Simplifying Linearizability Proofs with Reduction and Abstraction

Tayfun Elmas[1], Shaz Qadeer[2], Ali Sezgin[1], Omer Subasi[1], and Serdar Tasiran[1]

[1] Koç University, İstanbul, Turkey
{telmas,asezgin,osubasi,stasiran}@ku.edu.tr
[2] Microsoft Research, Redmond, WA
qadeer@microsoft.com

Abstract. The typical proof of linearizability establishes an abstraction map from the concurrent program to a sequential specification, and identifies the commit points of operations. If the concurrent program uses fine-grained concurrency and complex synchronization, constructing such a proof is difficult. We propose a sound proof system that significantly simplifies the reasoning about linearizability. Linearizability is proved by transforming an implementation into its specification within this proof system. The proof system combines reduction and abstraction, which increase the granularity of atomic actions, with variable introduction and hiding, which syntactically relate the representation of the implementation to that of the specification. We construct the abstraction map incrementally, and eliminate the need to reason about the location of commit points in the implementation. We have implemented our method in the QED verifier and demonstrated its effectiveness and practicality on several highly-concurrent examples from the literature.

1 Introduction

Linearizability is a well-known correctness criterion for concurrent data-structure implementations [1]. A concurrent implementation, denoted *Impl*, is said to be linearizable with respect to a sequential specification, denoted *Spec*, if every concurrent operation *op* of *Impl* takes effect atomically between its call and return points, where the correct effect is described by a sequential operation *op'* in *Spec*.

The typical proof of linearizability establishes an *abstraction map*, from *Impl*-states to *Spec*-states [2], and shows that only one action of *op*, called the *commit action*, is mapped to *op'*, and other actions are mapped to stuttering (identity) transitions in *Spec*. Under fine-grained concurrency control, constructing such a proof requires considerable expertise. First, identifying the commit action becomes nontrivial when *op* is written in terms of many small actions that make visible changes to the state. It further complicates the analysis when the commit point is determined at runtime depending on thread interleavings. Second, while the abstraction map relates *Impl*-states to *Spec*-states, it must also filter out the effects of the partially completed operations of *Impl* on the state except for the

commit action. This includes completing partial operations or rolling back the effects of these operations back to a clean state [3].

In this paper, we present a new method for proving linearizability of programs with fine-grained concurrency. Our method permits more tractable proofs by eliminating the above difficulties of constructing an abstraction map and not requiring the identification of the commit points. In [4], we showed that by interleaving reduction with abstraction, we can increase atomicity to the point that assertions in a concurrent program can be verified by sequential (local) reasoning. In this work, we argue that program rewriting guided by atomicity is an effective method for proving linearizability and present a sound proof system and a supporting tool that realize this method in a formal and practical setting.

We prove that *Impl* is linearizable with respect to *Spec*, by transforming *Impl* to *Spec* via a sequence of phases. In a *reduction* phase, we alternate reduction and abstraction to mark a set of sequentially composed actions as atomic. These actions are collected together, and the effects of thread interleavings are eliminated. In a *refinement* phase, we couple variable introduction and variable hiding, in order to make the code closer to the specification. These techniques provide us with the ability to *syntactically* relate implementation of a data structure to a specification with a different representation. We also provide the soundness guarantee that, the proof transformations preserve the behaviors of the original program. Thus, one can simplify the program by growing atomic blocks and continue the linearizability proof with another method, e.g., separation logic [5].

Interleaving reduction and refinement phases supports the incremental construction of the abstraction map. By increasing atomicity, a reduction phase enables a following refinement phase to implicitly establish a simple and clean abstraction map towards the specification. A refinement phase also helps to improve a following reduction phase by eliminating *superficial conflicts*: Two equivalent operations might conflict on low-level (implementation) variables but this does not necessarily correspond to real conflicts in terms of the final specification. Our solution to this issue indirectly introduces a semantic hierarchy into mover checks in reduction, which is not particular to linearizability and is likely to be useful in any kind of reduction proof.

We have implemented our method in the QED verifier. We demonstrate the effectiveness and practicality of our method by proving linearizability of several tricky examples from the literature. All proofs are available online and reproducible using QED.

1.1   Related Work

Refinement between a concurrent program and its sequential specification is well-studied [2,6,7,8]. Previous work showed that, under certain conditions, auxiliary variables enable construction of an abstraction map to prove refinement [2,6]. However, in practice writing an abstraction map for programs with fine-grained concurrency remains a challenge since there are a large number of cases to consider. [3] used a complex abstraction map, called *aggregation function*, that completes the atomic transactions that are committed but not yet finished. The

refinement proofs in [1,9,10,11], despite being supported by automated proof checkers, all require manual guidance for the derivation of the proof, requiring the user to manage low-level logical reasoning. On the other hand, in our method the user guides the proof via code transformations at the programming language level. Recently, [5] provided a tool that automates the derivation of the proof using shape abstraction. To our knowledge, its automating ability is limited to linked-list based data structures and it still requires identification of the possible commit points.

Owicki-Gries [12,1] and rely-guarantee [13] methods have been used in refinement proofs. However, in the case of fine-grained concurrency, deriving the proof obligations in both approaches requires expertise. The idea of local reasoning is exploited by separation logic [14] which is not particularly useful for shared objects with high level of interference. In these cases, we show that abstraction is an important tool to reduce the effects of interference.

Wang and Stoller [15] statically prove linearizability of the program using its sequentially executed version as the specification. Their notion of atomicity is defined over a fixed set of primitives, which is limited in the case of superficial conflicts. On the other hand, our notion of atomicity is more general and supported by abstraction to prove atomicity even under high level of interference. They provided hand-crafted proofs for several non-blocking algorithms, and our proofs are mechanically checked. In [16], Groves gives a hand-proof of the linearizability of the nonblocking queue, by reducing executions the fine-grained program to its sequential version. His use of reduction is non-incremental, and must consider the commutativity of each action by doing a global reasoning, while our reasoning is local.

