# A compositional proof method for linearizability applied to a wait-free multiset

**Bogdan Tofan, Gerhard Schellhorn, Wolfgang Reif**

**Angaben zur Veröffentlichung / Publication details:**

Tofan, Bogdan, Gerhard Schellhorn, and Wolfgang Reif. 2014. "A compositional proof method for linearizability applied to a wait-free multiset." In *Integrated Formal Methods: 11th International Conference, IFM 2014, Bertinoro, Italy, September 9-11, 2014, proceedings*, edited by Elvira Albert and Emil Sekerinski, 357–72. Cham: Springer. https://doi.org/10.1007/978-3-319-10181-1_22.



# A Compositional Proof Method for Linearizability Applied to a Wait-Free Multiset

Bogdan Tofan, Gerhard Schellhorn, and Wolfgang Reif

Institute for Software and Systems Engineering
Augsburg University
{tofan,schellhorn,reif}@informatik.uni-augsburg.de

**Abstract.** We introduce a compositional, complete proof method for linearizability that combines temporal logic, rely-guarantee reasoning and possibilities. The basic idea of our proof method is that each process must preserve possibility steps as an additional guarantee condition for linearizability. To illustrate the expressiveness of our method, we apply it to a wait-free multiset implementation with intricate linearization points. Both the soundness of our method as well as its application to our multiset have been mechanized in the interactive verifier KIV.

**Keywords:** Temporal Logic, Rely-Guarantee Reasoning, Linearizability, Wait-Freedom, Multiset, Interactive Verification.

## 1 Introduction

Data structure implementations that offer fast concurrent access on multi-core machines are of particular importance. These implementations use fine-grained locking or non-blocking techniques that apply atomic hardware instructions instead of locks, e.g., compare-and-set (CAS). Thus a higher degree of parallelism can be achieved.

The central safety property of these implementations is linearizability [6]. Roughly speaking, it requires that each concurrent data structure behavior corresponds to some behavior of an abstract data type with atomic operations. Furthermore, linearizability imposes the following constraint on the order of abstract behaviors: It must preserve the order of concrete executions that do not overlap in time. A strong progress condition for non-blocking data structures is wait-freedom: Wait-free operations terminate in a finite number of steps, independent of the behavior of other processes. Wait-free implementations are particularly useful in real-time settings where the number of execution steps of an operation must be known beforehand.

Our proof method for linearizability is based on the well-known (intuitive) technique of identifying linearization points. The key idea behind this approach is that a linearizable operation appears to take effect instantaneously during its execution [6]. This point in time is called a linearization point: In simple cases, linearization points are internal and static, i.e., they coincide with one specific instruction of a running operation, independent from the overall concurrent

system execution. We call linearization points that depend on the concurrent behavior of other processes potential linearization points [1]. In more complex cases, linearization points can be external, i.e., they can happen with an instruction of *another* process. Algorithms with potential external linearization points are particularly challenging for proving linearizability.

Possibilities [6] formalize the intuition of identifying linearization points. Our proof method is based on the key insight [13] that backward simulation with possibilities is a complete proof strategy for linearizability. To reason compositionally about linearizability, we combine a rely-guarantee decomposition rule with possibilities: The basic idea is that each process must preserve possibilities as an additional guarantee condition which performs a step-local backward simulation. We specify and verify our proof method for linearizability in the logic Rely-Guarantee Interval Temporal Logic (RGITL), which offers an expressive framework for the symbolic execution of sequential/interleaved programs with temporal logic [14]. The logic makes it possible to verify safety and liveness properties. It is implemented in the interactive verifier KIV. Both the soundness of our proof method as well as its application to verify the multiset linearizable and wait-free are mechanized in KIV [8].

To illustrate the expressiveness of our proof method, we consider a novel multiset implementation with wait-free operations to insert, lookup and delete an element, respectively. While our multiset operations are pretty simple, they pose intricate linearization problems similar to Herlihy and Wing's queue [6]. In particular, the multiset has potential external linearization points that change the abstract representation and linearize several other running processes.

The structure of the rest of this paper is as follows: Section 2 introduces our wait-free multiset implementation and shows the challenges of proving it linearizable. Section 3 briefly introduces RGITL, in particular rely-guarantee reasoning in the logic. Section 4 then defines our proof method and Section 5 illustrates its application to verify the multiset correct. Finally, Section 6 discusses related work and Section 7 concludes with a brief summary and possible future work.

## 2 A Simple Wait-Free and Linearizable Multiset

### 2.1 The Multiset Implementation

We introduce a multiset data structure that can be accessed concurrently by an arbitrary finite number of processes that repeatedly execute one of the algorithms `INSERT`, `DELETE` or `LOOKUP` given in Figure 1. All individual (atomic) steps of these operations are executed in an interleaved manner. First we explain these operations, then we describe our overall concurrent system model.

The implementation stores elements $x$ of the multiset in a shared array $Ar$ of size $N \neq 0$. Each array slot either contains an element or *empty*. All operations get an element $x$ as input parameter (before the semicolon). They sequentially run through the array $Ar$ and compute a boolean output value $Out$ (these two are reference parameters).

```
INSERT(x; Ar, Out) {
 let Found = f, Pos = 0 in
   while ¬ Found ∧ Pos < N do {
     CAS(empty, x; Ar[Pos], Found);
     if ¬ Found
     then Pos := Pos + 1
   }; Out := Found}

LOOKUP(x; Ar, Out) {
 let Found = f, Pos = 0 in
   while ¬ Found ∧ Pos < N do {
     if Ar[Pos] = x
     then Found := t
     else Pos := Pos + 1
   }; Out := Found}
```

```
DELETE(x; Ar, Out) {
 let Found = f, Pos = 0 in
   while ¬ Found ∧ Pos < N do {
     CAS(x, empty; Ar[Pos], Found);
     if ¬ Found
     then Pos := Pos + 1
   }; Out := Found}

CAS(Exp, New; Curr, Out) {
 if* Curr = Exp
 then Curr := New, Out := t
 else Out := f}
```

**Fig. 1.** The Wait-Free Multiset Operations INSERT, LOOKUP and DELETE in RGITL

When operation INSERT finds an empty slot, it atomically replaces *empty* with $x$ using a CAS instruction. CAS atomically compares a current location *Curr* with an expected value *Exp*. If the values are equal it sets *Curr* to a new value *New* and returns true; otherwise it returns false. We specify this using parallel assignments separated by comma, which need one atomic step to execute, and **if***, which (in contrast to using **if**) does not take an extra step to execute its test. Local variables are introduced with **let**. Operation DELETE atomically assigns *empty* to the first slot in $Ar$ that it finds to contain $x$. Operation LOOKUP returns true if it finds the searched element throughout its scan, otherwise it returns false.

