{i \mid 0 \leq i < N \wedge M[i].elt = x \wedge M[i].stt = full\} \,|$$

where Abs is implicitly characterized by the function $count : \mathbb{Z} \times mset \to \mathbb{N}$. The function *count* counts the number of instances of a value in a multiset and hence uniquely characterizes a multiset. The multiset $Abs(M)$ contains exactly those elements x, for which there exists a slot holding the value x and that is in *full* state. As Elmas et al. already mentioned, this abstraction function is not useful. By choosing this function, the abstract multiset changes twice (in I14 and I15) during the execution of *InsertPair*. Hence, linearizability cannot be shown. One could think of adding a constraint to consider only those elements to be in the set, which are not locked, but this idea is even worse for two reasons. First, unlocking in *InsertPair* is still not atomic and just moves the problem to locations I16 and I17, where the inserted elements become visible to other threads. Second, *LookUp* would change the set during traversal of the array by locking and unlocking slots. See Section 3 for an abstraction function that can be used to show linearizability.

Another challenging property of the multiset implementation is its *LookUp* operation. It has no fixed LP in the code, since other threads may change the result of a *LookUp* call during their progress. For example, a *LookUp* call could test the set for an element x that is not contained in the set. Some other thread could insert x to a location, which was not tested yet. Before insertion the *LookUp* would have returned *false*, but afterwards it will return *true*. The LP of *LookUp* must be the last modification of the array that also has an influence on the result of *LookUp*. Obviously, such a location cannot be determined statically. In Section 4, we give an explanation of how local proof obligations can be used to overcome this problem.

[3] We sometimes refer to this as a process linearizes or passes its LP

Moreover, the implementation contains writing instruction to variables, which are potentially locked by other processes. This property turned out to be a limit to our local proof obligations as they were proposed in our previous work [DSW08]. However, a slight modification of our proof obligations (see Section 6) was sufficient to verify the case study linearizable.

3 Abstraction Function

As already mentioned in Subsection 2.3, in order to prove linearizability a naive characterization of when an element should be considered in the set is useless in this case. Although surrounded by locks, the *InsertPair* operation does not work atomically. So, the interesting question to answer is: How can we show that the obviously non-atomic execution of *InsertPair* changes the implemented set atomically? We do this by refinement. To show a refinement between the abstract multiset and the concrete array implementation, we have to find an abstraction function that abstracts the array state to a multiset state. Since, a multiset can contain several instances of the same element, its state is determined by the count of its elements. Hence, we have to define a *count* function to count instances of an arbitrary element i in the array M. As shown below, we take the sum of those slots, for which the predicate *In* is true. Therefore, we need to identify, when an element should be considered in the set and when it's not. This is the hardest part about the abstraction function.

$$count(i, Abs(M)) = \sum_{n=0}^{N-1} (\textbf{if } In(M,n,i) \textbf{ then } 1 \textbf{ else } 0)$$

$$\begin{aligned} In(M,m,i) \mathrel{\widehat{=}}\ & m < N \\ & \wedge M[m].elt = i \\ & \wedge M[m].stt = full \\ & \wedge notExLckRes(M,m,M[m].lock) \end{aligned} \qquad \begin{aligned} notExLckRes(M,m,pin) \mathrel{\widehat{=}}\ & (pin \neq none \\ & \Rightarrow \neg(\exists n.\, n \neq m \wedge n < N \\ & \wedge M[n].lock = pin \\ & \wedge M[n].stt = reserved)) \end{aligned}$$

As in a naive variant, the first part of our definition of *In* begins with: An element i is in the set, if its slot index is in the range of the array size N and the slot's *elt* attribute is holding the value i and the insertion status *stt* is *full*.

