A Framework for Formally Verifying Software Transactional Memory Algorithms

Mohsen Lesani[1], Victor Luchangco[2], and Mark Moir[2]

[1] University of California, Los Angeles, USA
lesani@ucla.edu
[2] Oracle Labs, Burlington, MA, USA
{victor.luchangco,mark.moir}@oracle.com

Abstract. We present a framework for verifying transactional memory (TM) algorithms. Specifications and algorithms are specified using I/O automata, enabling hierarchical proofs that the algorithms implement the specifications. We have used this framework to develop what we believe is the first fully formal machine-checked verification of a practical TM algorithm: the NOrec algorithm of Dalessandro, Spear and Scott.

Our framework is available for others to use and extend. New proofs can leverage existing ones, eliminating significant work and complexity.

1 Introduction

As multicore computing becomes ubiquitous, it is increasingly important to support effective concurrent programming for a wide range of programmers. *Transactional memory* (TM) [9] allows programmers to specify a sequence of operations on shared objects that should be executed as a transaction that appears to be applied without interference from concurrent transactions, and without concurrent transactions observing partial results of the sequence. Programmers do not specify *how* these guarantees are made; this is a responsibility of the system. TM aims to deliver to shared memory programmers the benefits that transactions provide to database programmers.

We present a framework for specifying the guarantees that a TM system must provide (i.e., the TM specification), modeling TM implementations, and verifying that the implementations provide the specified guarantees. Our framework is based on I/O automata and simulation proof techniques [11,12], which support hierarchical proofs by modeling both specifications and implementations as automata and proving simulation relations between these automata. The hierarchical proof approach allows a proof for one TM algorithm to leverage parts of the hierarchy constructed for other TM algorithms, thus significantly improving productivity. The framework is formalized in the PVS language [14,16].

Using this framework, we have achieved the first fully formal machine-checked verification of a practical TM algorithm, the NOrec algorithm [3]. As described in [10], we have also recently used the framework to clarify relationships between the TMS1, TMS2, and opacity correctness conditions (see Section 2.1). The primary goal of this paper is to give readers a concrete understanding of the nature of our framework and proofs, and to make the framework more approachable. Readers interested in more detail can contact us to obtain the framework and explore our proofs interactively using PVS.

Section 2 presents background on TM correctness conditions and algorithms, particularly the NOrec algorithm we have verified. Section 3 contains background on I/O automata and simulation proof techniques. Section 4 describes our framework, and Section 5 shares lessons we have learned that have made it significantly easier to construct and reuse proofs. We briefly summarize related work in Section 6, and conclude in Section 7.

2 Transactional Memory

A *transactional memory system* supports one or more shared objects, typically a memory object consisting of a set of locations, each of which supports read and write operations. A sequence of operations on such objects can be executed as a transaction. To guarantee that transactions appear not to interleave with each other, a transaction may sometimes *abort* so that it appears not to execute at all. Transactions that successfully complete are said to *commit*.

2.1 Specifications

Verifying a TM implementation requires a precise specification of what it means for it to be correct. No single TM correctness condition is universally accepted, and indeed, different conditions are appropriate for different contexts. We have recently studied this problem for TM algorithms intended to support transactional language features in languages such as C and C++ [5]. To avoid fatal errors such as divide-by-zero in this context, transactions—even those that ultimately abort—must observe behavior that is consistent with some execution in which all transactions that commit do so instantaneously [5,8]. Traditional correctness conditions for transactions in database systems—such as *serializability* [15]—do not ensure this.

In [5], we defined a general condition TMS1 and a more restrictive condition TMS2. TMS1 aims to allow all implementations that provide reasonable behavior for the intended context, and as a result is somewhat abstract. TMS2 is more restrictive, but is closer to the intuition behind many practical TM algorithms. Briefly, TMS2 requires a writing transaction to append a new state to a sequence of memory states during its commit operation, while a read-only transaction is allowed to read from any state that was the last state in that sequence at some point during the execution of the transaction. We proved in [5] that TMS2 implements TMS1, and we have recently proved the same result again using our framework, specifically by proving that TMS2 implements opacity [8], and opacity implements TMS1, thereby clarifying the relationships between these conditions and confirming our conjecture [5]. This result implies that, in order to prove that an algorithm satisfies the TMS1 condition, it suffices to prove that it satisfies TMS2. This is the approach we have taken for our NOrec proof.

