Back and Forth: Prophecy Variables for Static Verification of Concurrent Programs

October 13, 2009

Technical Report
MSR-TR-2009-142

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

This page intentionally left blank.

Back and Forth: Prophecy Variables for Static Verification of Concurrent Programs

Shaz Qadeer
Microsoft Research
qadeer@microsoft.com

Ali Sezgin Serdar Tasiran
Koc University
asezgin@ku.edu.tr stasiran@ku.edu.tr

Abstract

Several static proof systems have been developed over the years for verifying shared-memory multithreaded programs. These proof systems make use of auxiliary variables to express mutual exclusion or non-interference among shared variable accesses. Typically, the values of these variables summarize the past of the program execution; consequently, they are known as history variables. Prophecy variables, on the other hand, are the temporal dual of history variables and their values summarize the future of the program execution. In this paper, we show that prophecy variables are useful for locally constructing proofs of systems with optimistic concurrency. To enable the fullest use of prophecy variables in proof construction, we introduce tressa annotations, as the dual of the well-known assert annotations. A tressa claim states a condition for reverse reachability from an end state of the program, much like an assert claim states a condition for forward reachability from the initial state of the program.

We present the proof rules and the notion of correctness of a program for two-way reasoning in a static setting: forward in time for assert claims, backward in time for tressa claims. Even though the interaction between the two is non-trivial, the formalization is intuitive and accessible. We demonstrate how to verify implementations based on optimistic concurrency which is a programming paradigm that allows conflicts to be handled after they occur. We have incorporated our proof rules into the QED verifier and have used our implementation to verify a handful of small but sophisticated algorithms. Our experience shows that the proof steps and annotations follow closely the intuition of the programmer, making the proof itself a natural extension of implementation.

1. Introduction

The main challenge in proving a concurrent program is reasoning about interactions among threads on the shared memory. In a proof based on validating assertions that specify a program's desired behavior, one has to consider all possible interleavings of conflicting operations. Most existing methods verify programs at the finest level of granularity of atomic actions: only actions guaranteed to be executed without interruption by the runtime are considered to be atomic. At this level of granularity, there are a large number of possible interleavings. Proving the program at this level requires one to consider concurrency- and data-related properties at the same time and this results in complicated proofs.

We have recently developed a static verification method called QED [1] that alleviates this complexity. A proof in QED consists of rewriting the input program iteratively using abstraction and reduction so that, in the limit, one arrives at a program that can be verified by sequential reasoning methods. Reduction, due to [2], creates coarse-grained atomic statements from fine-grained ones. Whether statements can be thus combined depends on their *mover types*. Abstraction of a statement allows us to reason that it does not interfere with other atomic statements. Adding assertions over auxiliary *history* variables or relaxing transitions are two main abstraction methods. We will have more to say on these in Sec. 2. In short, though, as we shall see, abstraction leads to reduction which in turn may enable more abstraction. This proof method is supported by a tool also called QED. The QED tool provides a set of intuitive, concise and machine checked proof commands.

For sample implementations based on optimistic concurrency, our experience with QED suggests that expressing facts about concurrency control mechanisms in the form of assertions over history variables is unnatural and counter-intuitive. Correct operation of optimistic concurrency implementations, used in the implementation of non-blocking data structures or Software Transactional Memories (STM's) [3], do not depend on exclusive access to shared variables. The idea is to carry out computation *as if* no interference will occur and then, prior to termination, check whether this assumption is correct. If it is, then simply *commit*; if not, *roll-back* any visible global change and, optionally, re-start. In this case, it is not the prefix of the execution that leads to the statement that summarizes its interference, but rather, the suffix of the execution that leads from the statement.

Prophecy variables, the temporal dual of history variables, are ideal for this kind of reasoning. They are used to select at a state q a subset of all execution segments from q onwards. To the best of our knowledge, research on using prophecy variables so far has concentrated on execution-based refinement proofs (e.g., [4]). The auxiliary variables allowed in static proof systems are exclusively history variables.

In this paper, we incorporate prophecy variables as a new class of auxiliary variables into the static QED proof system for concurrent programs. Along with prophecy variables, in order to achieve their fullest use in proof construction, we introduce tressa annotations, as the dual of the well-known assert annotations. A tressa claim states a condition for reverse reachability from an end state of the program, much like an assert claim states a condition for forward reachability from the initial state of the program. Annotating actions with prophecy variables allows information about the rest of the execution to be used in deciding the mover types of actions which are checked locally. A tressa claim stating that an action followed by another action cannot lead to a final state of the program

because it contradicts with the current value of a prophecy variable becomes very useful in locally constructing proofs of systems with optimistic concurrency.

We present the proof rules and the notion of correctness of a program for two-way reasoning in a static setting: forward in time for assert claims, backward in time for tressa claims. Building on our initial work [1], we reformulate simulation and mover definitions, valid for both forward and backward reasoning. Even though the interaction between the two is non-trivial, the formalization is intuitive and accessible. We demonstrate how to verify a handful of small but sophisticated algorithms based on optimistic concurrency.

Related Work. Prophecy variables were introduced in [5] in the context of refinement proofs. They were used to define refinement mappings between specification and its implementation in cases where the mapping between abstract and concrete states depends on the rest of the execution. Subsequent work on prophecy variables were almost exclusively on refinement checking (e.g., [4]). Static verification is a well-known technique for concurrent program verification (e.g., [6, 7, 8]). A variety of techniques have been proposed for static verification of concurrent programs (e.g., [6, 7, 9, 8]). Some work on static verification use reduction as the key ingredient (e.g., [10, 11, 12]). However, the only work on prophecy variables in static verification we know of is by Marcus and Pnueli [13]. In the context of a static method for proving refinement between two transition systems, the authors present two sound ways of augmenting a sequential program with assignments that involve temporal logic formulas with future operators. Their soundness condition for annotating programs with auxiliary variables is, as expected, similar to ours. In contrast, our proof system targets concurrent software and the verification of assertions rather than refinement, and uses atomicity as a key reasoning tool.

Roadmap. In the next section, we briefly highlight QED method. We give a semi-formal description of reduction, abstraction and how they interact. We then give an example for which the current proof rules do not give an immediate solution. We demonstrate how prophecy variables can help in reduction. In Sec. 4, we formalize our framework, describing the programming language syntax and semantics. For ease of presentation, we only use a subset of QED. In Sec. 5, we formalize prophecy variables, define the new mover conditions, and state the soundness theorem. In Sec. 6, we show in detail how to reason and use prophecy variables and tressa annotations in the proof of implementations using optimistic concurrency. We finish with concluding remarks.

2. QED- An Informal Summary

In this section, we will briefly describe the QED method (for a detailed account, please see [1]) . This section can be skipped by those who are familiar with QED.

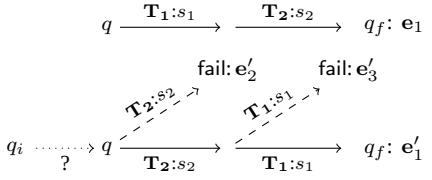
2.1 Reduction, Movers, Simulation

Reduction, the starting point of QED, combines sequentially composed atomic actions of appropriate mover types into a single atomic action. Consider two sequentially composed actions $s_1; s_2$. Now imagine that in any execution, s_1 executed by a thread t at state q followed by an arbitrary action s' executed by a different thread u reaches state q' implies that from the state q, s' executed by u followed by s_1 executed by t reaches the state q'; that is, from the same initial state, the same end state remains reachable after swapping the order of execution of the two actions. In such a case, s_1 is a *right-mover*, because, in any execution, it can commute to the right of any other statement without changing the end state. It is then sound to treat $s_1; s_2$ as a single atomic action. The mover type of an action depends on the existence of *conflicting* actions: pairs of actions accessing the same shared variable with at least one of them writing to it. Actions with only thread-local accesses are both left and right movers. Lock acquires are right-movers, since they cannot be immediately followed by an acquire or release of the same lock by another thread. Similarly, lock releases are left-movers.

```
...                                  ...
 acq(lock); //sets lock==tid          atomic { acq(lock);
                                               assert lock==tid; }
 g := x;                              atomic { assert lock==tid;
                                               g := x; }
 rel(lock); //sets lock==0            atomic { assert lock==tid;
                                               rel(lock); }
...                                  ...
```

Figure 1. Enabling commuting by adding assertions.

The QED method improves this idea of reduction by relaxing the requirement that the same end state be reached from every initial state. This is formalized as a simulation relation: as long as s_1 followed by s' is simulated by s' followed by s_1, for arbitrary s', action s_1 is a right-mover. The simulation relation used by QED is illustrated below.



The topmost line represents a pair of actions executed one after the other: from state q, s_1 by $\mathbf{T}_1$ followed by s_2 by $\mathbf{T}_2$ reaches state q_f. We say that $\mathbf{T}_1{:}s_1,\mathbf{T}_2{:}s_2$ is simulated by $\mathbf{T}_2{:}s_2,\mathbf{T}_1{:}s_1$ if one of the transition sequences depicted in the bottom exists.

(i) The sequence $\mathbf{e}'_1$ corresponds to the regular simulation (commutativity) condition: starting from the same starting state q, the same end state q_f is reached after swapping the order of execution of the two actions.

In the remaining cases, the transition sequence $\mathbf{T}_1{:}s_1,\mathbf{T}_2{:}s_2$ is simulated by assertion failures.

(ii) The sequence $\mathbf{e}'_2$ corresponds to the case where executing s_2 at q leads to an assertion violation, making q a failing state.

(iii) Similarly, $\mathbf{e}'_3$ is the sequence which ends with an assertion violation after executing s_1.

With this definition of simulation, QED transformations are guaranteed to preserve (and potentially increase) assertion violations in programs.

In a typical QED proof, at the final proof state, when atomic blocks are relatively large, all assertions are discharged. This means that, in the simulation check above, cases (ii) and (iii) were in fact vacuous. Put differently, later in the proof we realize (show) that, for states q reachable from an initial state of the program, case (i) of the simulation condition always applies. Assertion annotations make this information locally available to actions s_1 and s_2 and enable its use for mover checks earlier in the proof. For a proof of soundness of this approach, see [1].

2.2 Abstraction and Reduction

In QED, reduction is combined with abstraction resulting in a powerful proof methodology as explained next. QED decides on the mover type of each action by local checks. Each action is compared with every other action of the program, assuming they are executed by different threads at any state satisfying an invariant. In order to make mover checks local and efficient (linear in the number of program actions), this approach is forced to disregard all execution-specific information. For instance, even though lock-protected ac-

```
...                                ...
 t := g; // t local, g global       havoc t;
 CAS(g,t,t+1);                      CAS(g,t,t+1);
...                                ...
```

Figure 2. Atomic increment using Compare-And-Swap (CAS).

cesses are provably both-movers, at first, QED fails to assign any mover type to such accesses. Two types of abstraction are QED's mechanisms for providing such information to mover checks: annotating statements with assertions and relaxing transitions by replacing accesses to global variables with non-deterministic thread-local reads or writes.

Fig 1 presents an example of abstraction through assertions. The code on the left is the original snippet, where the global variable g is updated with the value of the local variable x. The update action is tagged as a non-mover because it apparently conflicts with itself: the end value of g may depends on the order of threads executing this action. However, this action is protected by a lock, so conflicting accesses to g cannot be concurrent – a fact expressed by annotating this action with an assertion as in the transformed code on the right in Fig. 1. Let tid be the variable that holds the unique thread identifier of each thread. Then clearly, two consecutive updates by different threads will always end with an assertion violation, proving that the update action is a both-mover. That these assertions fail in mover check does not imply that we have added extra failing behaviors to the original program. The assertions are our way of telling QED that two updates to g are not simultaneously enabled and they are left unproved as proof obligations to be eventually discharged, once all three actions are combined into a single atomic block.