The predicate *notExLckRes* ensures that *InsertPair* changes the set atomically. It states that if a slot is locked by some process *pin*, then there is no other slot, which is locked by the same process and is in *reserved* state. By choosing this definition, it is made sure that a *LookUp* does not change the abstract multiset. It never has more than one locked slot and neither changes the element value nor its insertion state. During an execution of *InsertPair* the interesting lines are 14 and 15. When execution reaches line 14, both slots are reserved and the first slot is set to *full* status. In the naive abstraction function (see Subsection 2.3), the *full* slot's element i would be counted as a new element of the set. Hence, the abstract multiset would change before reaching the LP in line 15. The early change is avoided by the predicate *notExLckRes*. By reaching line 15 the second slot's status is set to *full*. Since there is no other slot locked by the currently executed process, which is also in *reserved* state, the *notExLckRes* predicate evaluates to *true* for both slots. Thus, by our implicit definition of *Abs* via *In*, the *InsertPair* operation changes the multiset atomically at location $I15$ and we have found a valid abstraction function to prove linearizability.

To the best of our knowledge, we are the first to provide an abstraction function that allows linearization of *InsertPair*. We chose line 15 as an LP, because it seemed closest to our intuition of reaching the LP, when both slots' status attributes become *full*. However, this abstraction function is only one out of several possible functions and it would be interesting to see, how different abstraction functions relate to the proof effort.

4 Local Proof Obligations and Status Function

This section summarizes our approach to verifying linearizability. The approach is based on the global refinement theory given in [SDW12], which views linearizability as a specific non-atomic refinement problem, where each run of an operation implements the corresponding abstract atomic operation with the same inputs and outputs. For verification of such refinements, backward simulation alone is complete, which is also a result of [SDW12].

Since an arbitrary number of processes has to be considered, finding a backward simulation that works on the full concrete state CS is difficult, but unavoidable in general. However, in most cases it is sufficient to look locally at one process p, and to abstract all the other processes to a single process q. Thereby, all quantification over processes is avoided. Informally, this is possible, if the steps of other processes all "look the same" to p, which holds for the case study. No matter which process locks a certain element, the influence on the behaviour of p is the same.

For such cases, the global proof obligations can be reduced to sufficient local proof obligations, as given in [DSW11b]. The main proof obligation, that has to be verified for all internal steps COp of an operation is

$$\begin{array}{l}\forall gs, gs' : GS, lsp, lsq, lsp' : LS \bullet \\ \quad INV(gs, lsp) \wedge INV(gs, lsq) \wedge D(lsp, lsq) \wedge COp(gs, lsp, gs', lsp') \\ \quad \Rightarrow \\ \quad INV(gs', lsp') \wedge INV(gs', lsq) \wedge D(lsp', lsq) \wedge AOP_{pq}(Abs(gs), Abs(gs'))\end{array} \qquad (LPO)$$

Invoking and returning steps, e.g. $InsertPair0$ and $InsertPair18$ have simpler proof obligations, which we omit here due to lack of space. The proof obligation uses a global state gs (here: the array M) and the two local states lsp and lsq of processes p and q. Predicate INV is used as a local *invariant* for each process, that must be reestablished after each step. Predicate $D(lsp, lsq)$ is also an invariant specifying *disjointness* properties for two processes, e.g. that they cannot lock the same array element. A definition of INV and D is outlined in Section 5. Abs is the abstraction function as defined in Section 3.

The main linearizability condition is that each concrete step COp of the operation must implement a sequence of zero, one or two abstract steps AOP_{pq}, that correspond to linearization points. This sequence is always a subsequence of $AOP_p(inp, as, as', outp) \mathbin{⨾} AOP_q(inq, as', as', outq)$ where AOP_p (and similarly AOP_q) is the abstract step corresponding to the operation executed by process p. If, e.g. $lsp.pc = I3$, then $AOP_p = InsertPair$. Inputs and outputs of AOP_p (resp. AOP_q) must agree with those of the concrete operation. We omit them in LPO due to lack of space.

The sequence to be executed depends on whether p or q or both pass their linearization point (LP) in the step of process p. If none of them passes the LP, then the step must implement *skip*, i.e. $AOP_{pq}(Abs(gs), Abs(gs'))$ simplifies to $Abs(gs) = Abs(gs')$. If both pass their LP, both abstract steps must be executed. Note, that q abstracts many processes. Hence, it is necessary, that their abstract step does not change the state $as' = Abs(gs')$. This is the main restriction for the applicability of our local proof obligations.