2 Motivation and Overview

Our running example is a multiset of integers. Figure 1 shows the concurrent implementation (*Impl*), and the sequential specification (*Spec*), of `InsertPair` and `LookUp` operations[1]. The instruction `assume` ϕ blocks until ϕ becomes `true`, and `havoc x` assigns a nondeterminstic value to `x`. Our goal is to verify linearizability of *Impl* with respect to *Spec*.

Spec uses the variable `S`, which maps each integer to its cardinality in the multiset. Initially, `S` is empty, so `S[x]==0` for every integer `x`.

Impl contains an array `M` of `N` slots. For each slot, the `elt` field stores an integer, and the `stt` field indicates the status of the slot. The atomic `FindSlot` operation[2] allocates an empty slot by setting its `stt` field to `reserved`, and returns its index. `FindSlot` fails and returns `-1` if it cannot find any empty slot. The lock of each slot is acquired and released separately by `lock` and `unlock` operations, respectively.

---

[1] We omit the `Insert` operation to simplify the explanation.

[2] The original implementation of `FindSlot` uses fine-grain locking, and traverses the array using a loop similar to that of `LookUp`. In order to simplify the explanation, we use a version of `FindSlot` that has already been transformed using our proof steps.

Implementation (_Impl_)

```
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;
1 for (i := 0; i < N; i++) {
2  lock(M[i]);
3  if (M[i].elt==x && M[i].stt==full){
4    unlock(M[i]);
5    r := true; return;
6  } else unlock(M[i]);
7 }
8 r := false;

atomic FindSlot(x:int) returns (r:int)
1 if (forall 0<=i<N. M[i].stt != empty) {
2  r := -1;
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;
1 i := FindSlot(x);
2 if (i == -1) {
3  r := false; return;
4 }
5 j := FindSlot(y);
6 if (j == -1) {
7  M[i].stt := empty;
8  r := false; return;
9 }

10 M[i].elt := x;
11 M[j].elt := y;

12 lock(M[i]);
13 lock(M[j]);
14 M[i].stt := full;
15 M[j].stt := full;
16 unlock(M[i]);
17 unlock(M[j]);
18 r := true;
```

Specification (_Spec_)

```
var S: array [int] of int;
atomic LookUp(x:int) returns (r:bool)
  r := (S[x] > 0);
```

```
atomic InsertPair(x:int, y:int) returns (r:bool)
  if(r) { S[x] := S[x] + 1; S[y] := S[y] + 1; }
```

Fig. 1. The concurrent implementation and the sequential specification of multiset

A typical linearizability proof establishes an abstraction map that relates the slots M of _Impl_ to the map S of _Spec_. Let |A| denote the cardinality of the set A. The following abstraction map expresses the programmer's design intent clearly:

```
S[x]==|{ i | 0<=i<N && M[i].elt==x && M[i].stt==full }|
```

In words, for each integer x, the number of slots i in _Impl_ with M[i].elt==x and M[i].stt==full represents S[x] in _Spec_. When the proof is done at finest granularity of concurrency, more complicated variants of this abstraction map has to be used. In the following, we envision such a proof, and highlight common difficulties. We then illustrate how our proposed approach alleviates these difficulties and allows the proof to use the above map.

2.1   Challenges in a Typical Refinement Proof for Multiset

Abstraction maps and commit points.   Many techniques work by first selecting a commit point in every operation. The most likely choice for the commit point for InsertPair is line 16, since releasing the first lock makes the inserted element M[i].elt visible to other threads. Consider an abstraction map from _Impl_ to _Spec_ and suppose that line 16 of InsertPair is executed by _Impl_. This transition must be mapped to a single transition that increments S[x] and S[y] atomically. As a first try, let us consider the simple abstraction map introduced above:

```
S[x]==|{ i | 0<=i<N && M[i].elt==x && M[i].stt==full }|
```

This map does not work with this choice of commit point, because when lines 14 and 15 of `InsertPair` are executed, `S[x]` and `S[y]` are incremented, but the execution has not reached the commit point yet. In addition, the updates that are propagated to `S` are not atomic. Our next, slightly more sophisticated map below does not update `S[x]` and `S[y]` while the locks to these cells are held. (`HeldBy(M[i],t)` is true when thread `t` is holding the lock of `M[i]`):

```
S[x]==|{ i | 0<=i<N && M[i].elt==x && M[i].stt==full && !HeldBy(M[i],t)}|
```

The problem with this map is that every slot locked by a thread would be excluded from `S`. As a result, at line 16 (the commit point) the map would increment `S[x]` but not `S[y]` since `M[j]` is still locked. Thus, this map still does not accomplish the atomic specification state update we are after. The right map has to complete this partial update at the commit point by incrementing `S[y]` as well although the lock of `M[j]` is still held.

We next try different selections of commit points: lines 14, 15 or 17. For each of these choices, in order to produce the intended specification state and avoid non-atomic updates to it, an abstraction map must "roll back" effects of executions of `InsertPair` that have not reached their commit point, and must "complete" the effects of others that are past their commit point but have not yet finished. To accomplish this, the map must refer to not only the locking state but also the program counters of all threads.

Non-fixed commit points. Another issue that complicates the linearizability proof for multiset is that the commit action of `LookUp` is not fixed, but depends on the concurrently executing insertions by other threads. If `LookUp(x)` returns true its commit action is at line 3, where it finds out that the slot being visited contains `x` and is valid. When `LookUp(x)` fails, its commit point must be chosen as the first read of a slot it performs or earlier. This is because, in the absence of a `Delete` operation, it is possible that `x` gets inserted into a slot `M[i]`, after `LookUp` visits the ith slot and fails to find `x`, therefore, the commit point cannot be past the first read of a slot. Techniques that depend on the existence of a fixed commit point would be ineffective in such situations [13].

2.2   Proof by Reduction and Abstraction

Observe that the code blocks between lines 12-17 of `InsertPair` is *atomic*, i.e., any execution in which the actions of this block are interleaved with actions from other threads can be transformed into one in which actions of the commit block are contiguous. The technique we present allows us to express this fact and use it in a sound manner in a refinement or linearizability proof. Being able to treat the commit block as a single atomic action eliminates all of the potential difficulties outlined above.