Fig. 2 presents an example of relaxing transitions by read abstraction. First the value of the global variable g is read to the local variable t and then an atomic compare-and-swap (CAS) instruction attempts to increment g. Thus, this code executed by a thread t either atomically increments g or leaves it unchanged if a CAS instruction by another thread t' is interleaved between the read of g and the CAS by t. The read of g (by t) will be tagged as a non-mover because it fails to commute to the right of a CAS action executed by a different thread (t'). But, in this interference scenario, t does not update g and the value of t is irrelevant. We express this fact by abstracting the read of g. havoc t; assigns a non-deterministic value to t.Note that all executions of the original program (that may succeed or fail in incrementing g) are subsumed by the abstracted program, and the set of possible end values for g remains the same after this abstraction. The abstracted read is now tagged as a both-mover and can be combined with the CAS action into a single atomic block.

```
Copy(fr: Obj, to: Obj){

  action SS(fr):
      atomic{ version := fr.ver; value := fr.val;}

  action ConfNWrt(fr, to):
      atomic{ if (version == fr.ver)
                  {to.val := value; to.ver := to.ver + 1;}
              }
}

Wrt(to: Obj, newVal: int){
      atomic{ to.val := newVal; to.ver := to.ver + 1;}
}
```

Figure 3. The Copy procedure consisting of two atomic actions.

3. Motivation

In this section, we present an example which is interesting because it contains a pattern typical to optimistic concurrency control, and is difficult to handle with the current set of QED proof rules. We show how prophecy variables and tressa claims provide a simple proof of atomicity.

3.1 Atomic Copy: First Proof Attempt

The purpose the Copy procedure (Fig. 3) is to copy fr.value to to.value atomically. If Copy does not succeed, it leaves y unmodified. One can imagine Copy to be the body of a loop that is iterated until the atomic copy operation succeeds. In this example, objects have version numbers (.version) that get incremented atomically when the object's value field is modified. Copy consists of two atomic actions. The first SS(fr) takes a snapshot of the object fr into the local variables version and value. The second, ConfNWrt(fr,to), confirms that the version number has not changed since the snapshot was taken and copies fr.value to to.value. If the version number has changed, it leaves to unmodified. Wrt(to,newVal) atomically writes newVal to to.value.

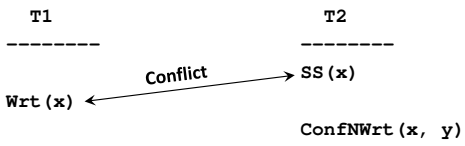


Figure 4. A thread interleaving with a conflict.

From the caller's point of view, Copy is atomic, because, when it succeeds in writing to the object to, the version number check guarantees that to has not been written to by another thread between SS(x) and ConfNWrt(x). When Copy fails, to is not modified. In a QED-style static proof, the atomicity of Copy is shown by attempting to show that either SS(fr) is a right-mover, or that ConfNWrt(fr,to) is a left-mover. In the presence of concurrent Write's by other threads, SS(fr) and ConfNWrt(fr,to) are not movers as they stand as the interleaving in Figure 4 shows. One must abstract one of these actions to make it into a mover without changing what Copy is meant to accomplish.

```
Copy(fr: Obj, to: Obj){

  action SS_Abs(fr):
      atomic{ havoc version, value; }

  action ConfNWrt(fr, to):
      atomic{ if (version == fr.ver)
                  {to.val := value; to.ver := to.ver + 1;}
              }
}

  T1                          T2
--------                    --------
Wrt(x)
                            SS_Abs(x)
                            ConfNWrt(x, y)
```

Figure 5. The initial proof attempt abstracts SS to SS_abs. This interleaving contains no conflicts, but the final value of y.val is arbitrary.

Since ConfNWrt writes to to.value, we avoid abstracting this action. We observe that, in the interleaving shown in Figure 4, the values of version and value are not used by ConfNWrt. Our

first proof attempt is therefore to abstract `SS(fr)` to `SS_Abs(fr)` (Fig. 5). The latter does not depend on `fr`, thus, does not conflict with any `Wrt` action. However, this turns out to be too much abstraction. In the interleaving in Figure 5, an arbitrary value is written to `y.val`.

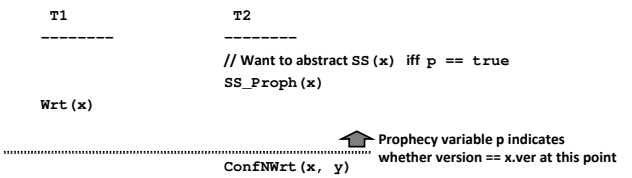


Figure 6. The prophecy variable `p`
.

3.2 Introducing Prophecy Variables to `Copy`

We would like to constrain the amount of abstraction we apply to `SS(x)`. We would like `version` and `value` to have nondeterministically-chosen values only in executions like the one in Fig. 4, in which a `Wrt` by another thread interferes with `Copy` and the atomic copy attempt fails. Introducing a prophecy variable `p` (local to the `Copy` procedure) allows us to do just that (Fig 6.)

`p` has the value `true` iff no interfering `Wrt(x)` occurs between taking a snapshot of `x` and confirming `x.ver == ver`. Put differently, `p` encapsulates how future thread interleaving nondeterminism is resolved in an execution: `p == true` iff `ConfNWrt` finds that `version == x.ver`. This is accomplished by "reverse-assignment" of the value `true` to `p` (denoted by `p =: true`) exactly when `version == x.ver`, as shown in Figure 7. `p =: true;` is shorthand for the action `atomic{ assume p == true; havoc p;}`. We refer to `p =: true` as reverse assignment because, if we imagine that we are going backwards in time along a given execution, this action has the effect of constraining earlier (between it and program start) values of `p` to `true`. If we think forward in time, initially the value of the prophecy variable is nondeterministic and guesses whether `version == x.ver` will be the case later when `ConfNWrt` executes. The execution of `ConfNWrt` is blocked if the guess expressed by the prophecy variable does not match reality.

Using `p`, we abstract the snapshot action only in desired executions, i.e., when `p` is `false`, as shown in the action `SS_Proph` in Fig. 7.

3.3 Tressa Annotations

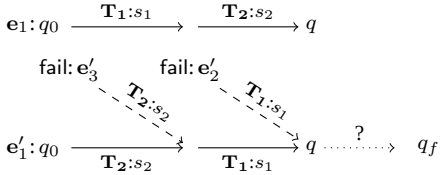
Recall that in QED proofs, assertions are used to annotate actions with information about execution history in order to remove apparent conflicts between actions (See abstraction through assertions in Section 2.) Annotating actions with assertions expressed in terms of prophecy variables is helpful in enabling further reduction in a similar way. However, assertions that refer to prophecy variables cannot be discharged by forward reasoning in time. To distinguish assertions that are discharged by backward reasoning from a final state of the program, we introduce the tressa construct. `SS_Proph` in Fig. 7 makes use of tressa statements. Similar to assertions, annotating an action with a tressa is always a valid abstraction.

In the `Copy` example in Fig. 7, the tressa claim in `SS_Proph(x)` states that if the prophecy variable is true but the value of `version` is not up to date, then this execution will eventually block and not reach a final state. With this tressa annotation, `SS_Proph(x)` becomes a right mover as illustrated in Fig. 8. This makes the entire `Copy` procedure atomic, and the tressa annotation is discharged by a simple sequential analysis.

```
Copy(fr: Obj, to: Obj){

 action SS_Proph(fr):
  atomic{  if (p) {version := fr.ver; value := fr.val;}
           else { havoc version, value;}

           tressa p ==> (version >= ver);
        }

 action ConfNWrt(fr, to):
  atomic{ if (version == fr.ver) {
              p =: true;
              to.val := value; to.ver := to.ver + 1;
          } else {
              p =: false;
          }
        }
}
```

Figure 7. The transformed `Copy` procedure makes use of a prophecy variable and a tressa annotation.

The remainder of this section provides a detailed explanation of how tressa annotations make mover checks pass.



Here, q_f represents a state in which all threads have reached the end of the code they are executing. Similarly to the mover check in the presence of assert annotations (Section 2.1), in the presence of tressa's, there are three ways a mover check to pass:

(i) Transition sequence starting at e'_1: q_f is reached starting from q_0. This is the conventional simulation definition.

(ii) e'_2: The tressa predicate of s_1 fails.

(iii) e'_3: The tressa predicate of s_2 fails.

With this definition of simulation in the presence of tressa's, QED transformations are guaranteed to preserve (and potentially increase) tressa violations in programs.

Again similarly to the case with assertions, the goal of a QED proof is to reach a proof state in which atomic blocks are large enough to discharge all tressa's. Intuitively, this means later proof steps allow us to conclude that, in this mover check, cases (ii) and (iii) were vacuous, i.e., for states q starting from which each thread can run its code to completion, it is not possible to violate the tressa predicates of s_1 and s_2. Put differently, executions starting from a state q that violates the tressa predicates will eventually get stuck. The tressa annotations allow us to use this information locally for mover checks earlier in the proof.

In the `Copy` example, when the prophecy variable `p` is false, `SS_Proph(x)` commutes to the right of `Wrt(x)` , since it is able to assign arbitrary values to `version` and `value`, as shown in the first part of Fig. 8. When `p` is `true`, `SS_Proph(x)` cannot be immediately followed by `Wrt(x)`. The tressa annotation in `p` is able to express this fact locally. If `p == true`, from any state s_2 that can be reached by executing `SS_Proph(x)` immediately followed by `Wrt(x)`, the program eventually blocks. All such s_2 violate the tressa annotation of `SS_Proph(x)`. The second part of Fig. 8 illustrates the case where the tressa on the left-hand side of the simulation check `SS_Proph(x).Wrt(x)` $\preceq$ `Wrt(x).SS_Proph(x)` fails.

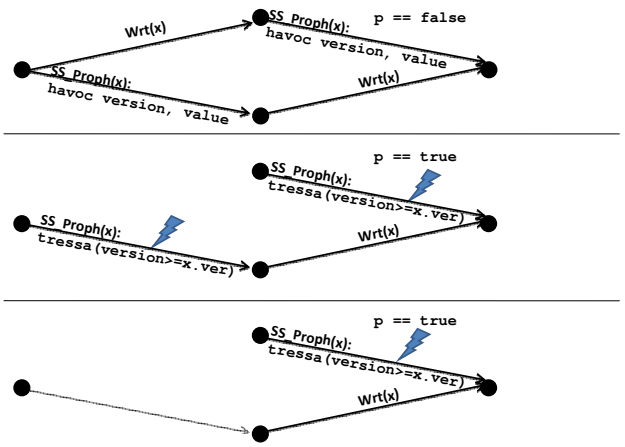


Figure 8. Why `SS_Proph(x)` commutes to the right of `Wrt(x)`, i.e., `SS_Proph(x).Wrt(x)` $\preceq$ `Wrt(x).SS_Proph(x)`. In each figure, the bottom and top parts of the diamond correspond to the left- and right-hand sides of this simulation check, respectively.

In the third part of the figure, this tressa succeeds, but the tressa on the right-hand side still fails. Thus, `SS_Proph(x)` commutes to the right of `Wrt(x)`.

4. Formalization

We start this section by formalizing the programming environment by giving the syntax and operational semantics of a simple programming language. We then build a proof system for this programming environment. The formalization given in this section closely follows that of QED as was given in [1].