4.1 Status of Linearization

Whether or not one of the processes p or q passes its linearization point is determined for each of the processes separately, using a *status* function that must be defined for verification.

$$\begin{aligned}&status : GS \times LS \to STATUS \quad \text{where}\\ &STATUS = IDLE \mid IN(Input) \mid INOUT(Input \times Output) \mid OUT(Output)\end{aligned}$$

The value of $status(gs, lsf(p))$ determines, whether process p has passed the linearization point of the operation it runs. Status $IN(in)$ means that the operation was called with input *in* and it did not pass its LP yet, while status $OUT(out)$ means that the operation already passed its LP and will return *out*. Status *IDLE* is used for idle processes. Status *INOUT* is explained below.

The proof obligation (not shown) for invocation instructions with input *in* demands a status change from *IDLE* to $IN(in)$. Similarly, the proof obligation for returning instructions demands that status changes from $OUT(out)$ to *IDLE*, and that *out* is returned.

While an algorithm is running, the status changes once from $IN(in)$ to $OUT(out)$ at the linearization point. In this case, the corresponding abstract operation with the *same* input and output must be in the sequence AOP_{pq}. Note, that the LP of q is allowed to be a step of p and vice versa. If the status of p (resp. q) is unchanged, the step implements a *skip* step, i.e. no abstract operation for p (resp. q) is in AOP_{pq}. For our case study we define the status function by case analysis.

```
$status(gs, ls) \mathrel{\widehat{=}}$
    **if** $ls.pc = Idle$ **then** $IDLE$
    **else if** $ls.pc \in \{I1, I5, I10, I11, I12, I13, I14, I15\}$
        $\vee\ (ls.pc = I2 \wedge ls.ini < N) \vee (ls.pc = I6 \wedge ls.inj < N)$ **then** $IN(ls.inx, ls.iny)$
    **else if** $ls.pc \in \{I16, I17, I18\}$ **then** $OUT(true)$
    **else if** $ls.pc \in \{I3, I4, I7, I8, I9\} \vee (ls.pc = I2 \wedge ls.ini = N)$
        $\vee\ (ls.pc = I6 \wedge ls.inj = N)$ **then** $OUT(false)$
```

The cases fix the LP for *InsertPair* returning true to the step at $I15$. Figure 2 shows how the status changes in such a run. At the LP, the abstract operation *InsertPair* is refined, all other steps refine *skip*. The definition also fixes the LP for returning false to the steps at $I1$ and $I5$, where calls to *FindSlot* fail to find a free slot and return N.

For *LookUp*, the situation is more complex. Consider the two runs shown in Figure 3, where process q runs *LookUp* searching for element x. Assume that x is not in the array initially. The algorithm first executes steps *Lookup*0 and *Lookup*1, entering the loop. At this point, it

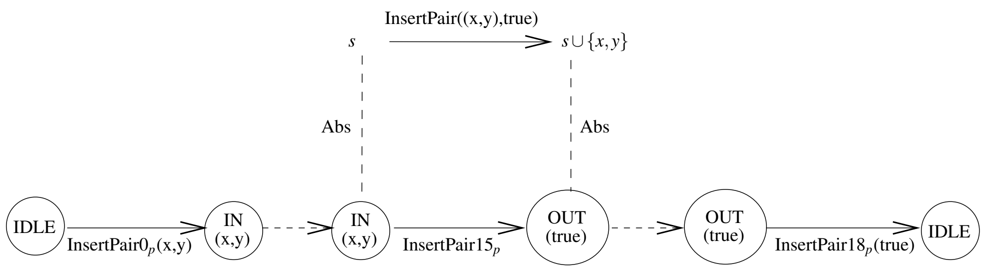

Figure 2: Status changes for *InsertPair* linearizing to true

has a chance to pass its LP, linearizing to false. We indicate this *potential linearization point* by a change of the status to $INOUT(x, false)$. In general, a change of status from $IN(in)$ to $INOUT(in, out)$ must implement the abstract operation $AOP(in, as, as', out)$ *as well as* a *skip* step. It must therefore leave the abstract state unchanged.