In our method the proof is constructed by transforming *Impl* to *Spec*, both shown in Figure 1. This is done through a reduction phase followed by a refinement phase. In the reduction phase, we reduce the bodies of `InsertPair` and

`LookUp` to single atomic actions. This phase is guided by a simple hint about the locking discipline (see [17] for details of automating reduction).

In order to handle the non-fixed commit points of `LookUp`, we apply a transformation to separate its succeeding and failing executions. Since each failing iteration of `LookUp` is a left-mover, the failing branch of `LookUp` trivially reduces to a single action. For the successful branch, we apply an abstraction to the failing iterations that makes them also right-movers, and combine the abstracted iterations with the final, successful iteration. This reduces the successful branch into an atomic action. At the end, we obtain `LookUp` as a single atomic action that summarizes both successful and failing executions of the original code.

After transforming `InsertPair` and `LookUp` to single atomic actions, the locking state becomes unnecessary. We use variable hiding to clean up the calls to `lock` and `unlock`. Finally, we arrive at the representation of the multiset in *Spec* in three proof steps. First, we introduce the *Spec* variable `S` to the current version of the program. Then, we add (and prove) the following invariant, which links the new variable to the array `M`:

```
S[x]==|{ i | 0<=i<N && M[i].elt==x && M[i].stt==full }|
```

Recall that the above invariant establishes the simplest abstraction map that reflects the programmer's design intent. The invariant allows us to add the assignments `S[x] := S[x] + 1;` and `S[y] := S[y] + 1;` at the end of `InsertPair`. We follow the introduction of `S` with a variable hiding step in which we replace the bodies of `InsertPair` and `LookUp` with the corresponding bodies in *Spec* (Figure 1). Our soundness theorems given in Section 5 guarantee that transforming *Impl* to *Spec* using our rules implies the linearizability of *Impl*.

What is noteworthy about the proof we outlined is that it handles two separate concerns in separate proof steps: 1) concurrency control using locking and the `stt` field, and 2) relating the array-based representation of *Impl* to the representation in *Spec*. This example does not illustrate the use of variable hiding to eliminate superficial conflicts. In Section 6 we provide an example that does.

3   Concurrent Programs: Syntax and Semantics

Program. A program P is a tuple $P = \langle Global_P, Proc_P \rangle$. $Global_P$ is the set of uniquely-named global variables. $Proc_P$ is a set of procedures. A procedure is a tuple $\langle \rho, local_\rho, body_\rho \rangle$, where ρ is the *name*, $local_\rho$ is the set of *local variables*, and $body_\rho$ is the *body* of the procedure.

We distinguish the input variables $\overrightarrow{in}_\rho \subseteq local_\rho$ and the output variables $\overrightarrow{out}_\rho \subseteq local_\rho$. The tuple $\langle \rho, \overrightarrow{in}_\rho, \overrightarrow{out}_\rho \rangle$ is called the *signature* of the procedure. The signatures of the procedures in *Proc* form the signature of the program, denoted $\mathsf{Sig}(P)$. We employ the convention that the variables in $\overrightarrow{in}_\rho$ and $\overrightarrow{out}_\rho$ are read-only and write-only, respectively, while the rest of the variables in $local_\rho$ can be both read and updated.

We use Var_P to denote $Global_P \cup \bigcup_{\rho \in Proc} local_\rho$. We assume that each local variable is used in a unique procedure. Var'_P consisting of the primed version

of each variable in Var_P. We omit the subscripts when the program and the procedure are clear from the context.

Execution model. Let Tid be the set of all thread identifiers. For simplicity of presentation, we assume that procedure calls are inlined properly, assuming no recursion in the call chain. In general, our method applies to the inter-procedural case allowing recursion [4].

Without loss of generality, each thread calls one procedure ρ from $Proc$, and terminates when ρ returns. Statements of the procedures may refer to the current thread id through the special variable $tid \in Global$, whose domain is Tid.

Syntax. We assume that each atomic statement α, which we call an *(atomic) action*, is in the form: assert a; p. Let ρ be the procedure whose body contains α, and $V = Global \cup local_\rho$. The *assert predicate* a be over only unprimed variables from V. The *transition predicate* p is over both primed and unprimed variables in $V \cup V'$. For any action α, let ϕ_α and τ_α denote its assert and transition predicates. For instance, $\phi_\alpha = a$ and $\tau_\alpha = p$, for α given above.

We use sequential composition ($;$), choice ($\square$) and loop ($\circlearrowleft$) operators to form *compound statements*. We also define the *nullary action* stop, which appears only at runtime and intuitively marks the end of fully executing a statement.

Program states. A program state s is a pair consisting of

- a *variable valuation* σ_s that maps a thread id and a variable to a value, such that $\sigma_s(t, g) = \sigma_s(u, g)$ for all states s and thread id's t, u, whenever g is a global variable.
- a *code map* ϵ_s that keeps track of a (compound) statement for each thread, such that $\epsilon_s(t) = c$ means that at program state s, the remaining part of the program to be executed by thread t is given by c.

A program state s is called *initial* if $\forall t \in Tid. \exists \rho \in Proc. \epsilon_s(t) = body_\rho$, i.e. every thread is about to call a procedure. State s is called *final* if $\epsilon_s(t) = $ stop, for all $t \in Tid$. We write $Initial(s)$ (resp. $Final(s)$) to denote that s is an initial (resp. final) state.

Let $\sigma_s|_V$ denote the projection of valuation σ_s on $V \subseteq Var$. Define $s|_V$ to be the program state $(\sigma_s|_V, \epsilon_s)$. This definition also pointwise applies to collections of states.

Predicates over program variables. For an assert predicate x, let $x[t]$ denote the predicate in which all free occurrences of tid is replaced with t. We say that a program state s satisfies $x[t]$, denoted as $s \vDash x[t]$ or as $x[t](s)$, if $x[t]$ evaluates to true when all free occurrences of each unprimed variable v is replaced with $\sigma_s(t, v)$. An assert predicate is called a *state predicate* if it does not contain any free occurrence of tid.