4.1 Syntax

Actions: Atomic, Compound, Nullary, Full. First, we will assume that each atomic action α is in the form

$$\mathsf{assert}\ a; p; \mathsf{tressa}\ b$$

We require that a, the *assert predicate*, (b, the *tressa predicate*) be over only unprimed (primed) variables. The *transition predicate* p is over both primed and unprimed variables. For any action δ, let ϕ_δ, ψ_δ, τ_δ denote its assert, tressa and transition predicates, respectively. For instance, $\phi_\alpha = a$, $\psi_\alpha = b$ and $\tau_\alpha = p$, for α given above.

We use sequential composition ($;$) choice ($\Box$) and loop ($^{\circlearrowleft}$) operators to form *compound actions*. Formally, each atomic action is a compound action and for compound actions c_1 and c_2, $c_1;c_2$, $c_1 \Box c_2$ and $c_1^{\circlearrowleft}$ are also compound actions. We will represent each sequential code segment by a *full action*. A full action is either the *nullary action* stop which intuitively marks the end of the code, or a compound action c sequentially composed with the nullary action, $c;\mathsf{stop}$. Let $Atom$ and $Full$ denote the set of all atomic and full actions, respectively.

Note that, we have opted for the more intuitively appealing pseudo-language in the sample codes given in this paper. The meaning of each construct in the pseudo-language is either given informally or should be obvious. The language we describe here, on the other hand, is more suitable for formal treatment.

4.2 Semantics

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

$$\boxed{\hookrightarrow \subseteq Full \times (Atom \cup \{\lambda\}) \times Full}$$

$$\text{A-EVAL}\ \frac{\gamma \in Atom}{\gamma; c_1 \overset{\gamma}{\hookrightarrow} c_1} \qquad \text{C-LEFT}\ \frac{\gamma = \lambda}{c_1 \Box c_2 \overset{\gamma}{\hookrightarrow} c_1} \qquad \text{C-RIGHT}\ \frac{\gamma = \lambda}{c_1 \Box c_2 \overset{\gamma}{\hookrightarrow} c_2}$$

$$\text{L-ITER}\ \frac{\gamma = \lambda}{c_1^{\circlearrowleft} \overset{\gamma}{\hookrightarrow} c_1; c_1^{\circlearrowleft}} \qquad \text{L-SKIP}\ \frac{\gamma = \lambda}{c_1^{\circlearrowleft}; c_2 \overset{\gamma}{\hookrightarrow} c_2} \qquad \text{S-EVAL}\ \frac{c_1 \overset{\gamma}{\hookrightarrow} c_2}{c_1; c_3 \overset{\gamma}{\hookrightarrow} c_2; c_3}$$

Figure 9. Obtaining all possible subactions of a given full action via the silent transformation relation, $\hookrightarrow$.

- a *variable valuation* σ_s that maps a thread id and a variable to a value,
- a *code map* ϵ_s that associates a thread with a full action.

We require 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. The code map ϵ_s keeps track of what each thread is to execute. For instance, $\epsilon_s(t) = c$ means that at program state s, the remaining part of the program to be executed by thread t is given by c. We will give the small step semantics for the execution of full actions below. A program state s is called *final* if $\epsilon_s(t) = \mathsf{stop}$, for all t.

Predicates over program variables. For an assert predicate x (over unprimed program variables), let $x[t]$ denote the same 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 $\sigma_s(t,v)$, its value seen by thread t.

Similarly, the pair of program states (s_1, s_2) satisfies a transition predicate $p[t]$ (over unprimed and primed variables), 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 is replaced with $\sigma_{s_1}(t,v)$ and each primed variable v' is replaced with $\sigma_{s_2}(t,v)$.

Finally, for a tressa predicate y (over primed program variables) and a thread t, the program state s satisfies $y[t]$, denoted as $s' \vDash y[t]$ or as $y[t](s')$, if $y[t]$ evaluates to true when each primed variable v' is replaced with $\sigma_s(t,v)$.

Configurations. The evaluation of a full action is given in terms of the *silent transformation relation*, $\hookrightarrow$, whose definition is given in Fig. 9. Intuitively, if we imagine the execution of a full action represented as a flowchart with an explicit control pointer denoting what to execute next, the silent transformation relation corresponds to advancing the control pointer over the flowchart not modifying any program variable's value. When this imaginary control pointer selects a branch, it is represented by the label λ which is called the *invisible transition*. Otherwise, the label is the content of the box over which the control pointer passes.

For full actions c and d, and a string $\overline{\gamma} = \gamma_1 \ldots \gamma_n$ over $Atom \cup \{\lambda\}$, we let $c \overset{\overline{\gamma}}{\hookrightarrow} d$ denote a sequence of silent transformations

$$c = c_0 \overset{\gamma_1}{\hookrightarrow} c_1 \ldots \overset{\gamma_n}{\hookrightarrow} c_n = d$$

A program state s' is in $\mathsf{conf}(s)$, the *configurations* reachable from program state s, if, for all t, there exists some string $\overline{\gamma}_t$ such that $\epsilon_s(t) \overset{\overline{\gamma}_t}{\hookrightarrow} \epsilon_{s'}(t)$. Intuitively, s' is a configuration of s if s' can be obtained by moving forward the control pointer of each thread's program an arbitrary number of, possibly 0, steps.

Let s and s' be program states, t be a thread id. Then, s' is called a (t,α)-*successor* of s (or s, a (t,α)-*predecessor of* s', if the following conditions hold:

- $\epsilon_s(t) \xrightarrow{\lambda^k \alpha} \epsilon_{s'}(t)$, for some $k \geq 0$.
- for all $u \neq t$, $\epsilon_{s'}(u) = \epsilon_s(u)$,

Intuitively, s' is a (t, α)-successor of s if at s thread t has α as a possible next action and s' is the same as s except the control flow at t skips over α. For any thread t and $\gamma \in Atoms$, (t, γ) is called a *transition label*.

Execution semantics. Alluding to the flowchart and control pointer analogy given above, the execution of a program can be seen as advancing the control pointer of each thread while making the effect of each atomic action passed over visible to variable valuations. Let α be an atomic action. We write $s \xrightarrow{(t,\alpha)} s'$ if

- s' is a (t, α)-successor of s,
- for all $u \neq t$ and for any local variable x, $\sigma_s(u, x) = \sigma_{s'}(u, x)$,
- for any variable g and thread u, $\sigma_{s'}(t, g) = \sigma_{s'}(u, g)$,
- $(s, s') \vDash \tau_\alpha[t]$.

In other words, $s \xrightarrow{(t,\alpha)} s'$ holds when t can execute α next, 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. Note that both assert and tressa commands behave like no-op's.

A *trace* is a sequence of transition labels, $\mathbf{l} = l_1 \dots l_k$. The trace moves a state s_0 to s_k, written $s_0 \xrightarrow{\mathbf{l}} s_k$, if there is a sequence of states $\langle s_i \rangle_{0<i\leq k}$, a *run* of P over $\mathbf{l}$, such that for all $0 < i \leq k$, $s_{i-1} \xrightarrow{l_i} s_i$.

The run is *maximal* if s_k cannot make any transition. The run is *exhaustive* if s_k is final (it is maximal and $\epsilon_{s_k}(t) = \mathsf{stop}$, for all threads t). Henceforth, we will always consider maximal runs.

4.3 Proof and Correctness

Proof state. A *proof state* is the tuple $(\mathcal{P}, \mathcal{I})$, where $\mathcal{P}$ and $\mathcal{I}$ are called the program and the forward invariant, respectively. The program $\mathcal{P} \subset Full$ is a set of *procedures*. The forward invariant $\mathcal{I}$ is a predicate over unprimed global variables appearing in the program. It is a predicate that has to be preserved by each atomic action in $\mathcal{P}$. An atomic action α preserves the forward invariant $\mathcal{I}$, written $\mathcal{I} \rightleftarrows \alpha$, if $s_1 \xrightarrow{(t,\alpha)} s_2$ and $s_1 \vDash \mathcal{I}$ imply $s_2 \vDash \mathcal{I}$. In other words, $\mathcal{I}$ is preserved by α if $\mathcal{I}$ cannot be falsified (changed from true to false) by any execution of α. If all the atomic actions of program $\mathcal{P}$ preserve the invariant $\mathcal{I}$, $\mathcal{P}$ is said to preserve $\mathcal{I}$, written $\mathcal{I} \rightleftarrows \mathcal{P}$.

A program state s is called an initial program state of $(\mathcal{P}, \mathcal{I})$ if $s \vDash \mathcal{I}$,[1] there are only finitely many t such that $\epsilon_s(t) \neq \mathsf{stop}$ and for each such t, $\epsilon_s(t)$ is in $\mathcal{P}$. We will let Tid be the (finite) set $\{t \mid \epsilon_s(t) \neq \mathsf{stop}\}$.

For a non-initial (resp. non-final) program state s (resp. r), define $\mathsf{lst}(s)$ (resp. $\mathsf{fst}(r)$) as the set of all transition labels $l = (t, \alpha)$ such that there exists some program state s' (resp. r') with $s' \xrightarrow{l} s$ (resp. $r \xrightarrow{l} r'$). That is, $(t, \alpha) \in \mathsf{lst}(s)$ means that the last action that thread t performed prior to reaching s is α. Similarly, $(t, \alpha) \in \mathsf{fst}(s)$ means that α can be the first action executed by thread t at state s. Note that, either set contain more than one label for the same thread due to possible branching.

Forward and backward violations. With the introduction of tressa predicates, correctness not only implies the impossibility of reaching an assert violation from an initial program state, but also the impossibility of reaching a final state starting from a state violating a tressa predicate. The former kind of violation is named a *forward violation*, whereas the latter is called a *backward violation*. For the formal definitions to follow, fix a proof state $(\mathcal{P}, \mathcal{I})$.

DEFINITION 1 (forward violation). *A run $\langle s_r \rangle_{0\leq r\leq n}$ of $\mathcal{P}$ is called a* forward violation *(f-violation) if the following conditions hold:*

- *s_0 is an initial state of $(\mathcal{P}, \mathcal{I})$,*
- *$\neg\phi_\beta[u](s_n)$ evaluates to true for some $(u, \beta) \in \mathsf{fst}(s_n)$.*

Intuitively, a forward violation is a run of $\mathcal{P}$ that starts from an initial program state s_0 and reaches a program state s_n which violates the assert predicate, ϕ_β, of an action β which thread u can execute at state s_n. It is important to note that the transition predicate of β, τ_β, does not need to be satisfied at s_n; if its assert predicate is violated, the outgoing transition (from s_n) is ignored in f-violation.

DEFINITION 2 (backward violation). *The run $\langle s_r \rangle_{0\leq r\leq n}$ of $\mathcal{P}$ is called a* backward violation *(b-violation) if the following conditions hold:*

- *$s_0 \in \mathsf{conf}(s)$ for some initial state s of $(\mathcal{P}, \mathcal{I})$,*
- *s_n is a final state of $\mathcal{P}$,*
- *$\mathcal{I}(s_0) \wedge \neg\psi_\alpha[t](s_0')$ evaluates to true for some $(t, \alpha) \in \mathsf{lst}(s_0)$,*

Intuitively, a backward violation is a run of $\mathcal{P}$ that ends at a final state s_n, starts at a configuration s_0 of an initial program state s such that there is a thread t which could have executed α prior to reaching s_0 and the tressa predicate of α, ψ_α, is violated by s_0. Again, as in forward violation, we do not require that there exist a state s' such that $(s', s_0) \vDash \tau_\alpha[t]$; if the tressa predicate is violated, the incoming transition (into s_0) is ignored.