Status $INOUT(in, out)$ indicates, that there is a chance an operation already linearized to a specific value *out* earlier. Given a state with this status, further changes to $OUT(out)$ are allowed (fixing the decision to linearize), but also to $IN(in)$ (revoking the decision). In our proof obligation, steps with these status changes implement *skip*. From $INOUT(in, out)$ it is also possible to change the status directly to $OUT(out')$ or $INOUT(in, out')$ with a new linearized value out'. This change revises the decision to linearize to *out*, and immediately establishes a new (definitive or potential) linearization point. Both changes must implement the corresponding abstract operation, just as if the prior status has been $IN(in)$.

Assume the example run continues with *Lookup* checking some of the first array elements. If process q is preempted and another process p runs an $InsertPair(x, y)$ that passes its LP (i.e. executes *InsertPair*15) adding x to the array, two cases are possible: In the first case, x is added at a position in the array, that *LookUp* has already traversed. This case is shown as the lower of the two runs. In this case the old LP must be kept. The status does not change. Note that linearizing to false is now no longer possible, but the information about a potential LP is kept in the status. The lower run finally leaves the loop as if it has linearized before another process added element x. At that point it modifies the status to $OUT(false)$, and the proof obligation for the return operation ensures that indeed *false* is returned.

In the second case, process p adds element x at a position that *Lookup* still has to traverse. In this case, the decision of linearizing to *false* was wrong and must be revised. The upper of the two runs in Figure 3 visualizes this case, in which the status establishing a new *potential* LP changes to $INOUT(x, true)$. This allows more revisions of the status, but since we currently do not consider a *Delete* operation, which could remove x again, the status change could alternatively be directly to $OUT(true)$.

Note that for the upper run the status also changes for *InsertPair*15. Therefore, in the local proof obligation AOP_{pq} consists of an abstract step for both *InsertPair* and *Lookup*. When the run finally leaves the loop in *LookUp*, the status makes the decision of linearizing to a particular value permanent. The status function for *Lookup* is

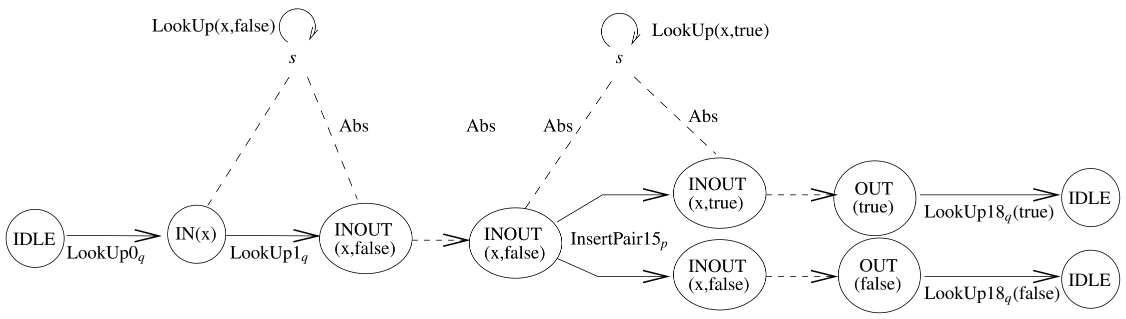

Figure 3: Status changes for *Lookup*

$status(gs, ls) \mathrel{\widehat{=}}$

$\quad$ **if** $ls.pc \in \{L1, L2, L3\}$ **then** $INOUT(ls.lx, \exists m \bullet ls.li \leq m < N \wedge In(M, m, ls.lx))$

$\quad$ **else if** $ls.pc \in \{L6, L7\}$ **then** $INOUT(ls.lx, \exists m \bullet ls.li < m < N \wedge In(M, m, ls.lx))$

$\quad$ **else if** $ls.pc \in \{L4, L5\}$ **then** $OUT(true)$

$\quad$ **else if** $ls.pc = L8$ **then** $OUT(false)$

Observe that when the status is $INOUT(in, out)$, the only meaningful value for *out* is easy to determine: It is the one that an operation would return, if it would run to completion from the current state without any interleaved steps from other processes.

5 Interference Freedom

In our refinement setting a process is executed symbolically. Each step of a process is interleaved with every possible step of another process. These processes may interfere during execution as they modify the global state. By interfering with each other, a process may invalidate another processes' invariant. An invariant *INV* describes what a process may rely on during its execution. To rely on its own *INV* one has to prove that a process does not conflict with the *INV* of any other process, i.e. it has to guarantee other processes' *INV*. This approach can be viewed as an adaption of the interference freedom conditions of Owicki and Gries [OG76]. A more detailed explanation of our adaption of interference freedom was published in [DSW11a].