In the following, **let** $\underline{V}$ **in** introduces arbitrary initial values for variables $\underline{V}$. For better readability we will write process identifiers $p\colon \mathsf{N}_0$ as subscripts rather than as an input parameter or function argument.

RGITL offers an operator $\|$ which interleaves[1] steps of its first and second component. Thus we can specify an overall concurrent system

$$\mathtt{SPAWN}_n(S)\ \{\ \textbf{if*}\ n = 0\ \textbf{then}\ \mathtt{PROC}_0(S)\ \textbf{else}\ \{\mathtt{PROC}_n(S)\ \|\ \mathtt{SPAWN}_{n-1}(S)\}\ \}$$

that recursively interleaves $n+1$ processes $\mathtt{PROC}_p(S)$ with identifiers $p \leq n$. The overall system state is $S\colon$ *state*. Each process repeatedly executes an operation $\mathtt{COP}_p(I, In; S, Out)$

$$\mathtt{PROC}_p(S)\ \{\ \{\textbf{let}\ I, In, Out\ \textbf{in}\ \mathtt{COP}_p(I, In; S, Out)\}\textbf{*}\ \}$$

with some operation index $I\colon$ *index*, input $In\colon$ *input* and output $Out\colon$ *output*. The star operator * denotes arbitrary iteration.

For the multiset, the operation index is one of $ins \mid del \mid lkp$, the input is an element, the state is $Ar$, the output is of type *bool*, and we instantiate $\mathtt{COP}_p$ as $\{$**if*** $I = ins$ **then** INSERT$(In; S, Out)$ **else if*** $I = del$ **then** DELETE$\ldots\}$.

[1] The version here does not assume (weak) fairness.

### 2.2 The Abstract and Concrete Specifications for Linearizability

To better understand the challenges of proving our multiset implementation linearizable, we first define its semantics in terms of an abstract specification. Then we briefly explain how to extend the abstract/concrete specifications with execution histories that represent the visible behaviors for linearizability.

Our abstract specification is based on *atomic* operation relations

$$AOP(I)(In, AS, AS', Out)$$

where $I$ is again the operation index, $AS/AS'$ is the abstract state before/after an atomic $AOP$-transition and $In/Out$ are input and output values, respectively. Initial abstract states are specified according to a predicate $AInit(AS)$.

For the multiset, the abstract state is an algebraic multiset $Ms$ and we define $AOP(lkp) = ALookUp$, $AOP(del) = ADelete$ and $AOP(ins) = AInsert$ as the following atomic relations: The lookup relation $ALookUp$ leaves the multiset unchanged and sets its output to true iff the input element $x$ occurs at least once in the current multiset ($x \in Ms$).

$$ALookUp(x, Ms, Ms', Out) \equiv Ms' = Ms \wedge (Out \leftrightarrow x \in Ms)$$

The $ADelete$ relation removes one occurrence of its input element $x$ from the current multiset $Ms$ if $x$ occurs in the multiset, otherwise it leaves the multiset unchanged and returns false (where $\{\!|.|\!\}$ denotes a multiset).

$$ADelete(x, Ms, Ms', Out) \equiv Ms' = Ms \setminus \{\!|x|\!\} \wedge (Out \leftrightarrow x \in Ms)$$

Finally, the insert relation $AInsert$ either adds its input $x$ to the current multiset (this increases the number of occurrences of $x$ by 1) and returns true, or it non-deterministically returns with output false and leaves the multiset unchanged. Restricting $AInsert$ to only return false if the multiset is full w.r.t. a predefined bound on the number of elements (typically the size $N$ of the array) would make the implementation non-linearizable.[2]

$$AInsert(x, Ms, Ms', Out) \equiv Ms' = Ms \cup \{\!|x|\!\} \wedge Out \vee Ms' = Ms \wedge \neg\, Out$$

Linearizability defines the behaviors of concrete and abstract operations in terms of execution *histories* which are finite sequences of events. An event $e\colon event$ models either the invocation $inv_p(I, In)$ or the return $ret_p(I, Out)$ of a particular operation $I$ that is invoked by a process $p$ with some input, possibly returning an output. We use the following simple selectors on events: $e.p/e.i$ selects the process identifier/the operation index and $inv_p.in/ret_p.out$ are the associated input/output values.

Linearizability extends the abstract operation relations with history parameters $Hs/Hs'$ and these extended operations additionally add a pair of an invoke

[2] For the same reasons, an atomicity check [4] for our multiset fails, since running the concrete code without interruption as an abstract specification does not offer the possibility to return false non-deterministically.

and return event to $Hs$ with every $AOP$-transition. Sequences of such operations that are executed by a finite number of processes from an initial state (where $AInit$ holds and $Hs$ is empty), generate the histories of the abstract specification.