Note that, a forward violation does not have to lead to a final state, much like a backward violation does not have to start from an initial state.

DEFINITION 3 (Violation-free). *A proof state $(\mathcal{P}, \mathcal{I})$ is* violation free *(vf) if it does not allow a run that is either a forward or a backward violation; it is called* non-violation free, *(non-vf), otherwise.*

4.4 Simulation and Composition

In this section, we will define the simulation relation between two atomic actions and prove that simulation preserves violations of the program. We will also define the composition of two atomic actions which will be used in a proof rule making use of mover types explained in the following section.

DEFINITION 4 (Simulation). *Let $\alpha = \mathsf{assert}\ a; p; \mathsf{tressa}\ b$, $\beta = \mathsf{assert}\ c; q; \mathsf{tressa}\ d$, t be an arbitrary thread id and $\rho = (\mathcal{P}, \mathcal{I})$ be a proof state. We say β simulates α at ρ, written $\alpha \preceq_\rho \beta$, if the following three conditions hold:*

S1 $c \Rightarrow a$,
S2 $d \Rightarrow b$,
S3f $p \Rightarrow q \vee \neg c$,
S3b $p \Rightarrow q \vee \neg d$

Whenever clear from the context the proof state subscript will be dropped.

The simulation conditions are relaxed in certain cases. Intuitively, S3f, along with S1, is used to preserve forward violations: If there was a forward violation with α, there has to be a forward violation with β substituted in place of α. If each assert predicate is true, there can be no forward violation, thus condition S3f becomes unnecessary (S1 becomes trivially satisfied with a being

[1] The thread id is ignored for invariants, since all threads agree on the value of all global variables.

identical to true). Thus, if ρ is such that the assert predicate of each $\alpha \in Atoms(\mathcal{P})$ is identical to true, the condition S3f is not required to hold. In other words, if the program contained only tressa annotations, then we require only S1, S2 and S3b to hold. A similar argument holds for backward violations, tressa predicates and S3b: If ρ is such that the assert predicate of each $\alpha \in Atoms(\mathcal{P})$ is identical to true, the condition S3f is not required to hold. That is, if the program contained only assert annotations, then we require only S1, S2 and S3f to hold.

We will sometimes restrict a simulation relation to a set of program state pairs represented by a logical formula. Formally, $\alpha \preceq^{\Theta} \beta$ if the simulation conditions hold for all state pairs that satisfy Θ. For instance, if there is no s_2 such that $(s_1, s_2) \vDash \Theta$, then the first simulation condition S1, $c(s_1) \Rightarrow a(s_1)$, does not need to hold for $\alpha \preceq^{\Theta} \beta$ whereas $\alpha \preceq \beta$ would fail if S1 failed for s_1. Let $Atoms(\mathcal{P})$ be all atomic actions α of $\mathcal{P}$, that is, $s \xrightarrow{(t,\alpha)} s'$ holds for some $s \in \mathsf{conf}(s_i)$, where s_i is an initial state of $\mathcal{P}$.

LEMMA 1. *Let $\rho = (\mathcal{P}, \mathcal{I})$ be a non-violation free proof state. Let α be an atomic action in $Atoms(\mathcal{P})$. Let β be another atomic action such that $\alpha \preceq_\rho \beta$ holds. Then, $(\mathcal{P}', \mathcal{I})$ is a non-violation free proof state, where $\mathcal{P}'$ is obtained by replacing α with β in $\mathcal{P}$.*

PROOF 1. *Take any violation of ρ in which α occurs. That replacing all occurrences of α with β in the violation will lead to the construction of another violation in $(\mathcal{P}', \mathcal{I})$ follows directly from the definition of simulation.*

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,

$$\mathsf{wp}(p, x) = \{s \mid \forall s'.\, p(s, s') \Rightarrow x(s')\}$$

Similarly, $\mathsf{sp}(x, p)$ denotes the *strongest post-condition* of predicate x for transition predicate p, and stands for all next states that cannot be reached after executing p from states violating x. Formally,

$$\mathsf{sp}(x, p) = \{s' \mid \forall s.\, p(s, s') \Rightarrow x(s)\}$$

Finally, for two transition predicates p and q, define their composition $p \cdot q$, as the transition predicate

$$p \cdot q = \{(s_1, s_2) \mid \exists s_3.\, p(s_1, s_3) \wedge q(s_3, s_2)\}$$

As actions are reduced, we need a formal mechanism to define the resulting atomic action by specifying what its assert, tressa predicates and transition predicate are. The following definition provides this mechanism in terms of wp, sp and $\cdot$.

DEFINITION 5. *Let α and β be two atomic actions. Define their* composition, *$\alpha \circ \beta$, as the atomic action*

$$\mathsf{assert}\ \phi_\alpha \wedge \mathsf{wp}(\tau_\alpha, \phi_\beta); \tau_\alpha \cdot \tau_\beta; \mathsf{tressa}\ \psi_\beta \wedge \mathsf{sp}(\psi_\alpha, \tau_\beta)$$

5. Proof Rules

In this section, we will define the new rules enabling the use of tressa and assert claims . In order to make the paper self-sufficient, Figure 10 lists the proof rules of [1] relevant to the subset we are using in this paper. The rule ANNOT-H is for annotating atomic actions with a new (history) variable. The rule INV is for strengthening of the invariant. The rule SIM is for abstracting an action by replacing it with one that simulates it. The rules RED-L, RED-S, RED-C are for reducing loops, sequential composition and conditional branches of two atomic statements, respectively.

5.1 Prophecy Variable Introduction

The main concern when adding a new variable into the program is to annotate statements so that no terminating execution of the original program is left out. That is why the ANNOT-H rule for introducing history variables into the program requires a transition for every valuation of the auxiliary variable: if the original program makes a transition over a certain valuation of variables, so will the new program over the same valuation for any value of the history variable. Prophecy variables should satisfy a similar requirement. The condition that has to be satisfied for prophecy variables, however, is the dual of that of a history variable. Prophecy variable introduction requires the new transition be defined for all next state values of the prophecy variable. The formal condition for prophecy variable introduction is given by the following ANNOT-P rule.

$$\boxed{\mathcal{P}_1, \mathcal{I}_1 \dashrightarrow \mathcal{P}_2, \mathcal{I}_2}$$

ANNOT-H
$$\frac{\begin{array}{c} a \notin Var \quad 1 \le i \le n \quad Atoms(\mathcal{P}) = \{\alpha_1^i\} \\ \phi_{\alpha_1}^i = \phi_{\alpha_2}^i \quad \psi_{\alpha_1}^i = \psi_{\alpha_2}^i \quad \mathcal{I} \rightleftarrows \alpha_2^i \\ \vDash \tau_{\alpha_1}^i \Rightarrow \forall a. \exists a'.\, \tau_{\alpha_2}^i \end{array}}{\mathcal{P}, \mathcal{I} \dashrightarrow \mathcal{P}[Var \mapsto Var \cup \{a\}, \alpha_1^i \mapsto \alpha_2^i], \mathcal{I}}$$

INV
$$\frac{\mathcal{I}_2 \Rightarrow \mathcal{I}_1 \quad \mathcal{I}_2 \rightleftarrows \mathcal{P}}{\mathcal{P}, \mathcal{I}_1 \dashrightarrow \mathcal{P}, \mathcal{I}_2}$$

SIM
$$\frac{\alpha \preceq_{(\mathcal{P},\mathcal{I})} \beta}{\mathcal{P}, \mathcal{I} \dashrightarrow \mathcal{P}[\alpha \mapsto \beta], \mathcal{I}}$$

RED-C
$$\frac{\gamma = \mathsf{assert}\ \phi_\alpha \wedge \phi_\beta; \tau_\alpha \vee \tau_\beta; \mathsf{tressa}\ \psi_\alpha \wedge \psi_\beta}{\mathcal{P}, \mathcal{I} \dashrightarrow \mathcal{P}[\alpha \square \beta \mapsto \gamma], \mathcal{I}}$$

RED-S
$$\frac{\mathcal{P}, \mathcal{I} \vdash \alpha_1 : \mathbb{R} \quad \text{or} \quad \mathcal{P}, \mathcal{I} \vdash \alpha_2 : \mathbb{L}}{\mathcal{P}, \mathcal{I} \dashrightarrow \mathcal{P}[\alpha_1; \alpha_2 \mapsto \alpha_1 \circ \alpha_2], \mathcal{I}}$$

RED-L
$$\frac{\begin{array}{c} \mathcal{P}, \mathcal{I} \vdash \alpha : m \qquad m \in \{\mathbb{R}, \mathbb{L}\} \qquad \mathcal{I} \rightleftarrows \beta \\ \vDash \phi_\beta \Rightarrow \tau_\beta[Var/Var'] \qquad \mathcal{I} \vdash \beta \circ \alpha \preceq \beta \end{array}}{\mathcal{P}, \mathcal{I} \dashrightarrow \mathcal{P}[\alpha^{\circlearrowleft} \mapsto \beta], \mathcal{I}}$$

Figure 10. The proof rules of the QED method.

ANNOT-P
$$\frac{\begin{array}{c} a \notin Var \quad 1 \le i \le n \quad Atoms(\mathcal{P}) = \{\alpha_1^i\} \\ \phi_{\alpha_1}^i = \phi_{\alpha_2}^i \quad \psi_{\alpha_1}^i = \psi_{\alpha_2}^i \quad \mathcal{I} \rightleftarrows \alpha_2^i \\ \vDash \tau_{\alpha_1}^i \Rightarrow \forall a'. \exists a.\, \tau_{\alpha_2}^i \end{array}}{\mathcal{P}, \mathcal{I} \dashrightarrow \mathcal{P}[Var \mapsto Var \cup \{a\}, \alpha_1^i \mapsto \alpha_2^i], \mathcal{I}}$$

LEMMA 2. *Let $\rho_1 = (\mathcal{P}_1, \mathcal{I}_1)$ be a proof state. Let ρ_2 be the proof state obtained from ρ_1 by an application of the* ANNOT-P *rule. Let $\langle s_i \rangle_{1 \le i \le n}$ be a run of $\mathcal{P}_1$. Then, there exists a run $\langle s_i' \rangle_{1 \le i \le n}$ of $\mathcal{P}_2$ such that for all i, s_i and s_i' have the same code maps and variable valuations except for the prophecy variable a introduced by the* ANNOT-P *rule.*

PROOF 2 (Sketch). *By induction on the length of the run, n. Construct the run backwards, starting from the end state and make the observation that for each state, due to the premise of the* ANNOT-P *rule, there always exists a value of the prophecy variable in the preceding state such that the transition of $\mathcal{P}_1$ is enabled in $\mathcal{P}_2$.*

5.2 Mover Checks

QED depends on reduction and reduction is the act of merging atomic actions of suitable mover types, as can be seen from the rules RED-S and RED-L. In our previous work [1], we defined mover types with only forward violations in mind. Below, we re-define mover types to account for both forward and backward violations. We also establish the correctness of the definitions via soundness results.

Let $\mathsf{pre}(tp, x)$, the *pre-image* of predicate x for the transition predicate tp, denote the predicate only satisfied by all the states in the set $\{s \mid \exists s'.\, tp(s, s') \wedge x(s')\}$. Intuitively, $\mathsf{pre}(tp, x)$ gives all states s such that executing tp at s can reach a state s' which satisfies x. Similarly, $\mathsf{post}(x, tp)$, the *post-image* of x for tp, denotes the predicate only satisfied by all the states in the set $\{s' \mid \exists s.\, tp(s, s') \wedge x(s)\}$. Intuitively, $\mathsf{post}(x, tp)$ gives all states s' that can be reached by executing tp from some state s satisfying x. A label (u, β) *follows* another label (t, α) in program $\mathcal{P}$, if there exists a program state s such that $(t, \alpha) \in \mathsf{lst}(s)$ and $(u, \beta) \in \mathsf{fst}(s)$.