2.2 The NOrec Algorithm

NOrec [3] significantly reduces low-contention overhead as compared to previous TM algorithms such as TL2 [4] by eliminating *ownership records*, which hold TM metadata that is used when an associated location is accessed. NOrec achieves this by using a sequence lock (seqlock) that is acquired by every transaction that successfully writes any shared location. The seqlock is implemented by a counter that is incremented upon acquisition and release: the lock is free when the counter's value is even; it is held by the transaction that most recently incremented it when the value is odd. Although this lock limits scalability, it is held only while a transaction is committing, and NOrec's low overhead makes it attractive in low-to-moderate contention workloads.

Briefly, NOrec works as follows: When a transaction begins, it checks that the seqlock is free, and records a "snapshot" of the lock value. (Whenever a transaction discovers the seqlock is held by some other transaction, it waits until the lock is released before continuing.) To write a shared location, a transaction records the location and the value to be written to it in a private *write set.* These changes are written to the shared locations only when the transaction commits.

A transaction records values it reads in its private *read set.* After reading a location l, a transaction checks that the lock value has not changed since the transaction's most recent snapshot. If the lock value has changed, then the transaction revalidates its read set by updating its snapshot of the lock and checking that every object in its read set has the previously recorded value (aborting if not), before reading location l again and checking that the lock value has not changed again. This process is repeated until the transaction aborts or the read set validation and subsequent rereading of l is successful; in the latter case, the value read from l is stored in the transaction's read set and returned to the transaction.

To commit, a transaction attempts to acquire the lock while ensuring that its value has not changed since the transaction's most recent snapshot. (If it has, the transaction revalidates its reads and refreshes its snapshot as described above before attempting again to acquire the lock.) After acquiring the lock, the transaction performs the writes recorded in its writeset and then releases the lock by incrementing its value once more. Because no transaction reads any location while the lock is held, the writes performed by a transaction while it is committing appear atomic to all other transactions.

3 Theory Background

In Sections 3.1 and 3.2, we briefly summarize the standard I/O automata theory and simulation proof techniques upon which our framework is built. We have not only formalized this theoretical foundation in PVS, but also verified within the framework the theorems from the literature that we have used.

3.1 Automata

We use simplified[1] *input/output automata* (IOAs) [11] to express TM correctness conditions and to model TM algorithms. An automaton A is a labeled transition system that consists of: a set $\mathit{states}(A)$ of states, with a nonempty subset $\mathit{start}(A) \subseteq \mathit{states}(A)$ of start states; a nonempty set $\mathit{acts}(A)$ of actions, partitioned into *external* and *internal* actions; and a transition relation $\mathit{trans}(A) \subseteq \mathit{states}(A) \times \mathit{acts}(A) \times \mathit{states}(A)$.We describe the states using a collection of *variables*, and the transition relation using a *precondition* (a predicate on states) and an *effect* (a set of assignments to variables) for each action.

An *execution fragment* of A is a sequence $s_0 a_1 s_1 \ldots$ of alternating states and actions of A such that $(s_{k-1}, a_k, s_k) \in \mathit{trans}(A)$ for all $k > 0$; a finite sequence must end with a state. An *execution* is an execution fragment with $s_0 \in \mathit{start}(A)$. A state is *reachable* if it appears in some execution. An *invariant* is a predicate that is true for all reachable states; it is typically proved by induction on the length of an execution.

The subsequence of external actions in an execution fragment is called its *trace*, and represents its externally visible *behavior*. The traces of an automaton A are the traces of its executions; we denote the set of such traces by $\mathit{traces}(A)$. These traces therefore represent the behavior that the automaton can exhibit.

We can interpret an automaton as a specification and as an implementation. For an "abstract" automaton A, interpreted as a specification, and a "concrete" automaton C, interpreted as an implementation, C *implements* A iff $\mathit{traces}(C) \subseteq \mathit{traces}(A)$: every behavior of the implementation is allowed by the specification.

This dual interpretation of automata enables *hierarchical proofs*: If automaton C implements another automaton B, and B implements automaton A, then C also implements A. When proving that one automaton implements another, it is often helpful to introduce "intermediate" automata to break the proof into more manageable pieces. These intermediate automata may represent classes of implementations that share common approaches and ideas, allowing proofs of implementations in the class to reuse properties already proved for the class, as discussed further in Section 4.2.