Similarly, we extend the state of our concurrent system model $\texttt{SPAWN}_n$ with a history variable $H$ such that each process now first adds an invoke event $inv_p(I, In)$ to $H$ before it executes the internal steps $\texttt{COP}_p(I, In; S, Out)$ that leave $H$ unchanged.

$$\texttt{COP}_p(I, In; S, H, Out)\ \{\\ H := H + inv_p(I, In); \texttt{COP}_p(I, In; S, Out); H := H + ret_p(I, Out)\ \}$$

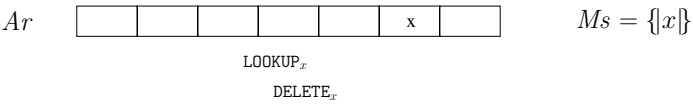
With its return, process $p$ adds a return event $ret_p(I, Out)$ to $H$. The overall system state is initialized using a predicate $Init(S, H)$ which requires $H$ to be empty. Hence, the visible behaviors $H$ of the extended system $\texttt{SPAWN}_n(S, H)$ consist of either i) $inv_p/ret_p$ events of a process $p$ that correspond to terminated executions of an operation or ii) pending invoke events where $p$ has added an invoke event to $H$ but not yet returned, i.e., it is still running. Due to preemption, invoke events in $H$ can be followed by events of *other* processes.

Roughly speaking, $\texttt{SPAWN}_n(S, H)$ is linearizable if every prefix of its histories $H$ corresponds to some history $Hs$ of the abstract specification that preserves the order of non-overlapping executions in $H$. (Two executions in $H$ are non-overlapping iff the invoke event of one operation occurs *after* the return event of the other one.) However, it is cumbersome to reason about linearizability by searching for a corresponding abstract behavior for each possible concrete behavior [6]. Reasoning in terms of linearization points (possibilities) is more convenient: The basic idea is that the unique order of linearization points in a concurrent execution precisely determines the order of atomic operations in a corresponding abstract execution.

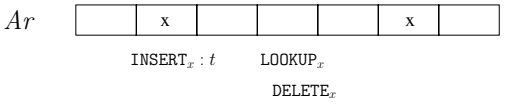
### 2.3 Challenges of Proving the Multiset Linearizable

This work started by looking at [3] where a *lock-based* multiset *without* a delete operation is shown to be linearizable. We and the authors of [3] first thought that adding a delete operation would violate linearizability. However, our presumed counter-example was flawed as we explain below. Our result here suggests that adding a (blocking) delete operation to their implementation should also be correct. Thus it solves an open challenge from [17].

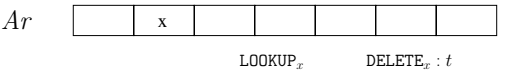
The presumed counter example was based on the following concrete execution. A lookup and a delete operation concurrently search for an element $x$ that lies ahead of their current positions but they have not reached $x$'s position yet:




Next, both operations are preempted and a concurrent insert operation successfully inserts $x$ below the current search indices of lookup and delete:




Then the delete operation runs to completion and removes $x$ from the upper part of the array:




Finally, the lookup operation completes and returns false.

This concurrent behavior seems to contradict linearizability: At least one occurrence of $x$ is always in the multiset while lookup returns false. It is however wrong to think that if some $x$ is always in the *array*, then it must also be in the *multiset*. Indeed, according to linearizability, the order of the abstract insert and delete operations may be changed here, since the respective concrete executions do *overlap* in time. That is, the concrete history

$$inv_p(lkp, x),\ inv_q(del, x),\ inv_r(ins, x),\ ret_r(ins, t),\ ret_q(del, t), ret_p(lkp, f)$$

can be correctly reordered to the abstract history

$$inv_q(del, x),\ ret_q(del, t),\ inv_p(lkp, x),\ ret_p(lkp, f),\ inv_r(ins, x), ret_r(ins, t)$$

where first the delete operation takes effect, deleting the initial occurrence of $x$ in the multiset, and thus making a lookup with false possible.

The concurrent execution above already motivates a central idea of our linearizability proof in terms of linearization points: Successful delete operations must potentially linearize *early* during their execution, before they actually delete their element from the array. Consequently, the abstract representation becomes a *collection of multisets*, since potentially linearizing a delete operation does not leave the abstract state unchanged and the linearization must be possibly revised due to future executions of other processes. To illustrate this effect, we consider the previous concurrent execution again:

Initially, no process is running and the abstract representation is merely $\{\{\!|x|\!\}\}$. In general, when no running delete operation exists, the abstract multiset is uniquely given by the elements in the array. As soon as a delete operation starts, it might have already linearized which gives possible multisets $\{\{\!|\}\!\}, \{\!|x|\!\}\}$ where the empty multiset $\{\!|\}\!\}$ results from deleting $x$ from the initial multiset $\{\!|x|\!\}$. After the insert operation succeeds, the abstract representation is either $\{\!|x, x|\!\}$, or $\{\!|x|\!\}$ if the delete has potentially linearized. Finally, as soon as the delete operation succeeds, $\{\!|x|\!\}$ becomes the only possible multiset again.

Thus we compute possible abstract multisets by executing running operations to the end, then abstracting the array content to a multiset. This corresponds to the general approach of [13] to compute observation trees.

Note that in the execution above, the lookup operation must linearize to false *with* the potential linearization of the delete operation that removes the *last* occurrence of $x$ from the multiset. If there were any delay between these two

linearizations, then a concurrent `INSERT` might insert $x$ below the current position of the lookup operation and linearizing to false would no longer be possible, since any possible abstract multiset would contain $x$. In this case, the linearization point of the delete operation is also an external potential linearization point for all running lookup/delete operations that can now complete with false.

The linearizability proof poses a further challenge for lookup/delete operations that return false: These operations must potentially linearize with false before they pass the first slot of the array. Starting with an empty array, after passing the first slot, a concurrent insert at the first slot makes linearizing these operations to false impossible. Together, successful delete operations, plus lookup/delete operations that return false, must potentially linearize *early* during their execution. Intuitively speaking, this allows us to move their linearization point towards the time of their invocation. We will formalize this intuition when we instantiate the abstraction relation of our proof method (see properties *DELt*/*DELf*/*LKPf* in Section 5.1).