5.1 Invariant

We split the *INV* into two predicates, where *LOCK-INV* is used to capture when a slot is locked or not and *ARRAY-INV* to capture other properties concerning the array. *LOCK-INV* models which slot is locked by a process at a certain program location. Since a process may have one, two or no locked slots during its execution, we used three different predicates to model these situations. The first and the last parameter of the predicates are the array and the process id. The parameter(s) in between represent the index of the locked slot(s). We identified five situations

$$INV(M, ls) \mathrel{\widehat{=}} LOCK\text{-}INV(M, ls) \wedge ARRAY\text{-}INV(M, ls)$$

$$\begin{aligned} LOCK\text{-}INV(M,ls) \mathrel{\widehat{=}} & \textbf{ if } ls.pc \in \{L3, L4, L6\} \textbf{ then } locksonly(M, ls.li, ls.pid) \\ & \textbf{ else if } ls.pc = I13 \textbf{ then } locksonly(M, ls.ini, ls.pid) \\ & \textbf{ else if } ls.pc = I17 \textbf{ then } locksonly(M, ls.inj, ls.pid) \\ & \textbf{ else if } ls.pc \in \{I14, I15, I16\} \textbf{ then } locksboth(M, ls.ini, ls.inj, ls.pid) \\ & \textbf{ else } locksnone(M, ls.pid) \end{aligned}$$

Note that program locations, where the lock gets acquired (e.g. *L*2) are not part of *LOCK-INV*. This is due to our transition like specification of program steps, where *INV* holds for the source state. For the same reason, slots are still locked at locations, where they actually get released.

$$\begin{aligned} ARRAY\text{-}INV(M,ls) \mathrel{\widehat{=}} & (ls.pc \in \{L2, \dots\} \Rightarrow ls.li < N) \\ & \wedge\ (validIni(ls) \wedge validInj(ls) \Rightarrow ls.ini \neq ls.inj) \\ & \wedge\ (ls.pc \in \{I5, \dots\} \Rightarrow M[ls.ini].stt = reserved) \\ & \wedge\ \dots \end{aligned}$$

The *ARRAY-INV* models all other properties that are related to the array *M* as a conjunction. Basically, it defines program location ranges, which imply some property, e.g. the index variable *ls.li* is smaller than the array length *N* or the slot indexed by *ini* must be in *reserved* state. We also modeled that the two slot indices *ini* and *inj* in *InsertPair* must be distinct, if reservation was successful. The range is determined by the conjunction of the predicates *validIni* and *validInj*. The full specification is available on our website [KIV12].

5.2 Disjointness

However, proving our proof obligations for non-interference may fail due to exhibiting situations between two processes that never occur in reality. As an example, consider the slot reservation of *FindSlot*. Two different processes would always reserve different slots and never the same. A proof may unfold a case, where two processes reserve the same slot. Such cases cannot be contradicted unless the specification forbids them explicitly. To overcome such cases, we use the disjointness predicate $D \subseteq LS \times LS$.

$$\begin{aligned} D(lsp, lsq) \mathrel{\widehat{=}} & (lsp.pid \neq lsq.pid \\ & \wedge\ (validIni(lsp) \wedge validIni(lsq) \Rightarrow lsp.ini \neq lsq.ini) \wedge \dots \\ & \wedge\ (li\text{-}locked(lsp) \wedge li\text{-}locked(lsq) \Rightarrow lsp.li \neq lsq.li) \wedge \dots) \end{aligned}$$

The first statement simply means that two different processes have different process ids. The following implications use program ranges as defined by some predicates (e. g. *validIni* or *li-locked*), which imply inequality of certain local variables. For instance, if two processes both perform a *LookUp* and both have their local variable *li* locked, their local values of *li* must be different. Otherwise, both processes would have locked the same slot. Due to lack of space, we omit parts of the disjointness specification and refer to our website for full specification [KIV12].