Similarly, the pair of program states (s_1, s_2) satisfies a transition predicate $p[t]$, denoted as $(s_1, s_2) \vDash p[t]$ or as $p[t](s_1, s_2)$, if $p[t]$ evaluates to true when each unprimed variable v (resp. v') is replaced with $\sigma_{s_1}(t, v)$ (resp. $\sigma_{s_2}(t, v)$).

Let $\mathsf{fv}(p)$ be the set of free variables in the (state or transition) predicate p.

Execution semantics. We assume a sequentially-consistent memory model. For thread t and $\gamma \in Atom$, (t, γ) is called a *transition label*. We say $s \xrightarrow{(t,\alpha)} s'$ holds when t can execute α next (in which case s' is a (t, α) successor[3] of s), all other threads do not update their control flow, all local variables of other threads remain the same, the global variables and local variables of t are updated so that the transition predicate of α is satisfied. Formally, $s \xrightarrow{(t,\alpha)} s'$ if $(s, s') \vDash \tau_\alpha[t]$ and for all $u \neq t$ and for any local variable x, $\sigma_s(u, x) = \sigma_{s'}(u, x)$.

Run. A run r of the program is a sequence of state transitions:

$$r = r_1 \xrightarrow{(t_1, \alpha_1)} r_2 \xrightarrow{(t_2, \alpha_2)} \cdots \xrightarrow{(t_{n-1}, \alpha_{n-1})} r_n$$

For the definitions that follow, we fix the run r above. Let $Tid(r)$ denote the set of threads occurring in r. Let r_i denote the ith program state, and $r(i)$, the ith transition label (t_i, α_i) in r. For a state predicate ϕ, we say that r is a run of P from ϕ if $Initial(r_1)$ and $r_1 \vDash \phi$.

The run is *maximal* if r_n cannot make any transition. Henceforth, we will only consider maximal runs.

Trace. A *trace* is a sequence of transition labels, $\mathbf{l} = l_1 \ldots l_k$. The trace moves a state s_1 to s_{k+1}, written $s_1 \xrightarrow{\mathbf{l}} s_{k+1}$, if there is a run r of P over $\mathbf{l}$, such that $r_j = s_j$, for all $1 \leq j \leq k + 1$ and $r_i \xrightarrow{l_i} r_{i+1}$.

Violation-freedom. A run r of P from ϕ is called a *violation* if $\neg\phi_\alpha[t](r_k)$ evaluates to true for some $(t, \alpha) \in \mathsf{next}(r_k)$. Intuitively, a violation is a run of P that starts from an initial program state s_1 and reaches a program state s_k which violates the assert predicate, ϕ_α, of an action α which thread t can execute at state s_k. A run is said to be *successful* if it is not a violation. We indicate a successful run as $s_1 \xrightarrow{\mathbf{l}} s_2$ and a violation as $s_1 \xrightarrow{\mathbf{l}} \mathsf{error}$.

4   Program Transformations

In this section, we formalize our notion of proof and introduce the rules for the proof calculus. A *proof state* is the pair $(P, \mathcal{I})$, where P is a program, and $\mathcal{I}$ is a state predicate, called the *inductive invariant* of the program. We require that for every proof state $(P, \mathcal{I})$, all the atomic actions of P preserve $\mathcal{I}$. An atomic action α preserves $\mathcal{I}$, written $\alpha \leftrightarrows \mathcal{I}$, if $s_1 \xrightarrow{(t,\alpha)} s_2$ and $s_1 \vDash \mathcal{I}$ imply $s_2 \vDash \mathcal{I}$.

A proof consists of rewriting the input program, denoted P_1, iteratively so that, in the limit, one arrives at a program, denoted P_n, that can be verified by sequential reasoning methods. Formally, the proof is expressed as $(P_1, \mathsf{true}) \dashrightarrow (P_2, \mathcal{I}_2) \dashrightarrow \cdots \dashrightarrow (P_n, \mathcal{I}_n)$ Each proof step is governed by a proof rule, which we present below.

---

[3] Our technical report [18] contains a more elaborate discussion of the operational semantics of our formal language.

The following proof rule states the general form of updating $\mathcal{I}$, replacing it with a stronger invariant.

Rule 1 (Invariant). *Replace invariant $\mathcal{I}_1$ with $\mathcal{I}_2$ if $\alpha \leftrightarrows \mathcal{I}_2$ for all the actions α in P, and $\mathcal{I}_2 \Rightarrow \mathcal{I}_1$.*

The basic idea in reduction and abstraction is to replace an action with another action that simulates the former.

Definition 1 (Simulation). *Let α, β be actions, t be an arbitrary thread id. We say β simulates α at proof state $(P, \mathcal{I})$, written $(P, \mathcal{I}) \vdash \alpha \preceq \beta$, if both of the following hold:*

$$\textbf{S1.} \quad (\mathcal{I} \wedge \neg\phi_\alpha) \Rightarrow \neg\phi_\beta \qquad \textbf{S2.} \quad (\mathcal{I} \wedge \tau_\alpha) \Rightarrow (\neg\phi_\beta \vee \tau_\beta)$$

Intuitively, **S1** states that if there is a violation with α, there has to be a violation with β substituted in place of α. **S2** states that for each violation-free run, replacing α with β results in either a violation, or a violation-free run with the same end state.

4.1 Reduction

Reduction, due to Lipton [19], creates coarse-grained atomic statements by combining fine-grained actions. An action α can be combined with another action if α is a certain kind of mover. A mover is an action that can commute over actions of other threads in any run. We write $(P, \mathcal{I}) \vdash \alpha : m$ to indicate that α is m−mover in the proof state $(P, \mathcal{I})$, where $m \in \{\mathbb{L}, \mathbb{R}\}$.

We decide that an action α is a mover by statically checking a simulation relation, that states that commuting α with every β can lead to the same state or goes wrong. An assert predicate x is *p-stable*, if $\forall s, s'. x(s) \wedge p(s, s') \Rightarrow x(s')$.