## 3 RGITL

We specify and verify our proof method and the multiset case study in the logic RGITL that we briefly introduce next. For a detailed exposition refer to [14].

### 3.1 Syntax and Semantics

The semantics of RGITL is based on intervals which are finite or infinite sequences of the form $I = (I(0), I'(0), I(1), I'(1), I(2), \ldots)$ where every $I(k)$ and $I'(k)$ is a state function that maps variables to values. The state transition from $I(k)$ to $I'(k)$ is called a program transition, whereas the transition $I'(k)$ to $I(k+1)$ from a primed to the subsequent unprimed state is an environment transition. Thus intervals alternate between program and environment transitions, similar to reactive sequences [12].

The logic discerns static variables $v$ (written lowercase) that do not change in any transition of an interval, from dynamic variables $V$ (written uppercase) that can change arbitrarily. Primed and double primed variables $V'$ and $V''$ are evaluated over $I'(0)$ and $I(1)$, respectively, if $I$ is not empty. (For an empty interval, both $V'$ and $V''$ are evaluated over $I(0)$.) Formulas $\varphi$ are higher-order/temporal logic expressions of boolean type: For instance, the temporal logic operator $\varphi_1 \,\mathbf{until}\, \varphi_2$ states that $\varphi_2$ holds in some state of a given interval and up to that state $\varphi_1$ holds. From this operator, the standard temporal logic operators eventually $\Diamond$ and always $\Box$ can be easily derived. For instance, formulas $\Box\, V = V'$ and $\Box\, V' = V''$ state that variable $V$ does not change in any program/environment transition of an interval.

Assertions in RGITL are based on the well-known sequent calculus where a sequent $\Gamma \vdash \Delta$ is valid if the conjunction of all formulas from the antecedent $\Gamma$ implies the disjunction of all formulas from the succedent $\Delta$. Programs in RGITL are formulas: A program restricts the program transitions of an interval

only. A typical program assertion $Init, \alpha, E \vdash \varphi$ states that program $\alpha$ satisfies property $\varphi$, starting in an initial state that satisfies predicate logic formula *Init*, given that the environment behaves according to formula $E$.

### 3.2 RG Reasoning

Rely-Guarantee (RG) reasoning [7] extends Hoare's well-known approach to reason about sequential programs with pre-/post-conditions to a concurrent setting: Assumptions of a process $p$ about possible environment transitions are specified using a two-state predicate $R_p$: *state* × *state* → *bool* over the entire program state. These are called rely conditions. In return, each process $p$ must specify guarantees for its steps using a further two-state predicate $G_p$: *state* × *state* → *bool*, called guarantee conditions.

RGITL offers native support for RG assertions which are a special type of temporal formulas: An RG assertion for partial correctness

$$Pre(S), Inv(S) \vdash [R(S', S''), G(S, S'), Inv(S), \alpha(S)]\ Post(S)$$

requires that final states of a program $\alpha$ satisfy the post condition *Post*: *state* → *bool* if the program starts in a state where the precondition *Pre*: *state* → *bool* holds; program transitions preserve $G$ and propagate the invariant *Inv*: *state* → *bool* if previous environment transitions satisfy $R$ and propagate *Inv*, respectively. This semantics can be easily formalized in the logic using the **until** operator, see [17].

Similarly, an RG assertion for total correctness (using $\langle\ .\ \rangle$ instead of $[\ .\ ]$) strengthens partial correctness by additionally requiring that $\alpha$ terminates if the environment always preserves the rely conditions and propagates the invariant. (We verify such liveness properties by induction over a given variant term.)

RGITL offers a Hoare-style calculus for the symbolic execution of RG assertions for partial and total correctness of sequential programs. For instance, we execute an assignment $(S := e); \alpha$ according to the following rule

$$\frac{\begin{array}{l} Pre(s_0), Inv(s_0), s_1 = e \vdash G(s_0, s_1) \\ Pre(s_0), Inv(s_0), s_1 = e \vdash Inv(s_1) \\ Pre(s_0), s_1 = e, R(s_1, S), Inv(S) \vdash \langle R, G, Inv, \alpha\rangle\ Post(S) \end{array}}{Pre(S), Inv(S) \vdash \langle R(S', S''), G(S, S'), Inv(S), (S := e); \alpha\rangle\ Post(S)} \quad (1)$$

where the static variables $s_0/s_1$ denote the state vector $S$ before/after the assignment. In its first/second premise, the rule requires proving the guarantee/invariant propagation for the assignment transition. In the third premise, the RG assertion must be shown for the rest program $\alpha$: The antecedent is typically simplified to the stable part of $Pre(s_0)$ over the assignment and the subsequent rely, i.e., to a formula $Pre_{new}(S)$ with $Pre(s_0) \wedge s_1 = e \wedge R(s_1, S) \rightarrow Pre_{new}(S)$.

Symbolic execution is practical for sequential but not for interleaved programs. Therefore, we apply RG decomposition rules for interleaved programs that reduce the verification to the constituent (sequential) sub-programs. Here

we use the following RG decomposition rule for $\texttt{SPAWN}_n(S)$ (ignoring operation indices, inputs/outputs and histories for a moment):

$$\frac{\begin{array}{l}1)\ \mathit{reflexive}(G_p),\ \mathit{transitive}(R_p),\ G_p(S,S') \rightarrow R_q(S,S') \\ 2)\ \mathit{Pre}_p(S') \wedge R_p(S',S'') \rightarrow \mathit{Pre}_p(S''),\ \mathit{Post}_p(S') \wedge R_p(S',S'') \rightarrow \mathit{Post}_p(S'') \\ 3)\ \mathit{Inv}(S), \mathit{Pre}_p(S) \vdash [R_p, G_p, \mathit{Inv}(S), \texttt{COP}_p(S)]\ \mathit{Post}_p(S) \\ 4)\ (\exists\ S.\ \mathit{Init}(S)) \wedge (\mathit{Init}(S) \rightarrow \mathit{Inv}(S) \wedge \bigwedge_{p \leq n} \mathit{Pre}_p(S))\end{array}}{\mathit{Init}(S) \vdash [\ \bigwedge_{p \leq n} R_p,\ \bigvee_{p \leq n} G_p, \mathit{Inv}(S), \texttt{SPAWN}_n(S)]\ \bigwedge_{p \leq n} \mathit{Post}_p(S)} \tag{2}$$