3.2 Simulation Proofs

One way to prove that C implements A is to use a *simulation relation* [12], which establishes a correspondence (not necessarily 1-1) between $\mathit{states}(C)$ and $\mathit{states}(A)$ such that for each step in any execution of C, there is a finite execution fragment of A with the same trace whose first and last states correspond to the pre- and post-states of the step, and execution fragments for successive steps can be "pasted together" into a single execution of A.

A *forward simulation* from C to A, for example, requires that every start state of C correspond to some start state of A, and that, for every step of an

[1] Our automata are simplified because we have not yet needed to explicitly compose automata and we have concentrated only on safety properties. We anticipate adding support for composition soon as it is needed for our ongoing work.

execution of C and every state of A corresponding to the prestate, there is a corresponding execution fragment of A that starts from that state and has the same external action, if any, as the step of C. Thus, given a forward simulation from C to A, and an execution of C, we can construct an execution of A with the same trace by starting from a corresponding start state and extending the execution with a corresponding execution fragment for each successive step of C. This implies that every trace of C is a trace of A.

Lemma 1. *If there is a forward simulation from C to A then C implements A.*

A forward simulation that is a function on the states of C is a *refinement.*

Sometimes, a forward simulation cannot prove that C implements A because knowledge of the future is needed in order to choose an appropriate execution fragment for a step of an execution of C. In such cases, *backward simulations* can be used. The conditions for a backward simulation are similar to those for forward simulations, but they allow an abstract execution to be constructed by working backwards from the last state of a (finite) execution of C, thus allowing use of knowledge of the future.

4 A Framework for Verifying TM Implementations

Our framework uses the PVS system [14,16], which supports a specification language based on typed higher-order logic, and tools for working in this language, including an interactive theorem prover that provides inference rules and decision procedures that are used in proofs. User guidance for a theorem can be saved and rerun for repeatable verification and can also be edited and applied to other theorems. Users can combine inference rules into high-level "strategies" that simplify proofs and promote reuse.

The foundation of our framework is a set of PVS theories that describe automata and simulations, as well as definitions and lemmas that facilitate reasoning about them. These foundational concepts are not TM-specific.

Our framework further comprises specific automata specifying TM correctness conditions (such as TMS2) and implementations (such as NOrec). We use several automata modeling specifications and implementations in varying levels of detail to construct hierarchical proofs that, for example, a detailed model of the NOrec algorithm correctly implements the TMS2 condition. All our proofs have been checked by the PVS prover. This section overviews our framework.

4.1 Foundations: Automata and Simulations

It is convenient, when defining an automaton in PVS, to have a single type that encompasses all its actions. In standard I/O automata theory, a simulation between two automata requires them to have the same external actions. This implies that all the actions of all automata in a proof hierarchy must be of the same type. Changes to this type—to add internal actions for a new automaton, for example—affect all automata in that proof hierarchy, triggering obligations to reverify every lemma and invariant, even those unrelated to the changes. This was a problem in some of our previous proofs, and is unacceptable in the context of developing a framework that includes many automata.

We address this problem by splitting an automaton into a *basic* automaton, which specifies its states, actions and transitions (`Automaton` theory in Figure 1), and a *view*, which maps its external actions to external *events* (`View` theory in Figure 2). Only the events need to be shared among automata.

To define a basic automaton, we define a type for states (usually a record type with components for modeling shared variables, private variables, control states, etc.), a type for actions, a predicate over the states to identify initial states, and a predicate that specifies the legal steps of the automaton. We then import the `Automaton` theory, shown in Figure 1, instantiating it with these elements.

The `Automaton` theory defines key properties of a basic automaton, such as its finite execution fragments, and what it means for a state to be reachable and for a state predicate to be an invariant of the automaton. We also prove several lemmas (not shown) that help us manipulate executions and prove invariants. For example, we use the following lemma to prove invariants by induction:

```
invariantInduction: LEMMA
  FORALL (p: pred[State]):
    (FORALL s: start(s) IMPLIES p(s))
    AND
    (FORALL s0, a, s1:
       reachable(s0) AND reachable(s1) AND p(s0) AND trans(s0,a,s1)
           IMPLIES p(s1))
    IMPLIES invariant(p)
```

(Although `reachable(s1)` is redundant—it is implied by `reachable(s0)` and `trans(s0,a,s1)`—we include it for convenience, as it allows us to apply already-established invariants to the poststate `s1` without proving each time that the poststate is reachable.)