Let $\mathsf{wp}(p, x)$, the *weakest (liberal) pre-condition* of predicate x for transition predicate p, stand for all states which cannot reach a state where x evaluates to false after executing p. Formally, $\mathsf{wp}(p, x) = \{s \mid \forall s'. p(s, s') \Rightarrow x(s')\}$. For two transition predicates p and q, define their composition $p \cdot q$, as the transition predicate $p \cdot q = \{(s_1, s_2) \mid \exists s_3. p(s_1, s_3) \wedge q(s_3, s_2)\}$. The operator $[\![\ ]\!]$ expresses the result of combining two actions to one atomic action.[4]

$$[\![\alpha; \beta]\!] = \mathsf{assert}\,(\phi_\alpha \wedge \mathsf{wp}(\tau_\alpha, \phi_\beta)); (\tau_\alpha \cdot \tau_\beta) \qquad [\![\alpha \square \beta]\!] = \mathsf{assert}\,(\phi_\alpha \wedge \phi_\beta); (\tau_\alpha \vee \tau_\beta)$$

Definition 2 (Left-mover). *Action α is a left-mover in proof state $(P, \mathcal{I})$, denoted $(P, \mathcal{I}) \vdash \alpha : \mathbb{L}$, if the following holds for every action β in P and every pair of distinct thread ids t and u: $(P, \mathcal{I}) \vdash [\![\beta[u]\ ;\ \alpha[t]]\!] \preceq [\![\alpha[t]\ ;\ \beta[u]]\!]$.*

Definition 3 (Right-mover). *Action α is a right-mover in proof state $(P, \mathcal{I})$, denoted $(P, \mathcal{I}) \vdash \alpha : \mathbb{R}$, if, for every action β in P, and every pair of distinct thread ids t and u: $(P, \mathcal{I}) \vdash [\![\alpha[t]\ ;\ \beta[u]]\!] \preceq [\![\beta[u]\ ;\ \alpha[t]]\!]$ and $\phi_\beta[u]$ is $\tau_\alpha[t]$-stable.*

---

[4] We assume that a transition predicate $\tau_\alpha[t]$ can only change the variables in the scope of t and that if t and u are running the same procedure, local variables are suitably renamed to prevent false conflicts.

The reduction rules below define the conditions under which non-atomic statements are transformed to atomic actions. We omit the rules about procedure calls and parallel composition which are similar to those of [4].

Rule 2 (Reduce-Sequential). *Replace occurrences of $\alpha \; ; \; \gamma$ with $[\![\alpha \; ; \; \gamma]\!]$ if either $(P, \mathcal{I}) \vdash \alpha : \mathbb{R}$ or $(P, \mathcal{I}) \vdash \gamma : \mathbb{L}$.*

Rule 3 (Reduce-Choice). *Replace occurrences of $\alpha \; \square \; \gamma$ with $[\![\alpha \; \square \; \gamma]\!]$.*

Rule 4 (Reduce-Loop). *Replace occurrences of $\alpha^{\circlearrowright}$ with β if the following hold:*

    L1. $(P, \mathcal{I}) \vdash \alpha : m \; s.t. \; m \in \{\mathbb{R}, \mathbb{L}\}$ **L2**. $\beta \leftrightarrows \mathcal{I}$

    L3. $\phi_\beta \Rightarrow \tau_\beta[\mathit{Var}/\mathit{Var}']$ **L4**. $(P, \mathcal{I}) \vdash [\![\beta \; ; \; \alpha]\!] \preceq \alpha$

4.2 Abstraction

The purpose of the abstraction rule is to replace an action with another action An abstraction step consists of replacing an action α with another action β, which in principle leads to less interference with other actions.

Rule 5 (Abstraction). *Replace the action α with action β if $\beta \leftrightarrows \mathcal{I}$ and $(P, \mathcal{I}) \vdash \alpha \preceq \beta$.*

This rule is usually applied for an action $\mathsf{assert}\, a; p$ by replacing it with 1) $\mathsf{assert}\, b; p$ such that $b \Rightarrow a$ or 2) with $\mathsf{assert}\, a; q$ such that $p \Rightarrow q$. While the former corresponds to adding extra assertions to the action, the latter adds more (non-deterministic) transitions.

4.3 Variable Introduction and Hiding

Intuitively, variable introduction rewrites some actions in the program so that these can refer to a new (*history*) variable. Variable hiding is the dual of variable introduction; each action is rewritten so that it does no longer refer to the hidden variable. Hiding a variable also requires quantifying out the variable in the invariant.

In order to ensure soundness, in both cases, we need a relation between actions over different sets of variables. For this, we extend our simulation relation ($\preceq$) for each rule. In addition, we require that the input and output variables of the procedures ($\overrightarrow{in}_\rho, \overrightarrow{out}_\rho$) are fixed during the proof; the rules below are not applicable to these variables.

Rule 6 (Add-Variable). *Add the new variable v to Var_P, and replace every action α with β whenever $(P, \mathcal{I}) \vdash \alpha \preceq_{+v} \beta$, which holds if the following are both valid:*

    A1. $(\mathcal{I} \wedge \neg\phi_\alpha) \Rightarrow (\forall v. \; \neg\phi_\beta)$ **A2**. $(\mathcal{I} \wedge \tau_\alpha) \Rightarrow (\forall v. \; \neg\phi_\beta \vee (\exists v'. \; \tau_\beta))$

Rule 7 (Hide-Variable). *Remove the existing variable v from the program, and replace the invariant $\mathcal{I}$ with $\exists v.\ \mathcal{I}$. Replace every action α with β whenever $(P, \mathcal{I}) \vdash \alpha \preceq_{-v} \beta$, which holds if the following are both valid:*

> **H1**. $(\exists v.\ \mathcal{I} \wedge \neg\phi_\alpha) \Rightarrow \neg\phi_\beta$ **H2**. $(\exists v, v'.\ \mathcal{I} \wedge \tau_\alpha) \Rightarrow (\neg\phi_\beta \vee \tau_\beta)$

Fix a thread t and a state s. In both of the rules, the first condition (**A1**, **H1**) states that violations are preserved. The second condition (**A2**, **H2**) states that transitions (over the common variables of α and β) are either preserved or additional violations are introduced.