The conclusion of the rule states that each transition of the interleaved system preserves some guarantee $G_p$ and propagates the invariant as long as the previous environment transitions satisfy all rely conditions $R_p$ and propagate the invariant. (Note that we can not prove total correctness for $\texttt{SPAWN}_n$, since the system can invoke infinitely many operations.) Premises 1), 2) and 4) are simple predicate logic conditions on the used RG conditions: Guarantees must be reflexive, relies transitive, and a guarantee step of a process $p$ must be a rely step for each other process $q$. Moreover, pre-/post-conditions must be stable over rely steps, since a process might start after/terminate before another process. Finally, there must exist an initial overall system state where predicate *Init*, the invariant and all pre-conditions hold. The central premise 3) requires proving an RG assertion for partial correctness of an individual operation $\texttt{COP}_p(S)$.

## 4 Proof Method: RG Reasoning with Possibilities

Our proof method for linearizability combines RG reasoning with possibilities as we explain next. The underlying system model is $\texttt{SPAWN}_n(S, H)$, Section 2.2.

### 4.1 Possibilities

Possibilities characterize linearizability in terms of linearization points (see Theorems 9 and 10 in [6]). Intuitively, our possibilities predicate $\mathit{Poss}(H, R, AS)$[3] holds if $H, R, AS$ has been reached by a finite sequence of invocation, linearization and return steps as defined below. Parameter set $R$ stores the return events for those running operations that have already linearized but not yet returned.

Formally, we define possibilities

$$\mathit{Poss}(H, R, AS) \equiv \exists\ AS_0.\ \mathit{AInit}(AS_0) \wedge \Delta\mathit{Poss}(([\,], \emptyset, AS_0), (H, R, AS))$$

as possibility steps $\Delta\mathit{Poss}$ on triples $(H, R, AS)$ that start with an empty history, an empty set $R$ and an initial abstract state $AS_0$. A possibility step is either an

[3] See [13], p. 248 for a comparison with the original syntax in [6].

invocation step *Invoke*, an abstract atomic operation step *Linearize*, or a return step *Return*

$$\Delta Poss \equiv (Invoke \vee Linearize \vee Return)^*$$

where $^*$ denotes the reflexive and transitive closure of the underlying relation.

In an invocation step, the executing process $p$ must not have a pending invocation event in $H$ ($nopi_p(H)$). The step adds an invoke event to $H$ but changes neither the return set $R$ nor the abstract state $AS$.

$$\begin{aligned} &Invoke((H, R, AS), (H', R', AS')) \equiv \exists\, p, I, In. \\ &\quad H' = H + inv_p(I, In) \wedge nopi_p(H) \wedge R = R' \wedge AS = AS' \end{aligned}$$

The execution of a linearization step requires a pending invoke in $H$ (denoted $pi(n, H)$ where $n < \#H$) for a process that has not yet linearized (no corresponding event in $R$). It executes an abstract atomic transition $AOP$ and adds the corresponding return event to $R$.

$$\begin{aligned} &Linearize((H, R, AS), (H', R', AS')) \equiv H = H' \wedge \exists\, n, Out. \\ &\quad pi(n, H) \wedge (\forall\, e.\ e \in R \rightarrow e.p \neq H(n).p) \\ &\wedge\ AOP(H(n).i)(H(n).in, AS, AS', Out) \wedge R' = R + ret_{H(n).p}(H(n).i, Out) \end{aligned}$$

We write $Lin_{I,Out}$ for a linearization step of operation $I$ with output $Out$.

Finally, a return step completes a running operation that has already linearized by removing its return event $e$ from $R$ and adding it to the history.

$$\begin{aligned} &Return((H, R, AS), (H', R', AS')) \equiv \\ &\quad AS = AS' \wedge \exists\, e.\ e \in R \wedge H' = H + e \wedge R' = R \setminus \{e\} \end{aligned}$$

To illustrate possibilities, we reconsider the concurrent multiset execution from Section 2.3 where the abstract multiset is $\{\!|x|\!\}$ initially and a lookup and a delete operation are invoked by processes $p/q$: Executing the *Invoke* steps for processes $p$ and $q$ we get a history $H = inv_p(lkp, x), inv_q(del, x)$. Now we have three possible continuations which yield possible values $(R, Ms)$ as follows: Either i) no *Linearize* transition is executed $(\emptyset, \{\!|x|\!\})$, or ii) the delete operation linearizes with true $(\{ret_q(del, t)\}, \{\!|\,|\!\})$, or iii) the delete operation linearizes with true and then the lookup linearizes with false $(\{ret_q(del, t), ret_p(lkp, f)\}, \{\!|\,|\!\})$.