The `View` theory (Figure 2) is parameterized by types for events and actions, a predicate identifying external actions, and a map from those actions to events, which we call a *view*. This theory defines the trace of a sequence of actions to be the subsequence of those actions that are external, mapped to events by the specified view. The `AutomatonWithView` theory (Figure 2) puts together a basic automaton and a view to define an automaton and its set of traces.

Views allow us to use different types for the actions of different automata, while retaining the ability to express that an external action of one automaton is "equal to" an external action of another, by mapping each to the same event. When views are 1-1 mappings, as they are in all our work to date, there is a straightforward isomorphism between automata in the standard theory and our "automata with views".

Views also add flexibility in modeling algorithms and specifications because multiple external actions of an automaton can be mapped to the same event. For example, when the actions of an automaton are deterministic (i.e., the post-state of a transition is uniquely determined by the prestate and the action), we

```
Automaton[State, Action: TYPE+,
          start: nonempty_pred[State],
          trans: pred[[State,Action,State]]]: THEORY
BEGIN

Step: TYPE = [State, Action, State]

IMPORTING finseq_props[State]

FiniteStepSeq: TYPE = [# actions: finseq[Action],
                         states: { ss: nonempty_finseq[State] |
                                   ss`length = actions`length + 1 }
                       #]
stepseq: VAR FiniteStepSeq

length(stepseq): nat = stepseq`actions`length

first(stepseq): State = first(stepseq`states)

last(stepseq): State = last(stepseq`states)

steps(stepseq): finseq[Step] =
  (# length := stepseq`actions`length,
     seq := LAMBDA (n: below[stepseq`actions`length]):
              (stepseq`states(n), stepseq`actions(n), stepseq`states(n+1))
   #)

finiteExecFrag(stepseq): bool =
  FORALL (n: below[length(stepseq)]): trans(steps(stepseq)(n))

finiteExecution(stepseq): bool =
  finiteExecFrag(stepseq) AND start(first(stepseq))

reachable(s: State): INDUCTIVE bool =
  start(s) OR (EXISTS (s0: State, a: Action): reachable(s0) AND trans(s0,a,s))

invariant(p: pred[State]): bool = FORALL (s: State): reachable(s) IMPLIES p(s)

END Automaton
```

Fig. 1. Definitions in `Automaton.pvs`

can specify the effect of actions with a function, which has various advantages, especially for automated theorem provers. For internal actions that are nondeterministic, we can create a variant of the automaton in which such actions have additional parameters, so that each parameterized action is deterministic. However, we cannot add parameters to a nondeterministic external action in standard I/O automata theory because doing so would change the externally visible behavior. Using automata with views, we can map each parameterized action to the same event as the original action.

The `Simulations` theory (not shown) takes as parameters the components for two automata, the events type that they share, and views mapping their

```
View[Event, Action: TYPE+,
     external: pred[Action],
     view: [(external) -> Event]]: THEORY
BEGIN

IMPORTING filter_props[Action]

trace(acts: finseq[Action]): finseq[Event] =
  map[(external),Event](view)(filter(external)(acts))

END View

AutomatonWithView[Event, State, Action: TYPE+,
                  start: nonempty_pred[State],
                  trans: pred[[State,Action,State]],
                  external: pred[Action],
                  view: [(external) -> Event]]: THEORY
BEGIN

IMPORTING Automaton[State, Action, start, trans]
IMPORTING View[Event, Action, external, view]

trace(stepseq: FiniteStepSeq): finseq[Event] = trace(stepseq`actions)

finiteTrace(eventseq: finseq[Event]): bool =
  EXISTS (fexec: (finiteExecution)): trace(fexec) = eventseq

END AutomatonWithView
```

Fig. 2. `View` and `AutomatonWithView` theories

respective external actions to events. It defines forward simulations and refinements, and also proves some lemmas (not shown). For example, the equivalent of Lemma 1 in our context states that the existence of a forward simulation implies finite trace inclusion between the two automata (`R` is universally quantified, and `CA` and `AA` are aliases for the two automata that are created by instantiating the `AutomatonWithView` theory with their components):

```
forwardSimulationImpliesFiniteTraceInclusion: LEMMA
  forwardSimulation(R) IMPLIES subset?(finiteTraces(CA), finiteTraces(AA))
```

Thus, one can prove that one automaton implements another by instantiating the `Simulations` theory with these automata, specifying a relation between their states, and proving that the relation satisfies the definition of a forward simulation. Similar definitions and lemmas are included for refinements. In addition to standard refinements, we define "simple refinements", in which each step of the concrete automaton corresponds to at most one abstract action. When it holds, this condition is more convenient to use as it avoids the need to specify and manipulate execution fragments.