5   Soundness Theorems

Given a proof $(P_1, \mathcal{I}_1) \dashrightarrow \cdots \dashrightarrow (P_n, \mathcal{I}_n)$, we now provide the soundness theorems. Each theorem relates P_n to P_1, providing a soundness guarantee for a particular use of our method. Due to lack of space, we provide the proofs in our technical report [18].

5.1   Proving Assertions

The first theorem is an extension of the main soundness theorem in [4]. Intuitively, the theorem states that proof steps preserve violations, and initial-final state pairs when the output program is good from the final invariant.

Good and Bad. In the following, we define $Good(P, \mathcal{I})$ as the set of pre- and post-state pairs associated with succeeding (maximal) runs of program P from states satisfying $\mathcal{I}$. $Bad(P, \mathcal{I})$ is the set of pre-states associated with violations. Formally,

$$Good(P, \mathcal{I}) = \{(s_1, s_2) \mid Initial(s_1),\ s_1 \vDash \mathcal{I},\ \exists l.\ s_1 \xrightarrow{\ l\ } s_2,\ Final(s_2)\}$$
$$Bad(P, \mathcal{I})\ = \{s_1 \mid Initial(s_1),\ s_1 \vDash \mathcal{I},\ \exists l.\ s_1 \xrightarrow{\ l\ } \mathsf{error}\}$$

P is said to be *good* from $\mathcal{I}$ if $Bad(P, \mathcal{I}) = \emptyset$; it is called *bad* from $\mathcal{I}$, otherwise.

Theorem 1. *Let $(P_1, \mathcal{I}_1) \dashrightarrow \cdots \dashrightarrow (P_n, \mathcal{I}_n)$ be a sequence of proof steps. Let $V = Var_{P_1} \cap Var_{P_n}$ and $X = (Var_{P_1} \cup Var_{P_n}) \backslash V$. The following hold:*

> **C1**. $Bad|_V(P_1, \exists X.\ \mathcal{I}_n) \subseteq Bad|_V(P_n, \exists X.\ \mathcal{I}_n)$
> **C2**. $\forall (s_1, s_n) \in Good|_V(P_1, \exists X.\ \mathcal{I}_n)$:
>   **a.** $s_1 \in Bad|_V(P_n, \exists X.\ \mathcal{I}_n)$ *or*   **b.** $(s_1, s_n) \in Good|_V(P_n, \exists X.\ \mathcal{I}_n)$

Note that, since the input and output variables of procedures are fixed during the proof, so the set V above will always be nonempty. A corollary of the above theorem is that, if P_n is good from $\mathcal{I}_n$, then P_1 is good from $\mathcal{I}_n$. This means that, one can prove the assertions in P_1 by gradually obtaining programs with coarser-grained concurrency using our proof rules.

5.2   Proving Linearizability

In this section, we establish a link between P_1 and P_n in the context of proving linearizability. For this, we first define *behavioral simulation*, a special kind of simulation that relates two programs through their observable behaviors over procedure input and output values.

Behavioral simulation. Let $r = s_1 \xrightarrow{1} s_n$ be a (maximal) run of the program. Let ρ be the procedure executed by t. We call the tuple $(t, \rho, \sigma_{s_1}(t, \overrightarrow{in}_\rho), \sigma_{s_n}(t, \overrightarrow{out}_\rho))$ the behavior of t in r and denote it by $\mathsf{beh}(r, t)$. The behavior includes the name of the procedure called by t, along with the values of the input and the output variables of the procedure[5]. We write $\mathsf{Beh}(r)$ to denote $\{\mathsf{beh}(r, t) \mid t \in Tid(r)\}$.

We define $\mathsf{fst}(r, t)$ and $\mathsf{lst}(r, t)$ be the indices of first and the last actions of t in r. Formally, with $L = \{i \mid r(i) = (t, \alpha)\}$, $\mathsf{fst}(r, t) = \min(L)$ and $\mathsf{lst}(r, t) = \max(L)$. Let $\ll_r$ be a partial order over $Tid(r)$ ordering threads that do not execute concurrently: $t \ll_r u$ if $\mathsf{lst}(r, t) < \mathsf{fst}(r, u)$.

Definition 4. *Let P and P' be two programs with $\mathsf{Sig}(P) = \mathsf{Sig}(P')$, and let $\mathcal{I}$ be a state predicate. Let $X_1 = \mathsf{fv}(\mathcal{I}) \setminus Var_P$ and $X_2 = \mathsf{fv}(\mathcal{I}) \setminus Var_{P'}$. P' behaviorally-simulates P from $\mathcal{I}$, denoted $P \lhd_\mathcal{I} P'$ if for each maximal run r of program P from $\exists X_1.\mathcal{I}$, there exists a maximal run r' of P' from $\exists X_2.\mathcal{I}$ such that 1) $\mathsf{Beh}(r) = \mathsf{Beh}(r')$ and 2) $\ll_r \subseteq \ll_{r'}$*

The following theorem connects behavioral simulation to the generic notion of linearizability. We say *P is linearizable to P' from $\mathcal{I}$* to restrict the definition of linearizability to runs of P and P' from $\mathcal{I}$. A program P is called an *atomic program* if for every $\rho \in Proc_P$, $body_\rho$ is an atomic action.

Theorem 2. *Let P' be an atomic program that is good from $\mathcal{I}$. A program P is linearizable to P' from $\mathcal{I}$ iff $P \lhd_\mathcal{I} P'$.*

The following theorem states that each good program reached during the proof behaviorally simulates the initial program.

Theorem 3 (Soundness). *Let $(P_1, \mathcal{I}_1) \dashrightarrow \cdots \dashrightarrow (P_n, \mathcal{I}_n)$ be a sequence of proof steps such that P_n is good from $\mathcal{I}_n$. Then forall $1 \leq i \leq n$, $P_1 \lhd_{\mathcal{I}_n} P_i$ holds.*

Theorems 2 and 3 provide two options for proving linearizability of P_1 to the intended specification from $\mathcal{I}$, represented by an atomic program P_n. First, one can complement another proof method with ours, by first performing the proof $(P_1, \mathsf{true}) \dashrightarrow \cdots \dashrightarrow (P_k, \mathcal{I})$, and then applying her method to prove that P_k is linearizable to P_n. Once the proof passes, this implies that P_1 is also linearizable to P_n, since our transformations preserve all the behaviors of the program relevant to linearizability. Alternatively, s/he can keep transforming $(P_k, \mathcal{I})$ up to $(P_n, \mathcal{I})$, and complete the full proof of linearizability in our system. Note that, for the theorems to ensure soundness in these cases, s/he must also prove that P_k (resp. P_n) is good from $\mathcal{I}$. The latter is formalized by the following.