### 4.2 Proof Method

Our proof method is a linearizability-specific instance of rule (2). Similar to premise 3) of the rule, our method essentially requires to show the following RG assertion for partial correctness of an individual process $p$

$$\begin{aligned} &nopi_p(H), Inv(S, H), \Box\ Out' = Out'' \\ &\vdash [R_p(S', H', S'', H'') \wedge R_p^{poss}(H', H''), G_p(S, H, S', H') \wedge G^{poss}(S, H, S', H'), \\ &\quad Inv(S, H), \mathtt{COP}_p(I, In; S, H, Out)]\ \mathbf{t} \end{aligned} \tag{3}$$

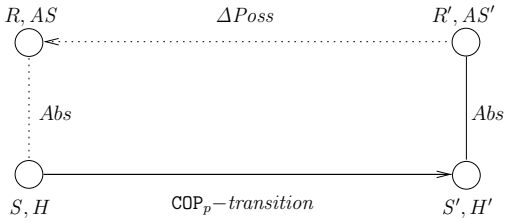



**Fig. 2.** Step-Local Backward Simulation

In (3), the rely/guarantee predicates $R_p/G_p$ can be chosen freely for each case study whereas predicates $R_p^{poss}/G^{poss}$ have linearizability-specific definitions that we introduce next. Predicate

$$G^{poss}(S,H,S',H') \equiv \forall\ R', AS'.\ Abs(S',H',R',AS') \rightarrow \\ \exists\ R, AS.\ Abs(S,H,R,AS) \wedge \Delta Poss(H,R,AS,H',R',AS')$$

ensures linearizability by propagating possibility steps backwards over each program transition as Figure 2 shows: For each transition of $\texttt{COP}_p$ from $S, H$ to $S', H'$, we must show that each abstract state $R', AS'$ that is related to $S', H'$ according to an abstraction relation $Abs$, has been reached by a finite number of possibility steps starting from some abstract state $R, AS$ that $Abs$ relates to $S, H$.

The main idea of the abstraction relation is to restrict the number of possible abstract states that must be propagated backwards in concrete proofs, by taking the concrete state into account. The abstraction relation must be *total* over invariant states $Inv(S,H) \rightarrow \exists\ R, AS.\ Abs(S,H,R,AS)$.

Furthermore, proof obligation (3) uses the following rely properties

$$R_p^{poss}(H',H'') \equiv \quad (nopi_p(H') \rightarrow nopi_p(H'')) \\ \wedge \forall\ n.\ pi(n,H') \wedge H'(n).p = p \rightarrow pi(n,H'') \wedge H'(n) = H''(n)$$

which ensure that after adding an invoke event $inv_p$ to $H$, this event remains pending and unchanged in $H$ throughout the entire execution of $p$. These properties obviously hold for an individual process of the concurrent system. They are required to propagate possibility steps during $p$'s execution, e.g., the linearization step *Linearize* requires a pending invocation in $H$ for the respective process.

In (3), the output variable *Out* is local, so the output that is computed by the internal steps in $\texttt{COP}_p$ corresponds to the output that is added to $H$ in the final return transition. The post-condition is trivial for simplicity, but using an extra predicate to allow more complex post-conditions is possible.

Finally, there must exist an initial concrete system state. All concrete initial states must correspond to abstract initial states where no process has linearized.

$$Init_H(S) \wedge Abs(S,H,R,AS) \rightarrow AInit(AS) \wedge R = \emptyset$$

**Theorem 1 (Compositional Proof Method for Linearizability).** *With the predicate logic side conditions above, proof obligation (3) is a compositional proof method for linearizability:*

$$Init_H(S), \texttt{SPAWN}_n(S,H), \Box\ (S' = S'' \wedge H' = H'') \\ \vdash \Box\ ((\exists\ r, as.\ Poss(H, r, as)) \wedge (\exists\ r, as.\ Poss(H', r, as)))$$

Intuitively, the theorem states that each prefix of $H$ has a possibility (which implies linearizability). Further details and a mechanized soundness proof are described at [8]. Our method is compositional as it ensures the overall system property of linearizability based on the process-local RG proof obligation (3). Completeness of the method follows from the completeness of the step-local backward simulation technique in [13] (based on Owicki/Gries reasoning [11]) and the completeness of RG reasoning w.r.t. the Owicki/Gries method [12].

Since we typically want to talk about local states in concrete RG specifications, procedure $\texttt{COP}_p$ in (3) can initialize local variables $S.LSf_p$ of process $p$ with the invocation transition using an initialization function $init(I)$. (Directly using a **let** for the initialization would hide relevant local state information from specifications. We leave locality properties for $LSf_p$ implicit in (3).)

## 5 Verifying the Multiset

To talk about local states in the multiset specifications, we introduce a function $LSf$ as part of the program state $S = LSf, Ar$ which stores the following local information for each process $p$: $I_p$ is the operation index, $In_p$ is the input element, $Found_p$ is the boolean flag that determines whether the operation has found the searched element and $Pos_p$ is the current index position of the running operation.

### 5.1 Instantiating the Abstraction Relation

The abstraction relation $Abs$ of our proof method formalizes the intuitive considerations from Section 2.3 by relating a concrete state $S, H$ to possible abstract states $R, Ms$ as $Abs(S, H, R, Ms) \equiv BASE \vee DELt \vee DELf \vee LKPf$. In the base case $BASE \equiv Ms = Absf(Ar) \wedge R = Linsf(LSf, Ar)$ the abstract multiset $Ms$ consists of all elements in $Ar$, computed by function $Absf$. Set $R$ corresponds to precisely those running processes which have either not found their searched element and are at the end of their scan or which have found it and set their found-flag to true. Function $Linsf$ computes the return events of these processes.

The second disjunct in the definition of $Abs$ describes the early linearization of a running delete operation (of process $p$) that *potentially* deletes the searched element that lies ahead of its current position at $Ar[n]$.

$$DELt(S, H, R, Ms) \equiv \exists\ p, n. \\ I_p = del \wedge \neg\ Found_p \wedge Pos_p \leq n < \#Ar \wedge Ar[n] = In_p \\ \wedge (\forall\ n_0.\ Pos_p \leq n_0 < n \rightarrow Ar[n_0] \neq In_p) \wedge ret_p(del, t) \in R \\ \wedge Abs(LSf, Ar[n]{:=}empty, H + ret_p(del, t), R \setminus \{ret_p(del, t)\}, Ms)$$

The definition can be viewed to consist of three steps. First a possible future state $(S', H')$ from current state $(S, H)$ is computed by running the remaining steps of $p$. This deletes input element $In_p = Ar[n]$ (the resulting array $Ar'$ is written $Ar[n]{:=}empty$) and adds the return event $ret_p(del, t)$ to $H$. Second, $Abs$ is called recursively to compute a possible abstract state $(R', Ms')$ for state $(S', H')$. If the recursive call chooses the base case then $Ms'$ is just the content of $Ar'$. Finally the effect of linearizing the delete early is to add the return event $ret_p(del, t)$ to the set $R'$. The final result of $Abs$ therefore is $(R, Ms) = (R' \cup ret_p(del, t), Ms')$.