In separate PVS theories (not shown), we also define backward simulations and *history mappings* [12], and prove similar lemmas about them. A history mapping between two automata is equivalent to a forward simulation from the first to the second and a refinement from the second to the first, showing that the automata are equivalent. We provide a rule for proving history mappings that requires less work than proving the two properties separately.

Our framework further comprises a set of PVS *strategies*, which help us to automate and hide parts of proofs. For example, by structuring our automata consistently, we can write strategies that automatically perform the mundane "unpacking" of definitions, thus making it easier to both construct and read proofs. We do not discuss our strategies further in this paper, but they is documented in the release notes of our framework.

4.2 TM-Specific Automata Included in the Framework

We define a number of TM-specific automata using the foundations described above. These automata, and the relationships we have proved between them, are depicted in Figure 3. The `TMS2` automaton produces exactly the set of traces allowed by the TMS2 condition presented in [5].

To prove that NOrec implements TMS2, we construct a hierarchical proof using several intermediate automata, each modeling a successively more detailed version of NOrec. In the simplest version, `NOrecAtomicCommitValidate`, the reading of shared objects (including checking that the global sequence lock is not held), validating a transaction, and committing a transaction (including writing all the changes in its write set) are each done in a single atomic step. No lock is needed in this version, because the lock is held only while a transaction is committing, which occurs in a single step in this automaton.

In `NOrecDerived`, validation and committing are no longer atomic, but reading a shared object and checking the global sequence number still is. `NOrec` models an abstract version of the NOrec algorithm, in which each step accesses at most one shared variable. Together, the proofs between these automata (Figure 3) verify an abstract version of NOrec that is consistent with synchronization support in real systems. However, we go one step further.

The `NOrecPaperPseudocode` automaton is a straightforward encoding of the pseudocode in [3], explicitly modeling details such as the control flow presented in [3]. For example, we explicitly use program counter values like begin2, corresponding to line 2 of the Begin procedure (Listing 3 in [3]), and validate6start and validate6iter, corresponding respectively to line 6 just before initializing the loop and just before executing the body of the loop beginning on line 6 of the Validate procedure (Listing 2 in [3]).

If the code for the NOrec algorithm were refactored without fundamentally changing it, we could verify the new version simply by repeating this last step for a different automaton encoding the new pseudocode, thus effectively reusing all of the more substantial proofs above the `NOrec` automaton in the hierarchy.

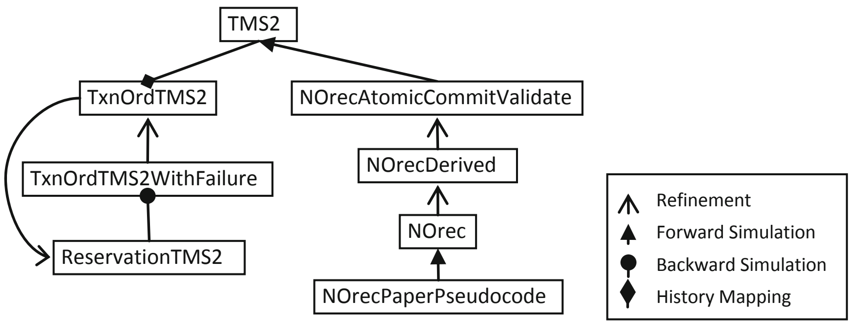

Fig. 3. Relationships between TM-specific automata in our framework. Direction of history map arrows indicates the forward simulation.

Some TM algorithms cannot be proved to implement TMS2 by a forward simulation. For example, in TL2 [4], a transaction "validates" the reads it has performed using a technique that ensures that its reads were consistent at the *beginning* of the validation process, but only determines that the validation was successful later. Thus, the transaction must take effect *before* it is known to have committed successfully. Exploiting such "knowledge of the future" in a simulation proof requires a backward simulation. Verifying a backward simulation can be challenging because it requires reasoning about extending an execution backwards from a poststate to a prestate.

To facilitate verification of such algorithms, we provide an alternate formulation of the TMS2 correctness condition as an automaton `ReservationTMS2`, in which a writing transaction "reserves" a place in the order of (writing) transactions before it knows whether its commit will succeed. This way, algorithms such as TL2 can be verified via a forward simulation to `ReservationTMS2`, reserving a transaction's place at the beginning of validation.