---

[5] Notice that the first and the last states of the run provide us the values of $\overrightarrow{in}_\rho$ and $\overrightarrow{out}_\rho$, respectively.

Corollary 1. *Let* $(P_1, \mathsf{true}) \dashrightarrow \cdots \dashrightarrow (P_n, \mathcal{I})$ *be a sequence of proof steps, such that* P_n *is an atomic program that is good from* $\mathcal{I}$. *Then,* P_1 *is linearizable to* P_n *from* $\mathcal{I}$.

6 Implementation and Experience

We implemented our proof method in the QED verifier. QED accepts as input a multithreaded program written in an extension of the Boogie programming language and a proof script. All the transformations are applied automatically, and when necessary, the preconditions of the transformations are checked, by generating verification conditions and feeding them to the Z3 SMT solver. Using QED, we mechanically proved the linearizability of the following programs:

- Lock-coupling linked list [13]
- Treiber's non-blocking stack [20]
- Non-blocking and two-lock queues [21]
- Non-blocking mutex lock implementation adapted from [22]

For each data structure, we chose a generic specification as the target of the proof, and were able to transform the program to the specification program through few reduction and refinement phases. The QED tool and the proof scripts of the above programs are available at **http://qed.codeplex.com**.

In the rest of the section, we overview the proof of the non-blocking queue, and describe how coupling variable introduction and hiding helps us to cope with superficial conflicts. This is an important limitation for reduction, and interestingly, our standard notion of abstraction on the existing variables (Section 4.2) does not help in this situation. Our solution to eliminating the conflict is to hide the variables on which the conflict happens; but, differently from the standard abstraction, introducing new variables, which will carry enough (semantic) information from the hidden variables and will not cause conflicts.

6.1 Non-Blocking Queue

Figure 2 shows the version of the non-blocking queue [21] after applying a reduction phase on the original implementation. Atomic action `Do_Dequeue` removes an element from the queue, and `Do_Enqueue` appends a new element to the queue. The implementation is lazy in that `Do_Enqueue` does not update the `Tail` variable after adding the new node. As a result, at any time `Tail` may point to any node between `Head` and `null`. The actions labeled `Move_Tail` and `Update_Tail` try to move the `Tail` towards the end of the list. This resembles relaxed balancing in concurrent implementation of tree-like data structures, in which restructuring the data structure is separated from actual operations, and delayed.

The predicate `Reach(next,k,l,m)` expresses that, from node `k`, following zero or more `next` pointers, we first reach `l` and then `m` [23]. The `Reach` predicate gives us the ability to do simple abstractions on actions accessing the list nodes. For

Implementation (*Impl*)

```
record Node { data: int; next: Node; }          Enqueue(x: int)
var Head, Tail: Node;                             var node, tail: Node;
                                                  atomic {
Dequeue() returns (x: int)                         node := new Node(x);
 var tail: Node;                                    node.next := null;
 while(true) {                                      }
Move_Tail: atomic {                                while(true) {
    havoc tail;                                   Move_Tail:atomic {
    assume Reach(next, Tail, tail, null)              havoc tail;
         && tail != null;                             assume Reach(next, Tail, tail, null)
    Tail := tail;                                             && tail != null;
  }                                                   Tail := tail;
 } // end while                                       }
Do_Dequeue: atomic {                               } // end while
 if (Head.next == null) {                          Do_Enqueue: atomic {
  x := null;                                         assume (Tail.next == null);
 } else {                                            Tail.next := node;  tail := Tail;
  assume (Head != Tail);                             }
  Head := Head.next;  x := Head.data;              Update_Tail: atomic {
 }                                                    if (Tail == tail) Tail := node;
}                                                    }
```

Specification (*Spec*)

```
atomic Dequeue() returns (x: int)                atomic Enqueue(x: int)
 if (Head.next == null) {                          node := new Node(x);
  return null;                                      node.next := null;
 } else {                                           _Tail.next := node;
  Head := Head.next; x := Head.data;               _Tail := _Tail.next;
 }
```

Fig. 2. The reduced implementation of the non-blocking queue and its specification

example, a former abstraction step in the reduction phase replaces the action `n := tail.next` with the action `havoc n; assume Reach(next,tail,n,n)`; while the former is not mover, the latter is.

In order to apply reduction, the only option is to show that `Move_Tail` is a right-mover, since `Do_Enqueue` and `Do_Dequeue` perform the actual operations, thus are not movers. `Move_Tail` conflicts with `Do_Enqueue` and `Do_Dequeue` on `Tail`. Notice that `Move_Tail` performs an internal operation that does not affect the semantics of the queue. Thus, these conflicts are superficial. Havocing `Tail` in the conflicting actions, or hiding `Tail` are a valid proof steps, and would make reduction pass. However, the resulting code would perform incorrect operation.