6 Verification of the Multiset

During a first attempt of proving the case study linearizable, we failed due to a slightly too weak precondition of our local proof obligations. In the following, we explain why we had to adapt our proof obligations. Finally, we summarize our verification results and discuss some experiences that we made while working with the theorem prover KIV.

6.1 Adaption of Local Proof Obligations

One of the challenges mentioned in Subsection 2.3 is the modification of potentially locked slots. For the case study, this may happen at program locations $I1$, $I5$, $I7$, $I10$ or $I11$, where the slots reserved may be locked by another process. We did not run into trouble with locations $I10$ and $I11$, but with the other locations. One reason for this is our abstraction function, which does not change the abstract multiset just by modifying a slot as long as the slot is still reserved.

At location $I7$, we found a case during our proofs, where the slot reserved by process p is also reserved and locked by another process q, which is also executing *InsertPair*. If q were at location $I14$ or $I15$, then the process p would linearize the other process by setting the slot from reserved to empty. Obviously, this cannot happen in reality and therefore should be avoided in the proof. Locations $I1$ and $I5$ have very similar characteristics to $I7$. Our attempt to prove (LPO) failed for $COp = InsertPair7$, since the precondition was too weak. The disjointness predicate $D(lsp, lsq)$ guarantees that p and q cannot reserve the same slot, but only if the local state $lsq = lsf(q)$ stores its *own* identifier $q = lsq.pid$. The latter was missing in our general theory, since local states were left unspecified. Hence, we could not use the disjointness predicate properly in this case.

However, the problem is repairable by adding process identifiers to local states and strengthening the precondition of (LPO). Since the local invariants $INV(gs, lsp)$, $INV(gs, lsq)$ and $D(lsp, lsq)$ are derived from a global invariant $INV(cs)$ for the full state $cs : CS$, defined by

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

it is possible to replace preconditions $INV(gs, lsq)$ and $D(lsp, lsq)$ with the stronger formula

$$\forall q : P \bullet q \neq lsp.pid \Rightarrow \exists lsq : LS \bullet INV(gs, lsq) \wedge D(lsp, lsq) \wedge lsq.pid = q$$

which is also implied by the global invariant. This adds a disjointness precondition to the process holding the lock, ruling out the situation described above.

6.2 Experiences and Results

The local proof obligation theory including its proofs of correctness is specified in the interactive theorem prover KIV [RSSB98]. Hence, we used KIV to instantiate proof obligations for the multiset case study and to mechanize our proofs. Overall, we needed to prove 45 lemmas within approximately 11000 proof steps. Less than 10% of the steps were done manually and most of them were quantifier instantiations. Some of the proofs were quite interaction intensive, because of the universal quantifier in the definition of the *In* predicate. Whenever we had to reason about one slot being modified, the *In* predicate forced us to reason about other slots, because slots with $stt = full$ can be in or outside the multiset and do not have to be modified directly to change this.

Most of the effort for specification and proofs was spent by our first author, who used this case study as an introduction to KIV and to the linearizability theory. It took him two month to get familiar with KIV and to create a first specification of the case study. Another month and two further major revisions (including adaption of theory) were necessary to complete the proofs.

7 Conclusion

In this paper, we presented a part of a multiset implementation, which we could successfully verify linearizable using our approach of local proof obligations. The multiset implementation is interesting, because it implements an obviously non-atomic insertion of two elements to a set. Finding an adequate abstraction function to show its linearizability is counter-intuitive, since the LP has to be fixed to a single instruction. We presented an idea, which can also be applied to other non-atomic cases. The operation *LookUp* has a non-fixed LP. To prove it linearizable, a backward simulation is necessary. Instead of doing a full backward simulation, we used the multiset as a second case study to *potential linearization points* [DSW11b], which simplify the reasoning effort.

While doing our proofs, we realized that the preconditions used for the local proof obligations as proposed before were slightly too weak. We fixed this by adding the disjointness property as part of the precondition. As a result we got a more generalized variant of local proof obligations. All proofs were mechanized in KIV and are available online [KIV12].