The third disjunct of $Abs$ similarly considers a running delete process that potentially linearizes to false as it does not see its searched element ahead

$$DELf \equiv \exists\, p.\\ I_p = del \wedge \neg\, Found_p \wedge Pos_p < \#Ar \wedge (\forall\, n.\ Pos_p \leq n < \#Ar \rightarrow Ar[n] \neq In_p)\\ \wedge\, ret_p(del, f) \in R \wedge Abs(LSf, Ar, H + ret_p(del, f), R \setminus \{ret_p(del, f)\}, Ms)$$

The last disjunct $LKPf$ for a lookup operation that returns false is symmetric to $DELf$. It is easy to see that the recursion in $Abs$ is well-founded, since it decreases the number of running processes.

### 5.2 The Main Proofs

Instantiating the RG parameters of our proof method is straight-forward: In the overall initial system state the array is empty. The invariant states that for each running process, the pending invocations in $H$ correspond to the respective local state information in $LSf$. The rely condition $R_p$ states that the length of the array is not concurrently changed and the guarantee $G_p$ is defined as the rely conditions of all other processes.

With these instances, the predicate logic premises of our proof method hold trivially. Therefore, we only focus on the central proof obligation (3) which requires to prove an RG assertion for partial correctness for each individual multiset operation run by a process $p$. To also prove wait-freedom of each multiset operation, we show its stronger version for total correctness by induction over the variant $\#Ar - Pos_p$. Symbolic execution of the algorithms leads to proof goals for each transition (first premise of rule (1) for an assignment) where the guarantee must be shown. In particular, $G^{poss}$ must hold for the transition, so a sequence of suitable abstract steps $\Delta Poss$ has to be chosen which makes the diagram of Fig. 2 commute. The choice is easy — usually the empty sequence since the step does not linearize any running algorihm — for all steps except for one step in each algorithm as detailed below. All proofs then are by well-founded induction over the number of running processes. They unfold the definition of $Abs$ for both states $(S, H)$ and $(S', H')$. The base case is usually trivial, each of the three recursive cases gives two states $(S_0, H_0)$ and $(S_0', H_0')$ shown in Fig. 3 that are reached by executing one pending operation to the end (indicated by the dashed line). Often the $\texttt{COP}_p$-transition commutes, i.e., it also modifies $(S_0, H_0)$ to $(S_0', H_0')$. Then the induction hypothesis (the dotted lines in the figure) closes the premise immediately. Otherwise, the state $(S_0'', H_0'')$ reached by executing the transition from $(S_0, H_0)$ must be

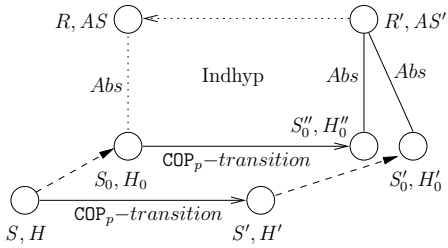



**Fig. 3.** Inductive proof scheme for backward simulation

shown to represent the same possible abstract states as $(S'_0, H'_0)$, which is usually proved as a lemma using the same proof principle again.

The complex case for the $\texttt{DELETE}_x$ operation is the invocation transition. The proof discerns three cases. First, the delete may not have linearized, i.e., the set $R'$ in Fig. 2 does not contain a return event for the process $p$ executing the operation. Then $\Delta Poss$ is empty. Otherwise, when the multiset after the invoke ($AS'$ in Fig. 2) still contains $x$, then just the delete linearizes, i.e., $\Delta Poss = Lin_{del,t}$. Otherwise, the linearization of the delete triggers some linearizations of lookups and deletes to false, i.e. $\Delta Poss = Lin_{del,t}; (Lin_{lkp,f} \vee Lin_{del,f})^*$. The exact sequence is determined by the difference between $R'$ and $R$.

The critical transition for $\texttt{LOOKUP}_x$ is finding $x$. In particular, if we consider a multiset (after this transition) that does not contain $x$, there must be a running delete process $q$ that (potentially) removes the last occurrence of $x$ right *after* $p$ linearizes with true. As a consequence, the transition then linearizes the current lookup process with true, the running delete process $q$ with true, plus again a sequence $(Lin_{lkp,f} \vee Lin_{del,f})^*$ of currently running lookup/delete processes that can now return with false.

The critical transition of $\texttt{INSERT}_x$ is when it puts $x$ in an empty array slot. This step can additionally linearize a running delete operation that now potentially deletes the element that has just been inserted. In this case the effects of the abstract insert and delete operations cancel each other (such behavior is typically found for data structures that use elimination [10]).

## 6 Related Work

The basic idea of our guarantee condition $G^{poss}$ is based on [13] where backward simulation is shown to be sound and complete for linearizability. The approach uses predicate logic and non-compositional Owicki/Gries reasoning [11]. The adaptation to our temporal logic setting with RG reasoning has the following benefits: Verified programs can be specified in an abstract programming language rather than as transition systems with program counters. More importantly, it avoids the manual encoding of *local* state information that merely reflects the control flow of a program. Such properties are automatically computed and propagated by the symbolic execution of RG assertions (see also [17] for a comparison of our two local proof methods for linearizability for a

restricted class of linearizable algorithms where potential linearization points do not modify the abstract state). Finally, we can verify liveness properties within one framework, here wait-freedom (total correctness) of the multiset operations. (For more challenging liveness proofs in RGITL see [15,14].)