We eliminate the conflict by coupling the hiding of `Tail` with introducing the history variable `_Tail` of the same type. Differently from `Tail`, `_Tail` always points to the end of the queue. We then associate the existing variables with the new variable `_Tail` by the following invariant.

```
Reach(next, Head, Tail, Tail) && Reach(next, Tail, _Tail, null)
                        && (_Tail != null) && (_Tail.next == null)
```

In order to satisfy the invariant, we add to the end of `Do_Enqueue` the assignment `_Tail := _Tail.next`. Once there is `_Tail` to keep track of the end of the list, we are ready to hide `Tail`. This is done by replacing the actions in the program with actions that do not refer to `Tail`, but now uses `_Tail` to access the end of the linked list. Figure 3 shows the version of the program after hiding `Tail`. Notice

```
record Node { data: int; next: Node; }        procedure Enqueue(x: int)
var Head, _Tail: Node;                          var node, tail: Node;
                                                atomic {
procedure Dequeue()                              node := new Node(x);
 var tail: Node;                                 node.next := null;
                                                }
 while(true) {
Move_Tail: atomic { havoc tail; }                while(true) {
 } // end while                                 Move_Tail: atomic { havoc tail; }
                                                 } // end while
Do_Dequeue: atomic {
  if (Head.next == null) {                      Do_Enqueue: atomic {
   x := null;                                      _Tail.next := node;
  } else {                                         _Tail := _Tail.next;
   Head := Head.next;  x := Head.data;             havoc tail;
  }                                                }
 }                                              Update_Tail: atomic { assume true; }
```

Fig. 3. The version of the non-blocking queue after hiding `Tail`

that the new form of Move_Tail does not perform any semantic operation in the new program, and does not conflict with other actions. In addition, the actions Do_Enqueue and Do_Dequeue now use _Tail to correctly perform their operations.

The hiding step also includes existentially quantifying Tail in the invariant given above. This produces the following invariant for the new program.

```
Reach(next, Head, _Tail, null) && (_Tail != null) && (_Tail.next == null)
```

We proceed with a reduction phase that combines the blocks into a single action for each operation. The combined operations, together with the above invariant (for simplicity, we omit parts of the representation invariant), give the correct behavior of a sequential queue implementation. Corollary 1 ensures that the original implementation in [21] is linearizable to this final program from the invariant. Note that it also possible to continue the proof with an extra refinement phase to prove the linearizability to a more generic specification of the queue.

References

1. Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. 12, 463–492 (1990)
2. Abadi, M., Lamport, L.: The existence of refinement mappings. Theor. Comput. Sci. 82, 253–284 (1991)
3. Park, S., Dill, D.L.: Protocol verification by aggregation of distributed transactions. In: Alur, R., Henzinger, T.A. (eds.) CAV 1996. LNCS, vol. 1102, pp. 300–310. Springer, Heidelberg (1996)
4. Elmas, T., Qadeer, S., Tasiran, S.: A calculus of atomic actions. In: POPL 2009: ACM Symposium on Principles of Programming Languages. ACM, New York (2009)
5. Vafeiadis, V.: Shape-value abstraction for verifying linearizability. In: Jones, N.D., Müller-Olm, M. (eds.) VMCAI 2009. LNCS, vol. 5403, pp. 335–348. Springer, Heidelberg (2009)

6. Hesselink, W.H.: Eternity variables to prove simulation of specifications. ACM Trans. Comput. Logic 6, 175–201 (2005)
7. Jonsson, B., Pnueli, A., Rump, C.: Proving refinement using transduction. Distrib. Comput. 12, 129–149 (1999)
8. Kesten, Y., Pnueli, A., Shahar, E., Zuck, L.D.: Network invariants in action. In: Brim, L., Jančar, P., Křetínský, M., Kucera, A. (eds.) CONCUR 2002. LNCS, vol. 2421, pp. 101–115. Springer, Heidelberg (2002)
9. Hendler, D., Shavit, N., Yerushalmi, L.: A scalable lock-free stack algorithm. In: SPAA 2004: ACM symposium on Parallelism in algorithms and architectures, pp. 206–215. ACM, New York (2004)
10. Gao, H., Groote, J.F., Hesselink, W.H.: Lock-free dynamic hash tables with open addressing. Distrib. Comput. 18, 21–42 (2005)
11. Colvin, R., Groves, L., Luchangco, V., Moir, M.: Formal verification of a lazy concurrent list-based set algorithm. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 475–488. Springer, Heidelberg (2006)
12. Owicki, S., Gries, D.: Verifying properties of parallel programs: an axiomatic approach. Commun. ACM 19, 279–285 (1976)
13. Vafeiadis, V., Herlihy, M., Hoare, T., Shapiro, M.: Proving correctness of highly-concurrent linearisable objects. In: PPoPP 2006 ACM Symposium on Principles and practice of parallel programming, pp. 129–136. ACM, New York (2006)
14. Amit, D., Rinetzky, N., Reps, T.W., Sagiv, M., Yahav, E.: Comparison under abstraction for verifying linearizability. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 477–490. Springer, Heidelberg (2007)
15. Wang, L., Stoller, S.D.: Static analysis for programs with non-blocking synchronization. In: ACM SIGPLAN 2005 Symposium on Principles and Practice of Parallel Programming (PPoPP). ACM Press, New York (2005)
16. Groves, L.: Verifying michael and scott's lock-free queue algorithm using trace reduction. In: CATS 2008: Symposium on Computing: the Australasian theory, Darlinghurst, Australia, pp. 133–142. Australian Computer Society, Inc. (2008)
17. Elmas, T., Sezgin, A., Tasiran, S., Qadeer, S.: An annotation assistant for interactive debugging of programs with common synchronization idioms. In: Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging (2009)
18. Elmas, T., Qadeer, S., Sezgin, A., Subasi, O., Tasiran, S.: Simplifying the proof of linearizability with reduction and abstraction. Technical Report, Koc University (2009), http://theorem.ku.edu.tr/tacas10tr.pdf
19. Lipton, R.J.: Reduction: a method of proving properties of parallel programs. Commun. ACM 18, 717–721 (1975)
20. Treiber, R.K.: Systems programming: Coping with parallelism. rj5118 (1986)
21. Michael, M.M., Scott, M.L.: Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In: PODC 1996: ACM symposium on Principles of distributed computing, pp. 267–275. ACM, New York (1996)
22. Krieger, O., Stumm, M., Unrau, R., Hanna, J.: A fair fast scalable reader-writer lock. In: ICPP 1993: The International Conference on Parallel Processing, Washington, DC, USA, pp. 201–204. IEEE Computer Society, Los Alamitos (1993)
23. Lahiri, S., Qadeer, S.: Back to the future: revisiting precise program verification using smt solvers. In: POPL 2008: ACM symposium on Principles of programming languages, pp. 171–182. ACM, New York (2008)