DEFINITION 6 (Right-mover). *Let $\rho = (\mathcal{P}, \mathcal{I})$ be a proof state and α be an atomic action in $Atoms(\mathcal{P})$. The action α is a right-mover if for any β in $Atoms(\mathcal{P})$, threads t, u with $t \neq u$, the following conditions hold:*

1. $\alpha[t] \circ \beta[u] \preceq^{\Theta} \beta[u] \circ \alpha[t]$, *with* $\Theta = \tau_\alpha[t] \cdot \tau_\beta[u] \Rightarrow \phi_\alpha[t] \vee \psi_\alpha[t]$,
2. $\mathsf{post}(\phi_\beta[u], \tau_\alpha[t]) \Rightarrow \phi_\beta[u]$

The first condition requires that (t, α) followed by (u, β), for arbitrary $t \neq u$, is simulated by (u, β) followed by (t, α) except possibly for state pairs (s_1, s_2) such that s_1 violates $\phi_\alpha[t]$, s_2 violates $\psi_\alpha[t]$ and s_2 is the program state reached from s_1 by executing (t, α) followed by (u, β). The second condition states that α cannot change the assert predicate of any other action from false to true.

A left-mover can be defined similar to right-mover using dual conditions.

DEFINITION 7 (Left-mover). *Let $\rho = (\mathcal{P}, \mathcal{I})$ be a proof state and α be an atomic action in $Atoms(\mathcal{P})$. The action α is a left-mover if for any β in $Atoms(\mathcal{P})$, threads t, u with $t \neq u$, the following conditions hold:*

1. $\beta[u] \circ \alpha[t] \preceq^{\Theta} \alpha[t] \circ \beta[u]$, *with* $\Theta = \tau_\alpha[t] \cdot \tau_\beta[u] \Rightarrow \phi_\alpha[t] \vee \psi_\alpha[t]$,
2. $\mathsf{pre}(\psi_\beta[u], \tau_\alpha[t]) \Rightarrow \psi_\beta[u]$

Let $\mathcal{P}, \mathcal{I} \vdash \alpha : \mathbb{R}$ denote that α is a right-mover at proof state $(\mathcal{P}, \mathcal{I})$. Similarly, $\mathcal{P}, \mathcal{I} \vdash \alpha : \mathbb{L}$ denotes that α is a left-mover. Besides the change in the mover definitions, the sequential reduction rule RED-S given in Fig. 10 remains the same in the presence of prophecy variable and tressa annotations.

We close this section by stating the soundness results. The lemma below establishes that reduction based on the above mover definitions cannot change a non-vf proof state into a vf proof state.

LEMMA 3 (Soundness of Reduction). *Let $\rho_1 = (\mathcal{P}, \mathcal{I})$ be a proof state. Let ρ_2 be the proof state obtained from ρ_1 by an application of the* RED-S *rule. If ρ_2 is violation free, then ρ_1 is also violation free.*

PROOF 3 (Sketch). *By contradiction. Without loss of generality, assume α_1 of the* RED-S *rule to be a right-mover. Assume ρ_1 to be non-violation free and ρ_2 to be violation free. Then, there must exist a violation in ρ_1 in which for some t, (t, α_1) are not (t, α_2) not consecutive in the violation. Starting from this execution, move each such (t, α_1) to the right until it either immediately precedes its matching (t, α_2) or (t, α_1) along with its matching (t, α_2) is removed from the execution. This moving around is feasible due to the definitions of the simulation relation and right-mover. The final execution, where each occurrence of (t, α_1) is immediately followed by (t, α_2) is a violation in ρ_2, establishing the contradiction.*

Finally, the theorem below establishes the soundness of the QED method. We define a *proof* as a sequence of proof states each of which is obtained from its immediate predecessor by an application of the proof rules defined in this section.

```
procedure ReadPair(a: int, b: int)
returns (s: bool, da: Obj, db: Obj)
{
  var va: int, vb: int;

1:  atomic { va := m[a].v; da := m[a].d; }
2:  atomic { vb := m[b].v; db := m[b].d; }
3:  s := true;
4:  atomic { if (va < m[a].v) { s:= false; } }
5:  atomic { if (vb < m[b].v) { s:= false; } }
6:  if (!s) { da := nil; db := nil; }
}

procedure Write(a: int, d: Obj)
{
  atomic { m[a].d := d; m[a].v := m[a].v+1; }
}
```

Figure 11. A collection that implements an atomic read of two distinct variables, `ReadPair`, and random access updates, `Write`.

THEOREM 1 (Soundness). *Let $(\mathcal{P}_0, \mathcal{I}_0) \dashrightarrow \ldots \dashrightarrow (\mathcal{P}_n, \mathcal{I}_n)$ be a proof. If the proof state $(\mathcal{P}_n, \mathcal{I}_n)$ is violation free, then so is $(\mathcal{P}_0, \mathcal{I}_n)$.*

6. Examples

In this section, we verify two examples, both making use of optimistic concurrency. The first is an implementation of an atomic snapshot of a pair of objects in the presence of concurrent updates to the objects. The second is an implementation of a set with methods for searching and inserting elements. In both of the examples, a finite number of threads share the global and execute one of the methods.

6.1 Pair Snapshot

Consider the code in Fig. 11. The `ReadPair` procedure is supposed to implement an atomic read of two addresses in the presence of concurrent updates done by the `Write` procedure. `ReadPair` succeeds and returns the read values along with a status flag denoting success, if it observes a consistent state of the memory for two addresses. Otherwise, it fails and sets its status flag to false denoting failure, along with setting the read values to default values (`nil`). Each call of the `Write` procedure updates the data value stored in an address and increments the version number for that address by one. We would like to prove that the `ReadPair(a,b)` method, when it returns true, behaves like an instantaneous read of the two addresses.

Intuition for Atomicity. There are two possible execution scenarios for `ReadPair(a,b)`. Imagine that thread t is executing `ReadPair(a,b)` and has executed line 1, the first read of `a` (henceforth *initial read*). Until the second read of line 4, which we will call the *confirming read*, if some other thread executes `Write(a,d)`, then `ReadPair` will observe two distinct states of `a` and hence will return false, representing this inconsistency. Mutatis mutandis for `b`, lines 2, 5 and `Write(b,d)`. We will call such executions as *inconsistent*. In other words, an inconsistent run of `ReadPair` returns `(false,nil,nil)`. An execution where interfering updates do not occur between the initial and confirming reads of either address will be called *consistent*.

Now each read action conflicts with an update to the same address. As such, neither of the read actions of `ReadPair` are movers in their current state. Observe that if `ReadPair` is to have an inconsistent execution, since the read values do not matter, their values can be abstracted away. Abstracting the read values will make all

```
procedure ReadPair(a: int, b: int)
returns (s: bool, da: Obj, db: Obj)
{
p1:  atomic { if (p[a]) { va := m[a].v; da := m[a].d; }
              else { havoc va, da; }
            }
p2:  atomic { if (p[b]) { vb := m[b].v; db := m[b].d; }
              else{ havoc vb, db; }
            }
p3:  s := true;
p4:  atomic { if (va < m[a].v) {
                s:= false; p[a] =: false; }
              else { havoc s, p[a]; } }
p5:  atomic { if (vb < m[b].v) {
                s:= false; p[b] =: false; }
              else { havoc p[b]; if(s) { havoc s; } } }
p6: if (!s) { da := nil; db := nil; }
}
```

Figure 12. Prophecy variable introduction, one per object.

```
procedure ReadPair(a: int, b: int)
returns (s: bool, da: Obj, db: Obj)
{
f1:  atomic { if (p[a]) { va := m[a].v; da := m[a].d; }
              else { havoc va, da; }
              tressa p[a] ==> va>=m[a].v; }
f2:  atomic { if (p[b]) { vb := m[b].v; db := m[b].d; }
              else{ havoc vb, db; }
              tressa p[b] ==> vb>=m[b].v; }
f3:  s := true;
f4:  atomic { if (va < m[a].v) {
                s:= false; p[a] =: false; }
              else { havoc s, p[a]; } }
f5:  atomic { if (vb < m[b].v) {
                s:= false; p[b] =: false; }
              else { havoc p[b]; if(s) { havoc s; } } }
f6: if (!s) { da := nil; db := nil; }
}
```

Figure 13. Complete annotation with tressa claims included.

the read actions both movers. However, we have to also take care of the consistent execution of `ReadPair`. In a consistent execution, a write to `a` cannot occur between the corresponding initial and confirming reads. Put differently, in a consistent execution, the initial read is a right-mover, the confirming read is a left-mover, because no update to the read address can occur between them. Note that, in a consistent execution, the read values do matter as they should be returned when `ReadPair` terminates successfully so abstracting away the read values in this case is not possible.

In the proof we will construct, initial reads will either read the exact value (consistent execution) or abstract away the reads (inconsistent execution). The decision will be made according to a prophecy variable per address whose value will be set according to the presence of a conflicting update before confirming read is done. The reverse assignment to the prophecy variables will be made in the confirming reads.

Prophecy variables. The code with prophecy variables introduced is given in Fig. 12. As we have hinted above, the prophecy variable, mapping each address to a boolean value, is reverse assigned in lines `p4` and `p5`. For an inconsistent execution, the prophecy variable is set to false. The initial reads are updated to make use of the prophecy variable values. Intuitively, `p[a]` equal to true means that the current execution will not see an interfering update until the confirming read of `a` is done. That is why when `p[a]` is true, the exact value of `q[a].d` is read. Similarly, when `p[a]` is false, the read values are abstracted away since the prophecy variable foresees interference.

```
procedure ReadPair(a: int, b: int)
returns (s: bool, da: Obj, db: Obj)
{
  atomic {
    if (p[a]) { va := m[a].v; da := m[a].d; }
    else { havoc va, da; }
    if (p[b]) { vb := m[b].v; db := m[b].d; }
    else { havoc vb, db; }
    s := true;
    if (va < m[a].v) {
      s:= false; p[a] =: false; }
    else { havoc s, p[a]; } }
    if (vb < m[b].v) {
      s:= false; p[b] =: false; }
    else { havoc p[b]; if(s) { havoc s; } }
    if (!s) { da := nil; db := nil; }
}
```

Figure 14. `ReadPair` reduced to a single atomic action.

The abstracted confirming reads, lines `p4`, `p5`, are left-mover. For instance, `p4`, coming immediately after a conflicting update is simulated by executing `p4` followed by the same update. However, the initial reads are still non-mover.

Consider the code given in Fig. 13. We have added tressa claims to the initial reads reflecting our intuition about the value of the prophecy variables. Since `p[a]` equal to true foresees no interference, we claim that any execution that violates the tressa predicate cannot terminate. Imagine the contrary: `p[a]` is true and `va>=m[a].v` is false. Observe that `m[a].v` is never decremented and `va` remains the same from `f1` onwards. When line `f4` is reached, the condition `va<m[a].v` will be true. The then branch of the if statement will be taken and the current value of `p[a]`, true, will not match the reverse assigned value, false, which will block the execution. It is important to note the role of additional blocking behavior which we deliberately inserted via the prophecy variable. It is also important to note that all this execution based reasoning is implied in the tressa claim whose main use comes in representing this kind of information in locally performing mover checks.