Doherty et al. [2] use forward/backward simulations in a non-compositional approach to verify linearizability based on IO-automata. They also report on model checking linearizability. In general, model checking linearizability can quickly find bugs, however, it does not consider all possible executions [19].

Recent work [9] describes an RG-based approach for proving linearizability which annotates potential linearizations in the concrete specifications using abstract auxiliary code that works on a state that roughly corresponds to our $R, AS$. In contrast, we separate concrete and abstract code using a step-local simulation. The approach is manual and only considers partial correctness, while we mechanically verify the soundness of our method as well as its application.

The proof obligations in [18] are restricted to "pure" linearization points that leave the abstract state unchanged (as in [1]) and thus cannot prove our multiset. In more recent work [5], a complete approach for proving linearizability for a specific type of purely blocking queue algorithms is introduced. (An operation is purely blocking if its infinite blocked executions never modify the shared state.) Proof obligations are not based on linearization points, but rather on a characterization of queue-specific behaviors. They mechanize a proof for Herlihy Wing's queue, but only give a manual soundness proof of their reduction. Nevertheless, as our wait-free multiset is purely blocking, it would be interesting to find such characterizations for multisets and to analyse how their proofs would relate to ours.

## 7 Conclusion

We have introduced a general proof method for linearizability based on possibilities. It improves the complete proof strategy of [13] by using RG reasoning and symbolic execution with temporal logic. We have illustrated the expressiveness of our method by verifying a novel wait-free multiset implementation with potential external linearization points that change the representation and linearize several other processes. We leave it for future work to investigate whether the multiset can be verified with only a fixed small number of local states instead of the full function $LSf$, by exploiting the symmetry of the underlying operations (similar to [16]). Another option for future work is to apply our techniques to further algorithms such as the elimination queue [10] that can be verified based on similar ideas.

**Acknowledgement.** We thank Stefan Schödel for verifying various lemmas of the case study in KIV.

## References

1. Derrick, J., Schellhorn, G., Wehrheim, H.: Verifying linearisability with potential linearisation points. In: Butler, M., Schulte, W. (eds.) FM 2011. LNCS, vol. 6664, pp. 323–337. Springer, Heidelberg (2011)

2. Doherty, S., Groves, L., Luchangco, V., Moir, M.: Formal verification of a practical lock-free queue algorithm. In: de Frutos-Escrig, D., Núñez, M. (eds.) FORTE 2004. LNCS, vol. 3235, pp. 97–114. Springer, Heidelberg (2004)
3. Elmas, T., Qadeer, S., Sezgin, A., Subasi, O., Tasiran, S.: Simplifying linearizability proofs with reduction and abstraction. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 296–311. Springer, Heidelberg (2010)
4. Flanagan, C., Freund, S.N.: Atomizer: A dynamic atomicity checker for multithreaded programs. In: Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, pp. 256–267. ACM, New York (2004)
5. Henzinger, T., Sezgin, A., Vafeiadis, V.: Aspect-oriented linearizability proofs. In: D'Argenio, P.R., Melgratti, H. (eds.) CONCUR 2013. LNCS, vol. 8052, pp. 242–256. Springer, Heidelberg (2013)
6. Herlihy, M., Wing, J.: Linearizability: A correctness condition for concurrent objects. ACM Trans. on Prog. Languages and Systems 12(3), 463–492 (1990)
7. Jones, C.B.: Specification and design of (parallel) programs. In: Proceedings of IFIP 1983, pp. 321–332. North-Holland (1983)
8. KIV: Presentation of KIV proofs for wait-free multiset (2014) (2013), `https://swt.informatik.uni-augsburg.de/swt/projects/ifm14.html`
9. Liang, H., Feng, X.: Modular verification of linearizability with non-fixed linearization points. In: Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2013, pp. 459–470. ACM (2013)
10. Moir, M., Nussbaum, D., Shalev, O., Shavit, N.: Using elimination to implement scalable and lock-free fifo queues. In: SPAA, pp. 253–262. ACM (2005)
11. Owicki, S.S., Gries, D.: An Axiomatic Proof Technique for Parallel Programs I. Acta Inf. 6, 319–340 (1976)
12. de Roever, W.P., de Boer, F., Hannemann, U., Hooman, J., Lakhnech, Y., Poel, M., Zwiers, J.: Concurrency Verification: Introduction to Compositional and Noncompositional Methods. Cambridge Tracts in Theoretical Computer Science, vol. 54. Cambridge University Press (2001)
13. Schellhorn, G., Derrick, J., Wehrheim, H.: How to prove algorithms linearisable. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 243–259. Springer, Heidelberg (2012)
14. Schellhorn, G., Tofan, B., Ernst, G., Pfähler, J., Reif, W.: RGITL: A temporal logic framework for compositional reasoning about interleaved programs. Annals of Mathematics and Artificial Intelligence (AMAI) (2014)
15. Tofan, B., Bäumler, S., Schellhorn, G., Reif, W.: Temporal logic verification of lock-freedom. In: Bolduc, C., Desharnais, J., Ktari, B. (eds.) MPC 2010. LNCS, vol. 6120, pp. 377–396. Springer, Heidelberg (2010)
16. Tofan, B., Schellhorn, G., Reif, W.: Formal verification of a lock-free stack with hazard pointers. In: Cerone, A., Pihlajasaari, P. (eds.) ICTAC 2011. LNCS, vol. 6916, pp. 239–255. Springer, Heidelberg (2011)
17. Tofan, B., Travkin, O., Schellhorn, G., Wehrheim, H.: Two approaches for proving linearizability of multiset. Science of Computer Programming Journal (to appear, 2014)
18. Vafeiadis, V.: Automatically proving linearizability. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 450–464. Springer, Heidelberg (2010)
19. Vechev, M., Yahav, E., Yorsh, G.: Experience with model checking linearizability. In: Păsăreanu, C.S. (ed.) Model Checking Software. LNCS, vol. 5578, pp. 261–278. Springer, Heidelberg (2009)