Elmas et al. were the first to verify the chosen case study linearizable [EQS$^+$10]. In their approach of using atomic actions [EQT09] a program gets transformed (abstracted and reduced) in several iterations until it is equal to the abstract specification by increasing the level of granularity of atomic actions. The approach needs no explicit specification of linearization points, since the implemented operations are reduced to atomic operations, for which a sequential reasoning is sufficient. The key to the approach is to identify code blocks that can be interleaved with other threads without conflicts, also referred to as commuting actions. Such code blocks are reduced to atomic blocks, if an SMT-solver is able to approve their commuting property. The approach establishes an abstraction mapping between the states of an abstract specification and its implementation, which requires that at least initial and final states are equal. Hence, the established mapping is weaker than a simulation. The identification of commuting actions is done manually and needs expertise. However, the specification effort is comparably low, because implementations do not need conversion into logic statements. In contrast to their approach, our focus lies in finding a valid abstraction function that can be used to verify our local proof obligations. Hence, an argument why an implementation is linearizable is part of the result in our approach. By using the interactive theorem prover KIV, we can automate proofs partially. Its visualization of proofs as trees can help to gain insights about why proofs fail and how they can be fixed (as was done with the local proof obligations during this case study).

Currently, we work on a linearizability proof of the case study with an approach combining temporal logic and rely guarantee reasoning [TSR11] to compare several verification approaches.

Acknowledgements: We would like to thank Bogdan Tofan for all his support. He helped us with the specifications in KIV and gave us important advice, when we carried out our proofs.

Bibliography

[ARR$^+$07] D. Amit, N. Rinetzky, T. W. Reps, M. Sagiv, E. Yahav. Comparison Under Abstraction for Verifying Linearizability. In *CAV*. Pp. 477–490. 2007.

[DSW07] J. Derrick, G. Schellhorn, H. Wehrheim. Proving Linearizability Via Non-atomic Refinement. In Davies and Gibbons (eds.), *IFM*. Lecture Notes in Computer Science 4591, pp. 195–214. Springer, 2007.

[DSW08] J. Derrick, G. Schellhorn, H. Wehrheim. Mechanizing a correctness proof for a lock-free concurrent stack. In *FMOODS 2008*. LNCS 5051, pp. 78–95. Springer, 2008.

[DSW11a] J. Derrick, G. Schellhorn, H. Wehrheim. Mechanically verified proof obligations for linearizability. *ACM Trans. Program. Lang. Syst.* 33(1):4, 2011.

[DSW11b] J. Derrick, G. Schellhorn, H. Wehrheim. Verifying Linearisabilty with Potential Linearisation Points. In *Proc. Formal Methods (FM)*. Pp. 323–337. Springer LNCS 6664, 2011.

[EQS$^+$10] T. Elmas, S. Qadeer, A. Sezgin, O. Subasi, S. Tasiran. Simplifying linearizability proofs with reduction and abstraction. In *In TACAS*. Pp. 296–311. Springer, 2010.

[EQT09] T. Elmas, S. Qadeer, S. Tasiran. A calculus of atomic actions. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. POPL '09, pp. 2–15. ACM, New York, NY, USA, 2009.

[HW90] M. Herlihy, J. M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM TOPLAS* 12(3):463–492, 1990.

[KIV12] Web presentation of KIV proofs for this paper. 2012. URL: http://www.informatik.uni-augsburg.de/swt/projects/MultiSet.html.

[OG76] S. S. Owicki, D. Gries. An Axiomatic Proof Technique for Parallel Programs I. *Acta Inf.* 6:319–340, 1976.

[RSSB98] W. Reif, G. Schellhorn, K. Stenzel, M. Balser. Structured Specifications and Interactive Proofs with KIV. In *Automated Deduction—A Basis for Applications*. Volume II, chapter 1: Interactive Theorem Proving, pp. 13 – 39. Kluwer, 1998.

[SDW12] G. Schellhorn, J. Derrick, H. Wehrheim. How to prove algorithms linearizable. In *Proc. CAV*. Springer LNCS, 2012.

[TSR11] B. Tofan, G. Schellhorn, W. Reif. Formal Verification of a Lock-Free Stack with Hazard Pointers. In *Proc. ICTAC*. Springer LNCS 6916, 2011.

[VP07] V. Vafeiadis, M. J. Parkinson. A Marriage of Rely/Guarantee and Separation Logic. In *CONCUR 2007*. LNCS 4703, pp. 256–271. 2007.