The rest of the proof is trivial as it consists of reducing the whole method into a single action and discharging the tressa claims using sequential analysis (or applying the definition of composition of actions). The final code is given in Fig. 14.

6.2 Lookup and Insert

Fig. 15 presents the `Lookup` and `Insert` methods for a bounded set of non-negative integers. Set elements are stored in an array in which duplicates are allowed. An array slot is taken to be empty if it contains -1. Initially, all slots are assumed to be empty. The contents of the set are given by the set of values in non-empty slots. Reads and writes to the array are protected by a separate lock per array index. For simplicity, in the figures we do not refer to this lock. Instead, we indicate what accesses are guaranteed to be atomic by use of this lock.

The `Insert` method starts from an arbitrary array index in order to reduce conflicts between concurrent executions of `Insert` on early array indices. It examines array slots in increasing order of indices and wraps around at the end of the array. `Insert` succeeds when it either finds an empty slot to which it atomically writes the new element, or it finds an occupied slot containing the element it was trying to insert. The method fails if all array slots are tried exactly once and each try finds a non-empty slot containing a different element. In this simplified implementation, there is no removal. `Lookup(x)` starts from the first array slot and searches in increasing order of indices for `x`. It returns true iff for some array index `i`, `q[i] == x`. Since `Insert` can start from an arbitrary index, `Lookup` must examine the entire array before deciding whether or not x is in the set.

```
procedure Lookup(x: data)
returns result: bool;
{
  f := false; i := 0;

  while (i<n && !f) { f := (q[i] == x); i := i+1; }
  result = f;
}

procedure Insert(x: data)
returns done: bool;
{
  havoc i; assume i<n;
  cnt := 0; f := false;

  while (cnt<n && !f) {
    if (*) {
      atomic { assume q[i]==-1; q[i] := x; f := true; } }
    else {
      if (*) { atomic { assume q[i]==x; f := true; } }
      else {
        atomic {
          assume q[i]!=x && q[i]!=-1;
          i := (i+1) mod n; cnt := cnt+1; }
      }
    }
  }
  done := f;
}
```

Figure 15. A bounded set with two methods for searching for an element, `Lookup`, and adding an element, `Insert`.

We would like to prove that the `Lookup` method can be summarized as an atomic block that returns true iff for some array index `i`, `q[i] == x`.

Intuition for Atomicity. Observe that all actions except the read of `q[i]` are thread-local, i.e., they are both movers. Then the only potential conflict which needs to considered is between the read of `q[i]` and the update to `q[i]` done by the `Insert` method when `q[i] == -1`.

Call an iteration of the `Lookup` loop for some `i` *failing* if `q[i] != x` (denoted by $F(i)$) and *succeeding* (denoted by $S(i)$) otherwise. Executions of `Lookup` that return false are of the following form

$$..., F(0), ..., F(1), ..., F(2), ..., F(n-1), ..., F(n), ...$$

while executions that return true are of the following form

$$..., F(0), ..., F(1), ..., F(2), ..., F(i-1), ..., S(i), ...$$

where ... represents a sequence of actions by other threads. The reduction-based proof is based on the following intuition. The commit action for `Lookup(x)`'s that return false is $F(0)$ because the set may contain x later in the execution. For `Lookup`'s that return true, the commit action is $S(i)$, since the action that writes the first `x` to an array slot may immediately precede $S(i)$.

In order to reduce the entire execution of the loop to an atomic action, for `Lookup's` that return false, we need all $F(k)$ to be left-movers in order to group them next to $F(0)$, while, for `Lookup`'s that return true, all $F(k)$'s must be right movers in order to move immediately to the left of $S(i)$. The two kinds of lookups seem to require different applications of reduction to prove atomicity.

To remedy this difficulty, we duplicate the loop. One copy represents the case where `Lookup` fails to find the element and returns false, and the other represents the case where `Lookup` finds the element and returns true. This split allows us to apply reduction differently in the two different cases.

After the split, `Lookup`'s that return false are handled easily. $F(k)$, which requires that `q[k] != -1`, commutes to the left of any other action. This is because once `q[k] != -1`, it never becomes -1 again, thus, all actions to the left of $F(k)$ must have left `q[k] == -1` unmodified.

```
procedure Lookup(x: data)
returns result: bool;
{
  f := false; i := 0;

  if (*) {
    while (i<n && !f) {
      atomic { f := (q[i] == x); }
      i := i+1;
      }
    assume !f;
  } else {
    while (i<n && !f) {
      atomic { f := (q[i] == x); }
      i := i+1;
      assume (!f && i<n);
    }
    f := q[i]==x;
    assume (f || i>=n);
    assume f;
  }
  result := f;
}
```

Figure 16. The `Lookup` method after some code transformations. The main loop is duplicated with the then branch representing the unsuccessful search, the else branch representing the failing iterations followed by the succeeding iteration.

For `Lookup`'s that return true, further abstraction is needed. It is clear that $F(k)$ does not commute to the right of an action α that writes `x` to `q[k]`. Thus, for `Lookup`'s that return true, we need to abstract the loop body so that $F(k)$ becomes a right mover. We accomplish this by allowing the loop body to set `f` to false even when `q[k] == x`. We perform this abstraction for all loop iterations except for the last one.

This contrived example mimics lookups in more realistic concurrent data structures. In these examples as well, the commit points and mover types depend on the method's return value which is only known in the future. In this example, we make only implicit use of prophecy variables. Most importantly, the return value of the method (i.e., the value of `f` at the end of the loop) acts as a prophecy variable. The two copies of the code after the split correspond to the two different values of the prophecy variable.

Code transformation. The code after the transformation explained above is given in Fig. 16. The main loop is duplicated and a non-deterministic choice, represented by `if(*)` and corresponding to whether `Lookup` returns true or false, picks either branch. The statements `assume f` and `assume !f` (both left-movers, since they refer to the local variable `f`) are appended to the two copies to mark them as such. This transformation preserves all executions of the original `Lookup`. In the else branch, the final iteration of the loop is peeled out in order to carry out the reduction proof outlined earlier.

Abstraction, prophecy variables and tressa claims. The annotated code is given in Fig. 17. Let us first analyze the abstraction done in the failing branch: appending `tressa !f` to the read of `q[i]`. This tressa annotation claims that this action can lead to program termination only when it is executed at a state where `q[i]` is not equal to `x`. Note that, this necessary condition for termination of the failing branch is due to the very end `assume !f`. Executions which violate this tressa annotation have "chosen the wrong branch", i.e., in order for these executions to terminate, control should have gone down the other non-deterministic branch.

Recall that we were trying to show that failing iterations were left-movers. The problematic case for the left-mover check for a failing iteration that reads `q[i]` occurs when it is preceded by the

```
procedure Lookup(x: data)
returns result: bool;
{
  f := false; i := 0;

  if (*) {
    while (i<n && !f) {
      atomic { f := (q[i] == x); tressa !f; }
      i := i+1;
      }
    assume !f;
  } else {
    while (i<n && !f) {
      atomic { havoc f; }
      i := i+1;
      assume (!f && i<n);
    }
    f := q[i]==x;
    assume (f || i>=n);
    assume f;
  }
  result := f;
}
```

Figure 17. The `Lookup` method after some abstraction and prophecy-tressa annotation.

action `assume q[i]==-1; q[i]:=x;` executed by another thread running `Insert(x)`.[2] Coming after the `Insert`, this iteration of the `Lookup` loop should be succeeding. Coming before the `Insert`, the iteration should be failing. This would imply that this failing iteration $F(i)$ is not a left-mover. But, intuitively, it should never be the case that an `q[i] := x` precede a failing iteration $F(i)$ in `Lookup(x)`. This is precisely what the tressa claim achieves. The left-mover check requires the simulation to hold only at those next states that satisfy the tressa predicate, which here is equal to `!f`. But `q[i]:=x;` followed by `f := q[i]==x;` sets `f` to true. Thus, the tressa claim allows us to ignore this problematic interleaving since any execution in which these two actions appear in that order cannot reach a final state. The tressa claim is discharged with the `assume !f` after this branch is proved to be atomic.

Let us now analyze the succeeding (else) branch. Abstracting the action `f := (q[i] == x)` to `havoc f` allows loop iterations $F(i)$ to commute to the right of actions that write to `q[i]`. The final succeeding iteration is a non-mover and the other actions are left-movers, and are all reduced into a single action. Here, we have implicitly made use of prophecy variable that indicates whether the current loop iteration is the final one or not.

Constituting a typical proof, it is worth repeating what we did in this example from a more general perspective. We started by adding annotations in the form of tressa claims so as to make actions of the proper mover type. This can be perceived as *borrowing* tressa's: an action becomes a mover thanks to the presence of the tressa claim but the correctness of the proof depends on correctly discharging the tressa claim; the proof onus is on the user. This step was followed by reduction by which actions were reduced according to their mover types. If the tressa claims were true and sufficient reduction occurred, each tressa claim would be discharged by a sequential (backward) analysis. This sequential analysis is actually implied by the definition of action composition, given in Sec. 4.4. In our example, we successfully discharged the tressa claims after reducing the loop bodies into single atomic actions.

The final reduced and simplified version of the method is given in Fig. 18.

[2] Imagine that both threads agree on the values of the local variables `i` and `x`.

```
procedure Lookup(x: data)
returns result: bool;
{
 atomic {
  f := false; i := 0;

  if (*) {
    while (i<n && !f) {
      f := (q[i] == x);
      i := i+1;
      }
    assume !f;
  } else {
    havoc f, i;
    assume (!f && i<n);
    f := q[i]==x;
    assume f;
  }
  result := f;
 }
}
```

Figure 18. The `Lookup` method reduced to a single atomic action.

7. Conclusion

In this paper, we incorporated prophecy variables into static verification. We achieved this by augmenting the static verification tool QED with a new proof rule for the introduction of prophecy variables into the program and with a new construct, tressa. We furthermore re-defined correctness and simulation to allow for reasoning in both forward and backward executions. We have demonstrated the usage of this new approach in the atomicity proofs of implementations based on optimistic concurrency.

Our next goal is to statically verify STM (Software Transactional Memory) implementations. Actually, the need for prophecy variables, and in general backwards reasoning in a static setting, manifested itself while we were doing preliminary work on STM verification. The copy and snapshot examples given in this paper encapsulate the notion of optimistic concurrency used in STM implementations.

References

[1] Elmas, T., Qadeer, S., Tasiran, S.: A calculus of atomic actions. In: POPL '09, New York, NY, USA, ACM (2009) 2–15

[2] Lipton, R.J.: Reduction: a method of proving properties of parallel programs. Commun. ACM **18**(12) (1975) 717–721

[3] Larus, J.R., Rajwar, R.: Transactional Memory. Morgan & Claypool (2006)

[4] Kesten, Y., Pnueli, A., Shahar, E., Zuck, L.D.: Network invariants in action. In: CONCUR '02, London, UK, Springer-Verlag (2002) 101–115

[5] Abadi, M., Lamport, L.: The existence of refinement mappings. Theor. Comput. Sci. **82**(2) (1991) 253–284

[6] Ashcroft, E.A.: Proving assertions about parallel programs. J. Comput. Syst. Sci. **10**(1) (1975) 110–135

[7] Owicki, S., Gries, D.: Verifying properties of parallel programs: an axiomatic approach. Commun. ACM **19**(5) (1976) 279–285

[8] Wang, L., Stoller, S.D.: Static analysis for programs with non-blocking synchronization. In: PPoPP '05, ACM Press (2005)

[9] O'Hearn, P.W.: Resources, concurrency, and local reasoning. Theor. Comput. Sci. **375**(1-3) (2007) 271–307

[10] Flanagan, C., Qadeer, S.: A type and effect system for atomicity. SIGPLAN Not. **38**(5) (2003) 338–349

[11] Freund, S.N., Qadeer, S.: Checking concise specifications for multithreaded software. Journal of Object Technology **3** (2004)

[12] Freund, S.N., Qadeer, S., Flanagan, C.: Exploiting purity for atomicity. IEEE Trans. Softw. Eng. **31**(4) (2005) 275–291

[13] Marcus, M., Pnueli, A.: Using ghost variables to prove refinement. In: AMAST '96, London, UK, Springer-Verlag (1996) 226–240

A. Proof of Theorem 1

Below, we construct the proof for the soundness of the proof rule, RED-S. We show that an application of RED-S cannot remove violations from a proof state. We analyze both kinds of violations, backward and forward, separately. In both cases, we assume the existence of a violation in ρ and show how to obtain a violation in ρ'. Roughly speaking, the idea is to show that if a violation in ρ exists, then there is also a violation in ρ such that for any thread t, all occurrences of (t, γ) is immediately preceded by (t, α).

In the backward case, we start with an arbitrary backward violation. We show how to obtain a backward violation in which every (t, α) is immediately followed by its *matching* (t, γ) (Lemma 6). This does not account for *isolated* occurrences of (t, γ) whose matching (t, α) does not occur in the violation at all. We then show how to obtain a backward violation with no such isolated (t, γ) by introducing their matching (t, α) into the backward violation (Lemma 7). It is then trivial to show that the existence of a backward violation in ρ implies the existence of a backward violation in ρ'.

The forward case follows a similar route. We first prove that if there is a forward violation in ρ, then there is a forward violation in which all occurrences of (t, α), except possibly when (t, α) is the very last label of the violation, imply a succeeding (not necessarily immediately) (t, γ) in the same violation (Lemma 13). Then, we show how to obtain a forward violation in which each (t, α) is immediately followed by its matching (t, γ) (Lemma 14).

For the following, we assume that α is a right-mover in ρ, ρ' is obtained from ρ by applying the sequential reduction rule, RED-S, to α and its immediate successor γ. Unless explicitly stated otherwise, a backward violation is assumed to be in the form

$$s_0 \xrightarrow{l_1} s_1 \xrightarrow{l_2} s_2 \ldots \xrightarrow{l_n} s_n$$

where for each $0 < i \leq n$, $l_i = (t_i, \alpha_i)$.

Preserving backward violations.

LEMMA 4. *Let α and β be two atomic actions, $t \neq u$ be two distinct thread id's and s be a program state. Let $\psi_{\alpha[t]\circ\beta[u]}(s)$ be false. If α is a right-mover, then $\psi_{\beta[u]\circ\alpha[t]}(s)$ must also be false. In particular, either $\psi_\alpha[t](s)$ is false, or there exists a state s' such that $s' \xrightarrow{(t,\alpha)} s$ and $\psi_\beta[u](s')$ is false.*

PROOF 4. *The only tricky part is the effect of Θ used in the first condition of right-mover. Observe that, if a state pair (s', s) does not satisfy Θ, we must have $\neg\psi_\alpha[t](s)$. But if $\psi_\alpha[t](s)$ fails, so does $\psi_{\beta[u]\circ\alpha[t]}(s)$. The rest follows from the definition of right-mover, simulation and $\circ$.*

A label $l_i = (t_i, \gamma)$ in a backward violation is *isolated* if $i > 1$ and $l_j = (t_i, \alpha)$ implies that $j > i$. Call an interval $[j, k]$ *safe* for label (t, α), if $0 < j < k$ are two index values such that for all $j < i \leq k$, $\psi_{\alpha_i[t_i]\circ\alpha[t]}(s_i)$ holds and $t_i \neq t$.

LEMMA 5. *Let $\mathbf{r} = \langle s_i \rangle_{0\leq i\leq n}$ be a backward violation in ρ. Let $[j, k]$ be safe for $l_j = (t_j, \alpha)$. Then, the run $\mathbf{r}'$*

$$s_0 \xrightarrow{l_1} s_1 \ldots \xrightarrow{l_{j-1}} s_{j-1} \xrightarrow{l_{j+1}} s'_j \xrightarrow{l_{j+2}} s'_{j+1} \ldots \xrightarrow{l_k} s'_{k-1} \xrightarrow{l_j} s_k \xrightarrow{l_{k+1}} s_{k+1} \ldots \xrightarrow{l_n} s_n$$

is also a backward violation in ρ.

PROOF 5. *Since, $\mathbf{r}'$ starts from the same initial state s_0 and ends at the same final state s_n, if it is a run of $\mathcal{P}$, then it necessarily is a backward violation in ρ. So, we have to show that $\mathbf{r}'$ is a run of $\mathcal{P}$. We prove the latter by induction on the difference $k - j$.*

- *Base Case ($k - j = 0$): $\mathbf{r}'$ is identical to $\mathbf{r}$ which by assumption is a backward violation.*
- *Inductive Hypothesis ($k - j \leq m$, $m \geq 0$): Assume that for any k, j such that their difference is less than or equal to m, $\mathbf{r}'$ is a run of $\mathcal{P}$.*
- *Inductive Step ($k - j = m + 1$): Consider the actions of l_j and l_{j+1}, α and α_{j+1}, respectively. Since, α is a right-mover and by assumption, $\psi_{\alpha_{j+1}[t_{j+1}]\circ\alpha[t_j]}(s_{j+1})$ holds, by definition of right-mover and simulation, $\tau_\alpha[t_j] \cdot \tau_{\alpha_{j+1}}[t_{j+1}](s_{j-1}, s_{j+1})$ implies $\tau_{\alpha_{j+1}}[t_{j+1}] \cdot \tau_\alpha[t_j](s_{j-1}, s_{j+1})$. Then, the sequence*

  $$s_0 \xrightarrow{l_1} \ldots \xrightarrow{l_{j-1}} s_{j-1} \xrightarrow{l_{j+1}} s'_j \xrightarrow{l_j} s_{j+1} \xrightarrow{l_{j+2}} \ldots s_n$$

  is a run in ρ. Applying the inductive hypothesis to k and $j + 1$ which is the new index of (t_j, α) completes the proof.

A label $l_j = (t_j, \alpha_j)$ is *unmatched* in $\mathbf{r}$, if $\alpha_j = \gamma$, $j > 1$, $t_{j-1} \neq t_j$, and l_j is not isolated.

LEMMA 6. *Let ρ contain a backward violation. Then, ρ contains a backward violation which has no unmatched labels.*

PROOF 6. *Let X be the set of backward violations in ρ. Let $Y \subseteq X$ consist of only those elements in X with shortest length. Let $\mathbf{r_y} \in Y$ be such that it has a minimal number of unmatched labels. To prove the lemma by contradiction, we assume that the number of unmatched labels, m, in $\mathbf{r_y}$ is greater than 0. Pick the rightmost unmatched label l_k for some $k > 1$. That is, for any $i > k$, $t_i \neq t_k$ and $\alpha_i = \gamma$ implies that either $l_{i-1} = (t_i, \alpha)$ or l_i is isolated. Let l_j be the matching label for l_k. Let $i < k$ be such that $[i, k - 1]$ is safe, but $[i - 1, k - 1]$ is not safe, for l_j. Consider the following two cases:*

- *$i > j$. By the choice of i, this means that $\psi_{\alpha_{i-1}[t_{i-1}]\circ\alpha[t_j]}(s_{i-1})$ is false. This in turn implies that either*

  $$s'_{i-2} \xrightarrow{l_j} s_{i-1} \xrightarrow{l_i} s_i \ldots \xrightarrow{l_n} s_n$$

  or

  $$s_{i-1} \xrightarrow{l_i} s_i \ldots \xrightarrow{l_n} s_n$$

  is a backward violation in ρ. Since both have length strictly less than n ($n - i + 2$ with $i > 2$), this contradicts the assumption that $\mathbf{r_y}$ belongs to Y.

- $i \leq j$. *In this case, we have* $[j, k-1]$ *safe for* l_j. *By Lemma 5, the following*

$$s_0 \xrightarrow{l_1} \dots s_{j-1} \xrightarrow{l_{j+1}} s'_j \xrightarrow{l_{j+2}} s'_{j+1} \dots s'_{k-2} \xrightarrow{l_j} s'_{k-1} \xrightarrow{l_k} s_k \dots \xrightarrow{l_n} s_n$$

is also a backward violation in ρ. *Since this run has one less unmatched label than and the same length as* $\mathbf{r_y}$, *its existence contradicts the assumption that* $\mathbf{r_y}$ *had the minimum number of unmatched labels among the backward violations of length* n.

The initial assumption that $m > 0$ *is false. So, there exists a backward violation with no unmatched labels.*

For a backward violation $\mathbf{r}$, let $len(\mathbf{r})$ and $iso(\mathbf{r})$ denote the length of and the number of isolated labels in $\mathbf{r}$. Call a run of length n α, γ-*matched* if for any $j < n$, $l_j = (t_j, \alpha)$ implies that $l_{j+1} = (t_j, \gamma)$.

LEMMA 7. *Let* ρ *contain a backward violation. Then, there exists an* α, γ*-matched backward violation in* ρ.

PROOF 7. *Let* X *be the set of backward violations that do not have unmatched labels. By Lemma 6,* X *is non-empty. Let* Y *contain all elements in* X *that satisfy* $len(\mathbf{r}) + iso(\mathbf{r})$ *is minimal in* X. *Pick* $\mathbf{r_y} \in Y$ *such that its number of isolated labels is minimal in* Y. *Let* $n = len(\mathbf{r_y})$ *and* $n_i = iso(\mathbf{r_y})$. *We will prove that* $n_i = 0$. *To prove it by contradiction, assume that* $n_i > 0$. *Let* $l_k = (t_k, \gamma)$ *be the rightmost isolated label in* $\mathbf{r_y}$. *Then, the matching label for* l_k *is* $l_0 = (t_k, \alpha)$. *Let* $i < k$ *be such that* $[i, k-1]$ *is safe, but* $[i-1, k-1]$ *is not safe, for* l_0. *Consider the following cases:*

- $i > 1$, *or* $i = 1$ *and* $\psi_\alpha[t_k](s_0)$ *is false. By the choice of* i, *this means that* $\psi_\alpha[t_k](s_{i-1})$ *is false, which implies that* $\psi_{\alpha[t_k] \circ \alpha_i[t_i]}(s_i)$ *is also false, by the definition of* $\circ$. *By Lemma 4,* $\psi_{\alpha_i[t_i] \circ \alpha[t_k]}(s_i)$ *is also false. But, since by the choice of* i, $\psi_\alpha[t_k](s_i)$ *is true, there exists a state* s' *such that* $\psi_{\alpha_i}[t_i](s')$ *is false and* $s' \xrightarrow{l_0} s_i$ *holds. This in turn implies that*

$$s'_{i-1} \xrightarrow{l_0} s_i \xrightarrow{l_{i+1}} s_{i+1} \dots \xrightarrow{l_n} s_n$$

is a backward violation in ρ. *By Lemma 5, the run above implies the existence of*

$$s'_{i-1} \xrightarrow{l_{i+1}} s'_i \dots \xrightarrow{l_{k-1}} s'_{k-2} \xrightarrow{l_0} s_{k-1} \xrightarrow{l_k} \dots \xrightarrow{l_n} s_n$$

which is still a backward violation with no unmatched labels. This run has length $n - i + 1$, *which is at most* n *with* $i > 0$, *but at most* $n_i - 1$ *isolated labels. But this contradicts with the assumption that* $\mathbf{r_y}$ *belongs to* Y *and has minimal number of isolated labels.*
- $i = 1$ *and* $\psi_\alpha[t_k](s_0)$ *is true. Since* $\mathbf{r_y}$ *is a backward violation, there must exist a label* (u, β) *such that* $\psi_\beta[u](s_0)$ *is false. Because* α *is a right-mover, and the choice of* i, *Lemma 4 implies that there exists* s' *such that* $\psi_\beta[u](s')$ *is false and* $s' \xrightarrow{l_0} s_0$ *holds. Then, the following*

$$s' \xrightarrow{l_0} s_0 \xrightarrow{l_1} s_1 \dots \xrightarrow{l_n} s_n$$

is a backward violation with length $n + 1$ *and* $n_i - 1$ *isolated labels. By Lemma 5,*

$$s' \xrightarrow{l_1} s'_0 \xrightarrow{l_2} \dots \xrightarrow{l_{k-1}}{}' s'_{k-2} \xrightarrow{l_0} s_{k-1} \xrightarrow{l_k} s_k \dots \xrightarrow{l_n} s_n$$

is also a backward violation. Observe that this run has no unmatched labels and hence is an element of X. *Since the sum of its length and the number of isolated labels it contains is* $n + 1 + n_i - 1 = n + n_i$, *by assumption it is also an element of* Y. *And since it contains fewer isolated labels than* $\mathbf{r_y}$, *it contradicts with the assumption that* $\mathbf{r_y}$ *contained the minimum number of isolated labels among the elements of* Y.

Thus, the assumption that $n_i > 0$ *is false. Since* $\mathbf{r_y}$ *is a backward violation with no unmatched labels and no isolated labels, it is by definition* α, γ*-matched.*

LEMMA 8. *Let* ρ *contain an* α, γ*-matched backward violation. Then,* ρ' *contains a backward violation.*

PROOF 8. *Let* $\mathbf{r}$ *be an* α, γ*-matched backward violation in* ρ. *First, consider the initial transition,* l_1. *If* $l_1 = (t_1, \gamma)$, *there are two possibilities:*

- $\psi_\alpha[t_1](s_0)$ *is false. In this case, the definition of* $\circ$ *implies that* $\psi_{\alpha \circ \gamma}[t_1](s_1)$ *is false.*
- $\psi_\alpha[t_1](s_0)$ *is true. Then, there must exist a label* (u, β) *such that* $u \neq t_1$ *and* $\psi_\beta[u](s_0)$ *is false. By Lemma 4, there exists a state* s' *such that* $\psi_\beta[u](s')$ *is false and* $s' \xrightarrow{(t_1, \alpha)}$ *holds.*

So, without loss of generality, we can assume that the α, γ*-matched backward violation does not start with a label* (t, γ) *for any* t. *The backward violation in* ρ', $\mathbf{r}'$, *starts from the same state* s_0 *and makes the same transitions as* $\mathbf{r}$ *as long as the label does not contain an* α. *Whenever* $s_i \xrightarrow{(t,\alpha)} s_{i+1} \xrightarrow{(t,\gamma)} s_{i+2}$ *occurs in* $\mathbf{r}$, *we let* $s_i \xrightarrow{(t, \alpha \circ \gamma)} s_{i+2}$ *in* $\mathbf{r}'$ *and continue from* s_{i+2}. *That this constructs a run in* ρ' *follows from the definition of* $\circ$ *and the construction of* $\mathcal{P}'$.

LEMMA 9. *Let* ρ *contain a backward violation. Then,* ρ' *contains a backward violation.*

PROOF 9. *By Lemma 7,* ρ *contains an* α, γ*-matched backward violation. By Lemma 8,* ρ' *contains a backward violation.*

Preserving forward violations.

A run is a *minimal* forward violation in ρ, if it is a forward violation in ρ and any of its prefix is not. A run is a *shortest* forward violation in ρ if there does not exist a forward violation in ρ of a shorter length.

LEMMA 10. *A shortest forward violation is also minimal.*

PROOF 10. *Follows from the definitions of shortest and minimal.*

LEMMA 11. *Let $\mathbf{r} = \langle s_i \rangle_{0 \leq i \leq n}$ be a shortest forward violation. If $l_n = (t_n, \alpha)$, then $(u, \beta) \in \mathsf{fst}(s_n)$ and $\neg\phi_\beta[u](s_n)$ imply that $u = t_n$ and $\beta = \gamma$.*

PROOF 11. *Assume the contrary. Let (u, β) be such that $u \neq t_n$ and $\phi_\beta[u](s_n)$ evaluates to false. By the definition of right-mover (second condition), $\phi_\beta[u](s_{n-1})$ must also be false. This contradicts the minimality of $\mathbf{r}$.*

A label $l_i = (t_i, \alpha)$ in a run is *isolated* if $l_j = (t_i, \gamma)$ implies that $j < i$. The *isolating distance* of a run is given as $n - j$ where n is the length of the run, j is the index of the rightmost isolated label (for all isolated labels l_k in $\mathbf{r}$, we have $j \geq k$).

LEMMA 12. *If there is an isolated label in a shortest forward violation, then there is a shortest forward violation which has an isolated label as the last label of the run.*

PROOF 12. *Consider X, the set of all shortest forward violations which contain an isolated label. Out of X, pick a run $\mathbf{r}'$ with a minimal isolation distance. Showing that $\mathbf{r}'$ has isolation distance 0 will prove the lemma. Assume contrary and let the isolation distance of $\mathbf{r}'$ be $m > 0$. Set $j = n - m$. This means that $l_j = (t_j, \alpha)$. First, observe that since $\mathbf{r}'$ is a minimal forward violation, $\phi_\alpha[t_j](s_j)$ evaluates to true. Since $j < n$, there is a label $l_{j+1} = (t_{j+1}, \beta)$. Since l_j is isolated, $t_j \neq t_{j+1}$. Since α is a right-mover,*

$$\alpha[t_j] \circ \beta[t_{j+1}] \preceq \beta[t_{j+1}] \circ \alpha[t_j]$$

must hold. Note that, $\Theta(s_j, s_{j+2})$ evaluates to true because $\phi_\alpha[t_j](s_j)$ evaluates to true. Since $\mathbf{r}'$ is a shortest forward violation, the simulation given above can only hold when $s_j \xrightarrow{(t_{j+1},\beta),(t_j,\alpha)} s_{j+2}$. Thus, the run which differs from $\mathbf{r}'$ only in the order of the j^{th} and $(j+1)^{th}$ labels is also a forward violation. However, this new run has an isolation distance $n - (n - m + 1) = m - 1$. This contradicts the initial assumption that $\mathbf{r}'$ has a minimal non-zero isolation distance. Thus, there exists in X a forward violation whose isolation distance is 0.

LEMMA 13. *Let $\mathbf{r} = \langle s_i \rangle_{0 \leq i \leq n}$ be a shortest forward violation. Then, $\mathbf{r}$ contains at most one isolated label.*

PROOF 13. *Let $l_j = (t_j, \alpha)$, $l_k = (t_k, \alpha)$ be isolated labels. Then, following the argument in the previous lemma, we can obtain a shortest forward violation which has $l_n = (t_j, \alpha)$. By Lemma 11, this implies that the only label in $\mathsf{fst}(s_n)$ whose assertion is violated at s_n is (t_j, γ). Similarly, we can obtain a shortest forward violation which $l_n = (t_k, \alpha)$. Again, by Lemma 11, this implies that the only label in $\mathsf{fst}(s_n)$ whose assertion is violated at s_n is (t_k, γ). Since γ is the unique successor of α and l_j, l_k are isolated labels, we must have $j = k$.*

Call a label $l_j = (t_j, \alpha)$ *unmatched*, if l_j is not isolated and $t_{j+1} \neq t_j$.

LEMMA 14. *Let ρ contain a forward violation. Then, there is a shortest forward violation $\mathbf{r}'$ which is α, γ-matched.*

PROOF 14. *By Lemma 13, we can assume that in a shortest forward violation there is at most one isolated label and in case it exists, we can assume that it occurs as the last label, l_n. Let Y be the set of all shortest forward violations. Out of Y, pick a run $\mathbf{r_m}$ which has the least number of unmatched labels. We need to prove that $\mathbf{r_m}$ is α, γ-matched. Assume the contrary and let a be the number of unmatched labels in $\mathbf{r_m}$. Let l_j be the rightmost unmatched label in $\mathbf{r_m}$. Since l_j is not isolated, there exists some $k > j+1$ such that $l_k = (t_j, \gamma)$. Choose k such that $j < o < k$ implies that $t_o \neq t_j$. In other words, all the labels between j and k belong to different threads. Since α is a right-mover, $\mathbf{r_m}$ is a shortest violation (no assertions can fail at intermediate states), the first condition of right-mover must hold. Using the same reasoning as was done in the proof of Lemma 12, we obtain the run*

$$s_0 \xrightarrow{l_1} s_1 \ldots s_j \xrightarrow{l_{j+1}} s'_{j+1} \ldots s'_{k-2} \xrightarrow{l_j} s_{k-1} \xrightarrow{l_k} s_k \ldots \xrightarrow{l_n} s_n$$

which is a shortest forward violation. Since the relative ordering of labels among $\{l_i\}_{i \neq j}$ remains the same, this run has one less unmatched label, contradicting the assumption that $\mathbf{r_m}$ has a non-zero number of unmatched labels. Thus, $\mathbf{r_m}$ is α, γ-matched.

LEMMA 15. *If $\mathbf{r}$ is an α, γ-matched forward violation in ρ, then there is a forward violation in ρ'.*

PROOF 15. *Similar to Lemma 9.*

LEMMA 16. *If ρ contains a forward violation, then ρ' contains a forward violation.*

PROOF 16. *Follows from Lemma 14 and Lemma 15.*

LEMMA 17. *Let $(\mathcal{P}, \mathcal{I}) \dashrightarrow (\mathcal{P}', \mathcal{I})$ be a proof step which applies the sequential reduction rule,* RED-S. *If $(\mathcal{P}', \mathcal{I})$ does not contain a violation, neither does $(\mathcal{P}, \mathcal{I})$.*

PROOF 17. *We have shown how to obtain a violation in ρ' from a violation in ρ when we took α as a right-mover. The case of γ of the proof rule* RED-S *being a left-mover is similar. This is due to the duality between forward and backward reasoning and the accompanying definitions. More explicitly, when γ is a left-mover, a forward violation in ρ' is constructed in the same way as a backward violation in ρ' was constructed when α was a right-mover. Similarly, when γ is a left-mover, a backward violation in ρ' is constructed in the same way as a forward violation in ρ' was constructed when α was a right-mover.*

PROOF 18 (Theorem 1). *Proof is by induction on the length of the proof. The base case, a proof of length 0, is trivial. The inductive step has to show that soundness is preserved for each rule application. The proof of* ANNOT-H *is similar to the proof of Lemma 2 whose sketch is given in the paper. The proof of* INV *is trivial. The proof of* SIM *is again sketched in the paper (Lemma 1). The proof of* RED-C *follows from the proof* SIM *since* γ *of* RED-C *simulates* $\alpha\Box\gamma$*. The proof of* RED-S *is given above. The proof of* RED-L *follows from the proofs of* RED-S *and* SIM *(think of* β *of the rule* RED-L *as simulating zero or more iterations of* α*).*