To prove that `ReservationTMS2` captures the TMS2 correctness condition, we show that it both implements and is implemented by `TMS2`. To do this, we introduce intermediate automata `TxnOrdTMS2` and `TxnOrdTMS2WithFailures`. `TxnOrdTMS2` is just like `TMS2` except that it records the initial state of the memory and a sequence of committed writer transactions (in the order that they commit) instead of the sequence of memory states that those transactions write. `TxnOrdTMS2WithFailures` is similar except that the sequence of transactions may include transactions that abort rather than commit. It is easy to verify that there are refinement mappings from `TxnOrdTMS2WithFailures` to `TxnOrdTMS2`, and from `TxnOrdTMS2` to `ReservationTMS2`, and a history mapping from `TMS2` to `TxnOrdTMS2`. A backward simulation is necessary only to show that `ReservationTMS2` implements `TxnOrdTMS2WithFailures`.

5 Our Experience

We have been verifying concurrent algorithms using PVS over several years, successively improving our framework, making it easier to construct, understand, and reuse proofs. There is undoubtedly still room for improvement. However, we have finally reached a point at which the machine-checked proofs we construct using our framework are often not significantly harder than rigorous hand proofs. In this section, we explain some details that have helped us to get to this point by improving both our productivity and the clarity of our proofs considerably. We also discuss ongoing issues with constructing formal, machine-checked proofs.

5.1 Reasons Why Proofs Are Easier Than Before

We are able to construct proofs more quickly and easily than before in part due to our increased facility with using PVS, particularly with its dependent type system and its inductive inference rules, in part due to improvements in our libraries defining the basic theory on automata and simulations, and in part due to our development of the libraries on basic data structures, particularly finite sequences. Although the concepts embodied in the `Automaton`, `View` and `AutomatonWithView` theories are essentially the same as those in the corresponding `Automata` theory of our earlier verification work, several factors have made our recent verifications significantly simpler.

First, changes in the way we represent sequences significantly simplified our proofs. In previous work, envisaging a framework that would evolve to also support progress proofs, we defined a type that could represent both finite and infinite sequences by using partial functions subject to a dependent typing condition to preclude "gaps" in the sequence. (PVS provides finite and infinite sequences, but not both in the same type.) While not conceptually difficult, the way PVS represents partial functions requires frequent conversions to distinguish values in the range of the function from "undefined"; this was a tedious and error-prone distraction in our previous work. It made proof sequents difficult to read, and generated many proof obligations due to type-checking conditions. It became clear that this was not worthwhile.

Thus far we have only done safety proofs for which finite sequences are sufficient. Therefore, our current framework uses only finite sequences, which has greatly simplified our proofs, both for writing and for reading, as well as allowing us to use the built-in definitions and lemmas in PVS. (Nonetheless, we did need to define some functionality on finite sequences, such as truncation, mapping a function over a sequence's elements, etc., as well as many lemmas to help us reason about sequences.) When we need sequences that can be either finite or infinite in future work, we plan to define a type whose elements can be either a finite sequence or an infinite sequence, using the built-in PVS theories for each, and to prove metatheorems to avoid duplication of proofs where possible.

Using the 'o' infix operator (defined in the PVS prelude of built-in theories) has also improved the readability of proof sequents.

Finally, PVS auto-rewrite rules can be included in a theory definition, so that they are automatically applied when the theory is imported, or they can be explicitly enabled in a proof script when needed. The latter option imposes more work on the user, but avoids spending time on applying the rules when they are not needed, and also prevents confusion that can arise when they are applied unexpectedly. We have chosen the latter option.

5.2 Using Our Framework to Verify the NOrec Algorithm

Unlike our previous verification efforts, we did not first write out a careful hand proof for NOrec and then attempt to translate it into PVS. Rather, we informally reasoned at a high level about why NOrec is correct, and tried to construct the formal proof guided by this informal reasoning and our past experience with simulation proofs. In particular, as described in the previous section, we defined an "intermediate" automaton that collapsed several steps of the NOrec algorithm into single atomic steps, and then successively refined that automaton until it had the granularity of the actual NOrec algorithm.

We were pleased to find that this approach worked quite well, and that using PVS to verify NOrec was not significantly more difficult than we estimate a similarly careful hand proof would have been. Indeed, in some respects, it was easier because when we discovered and corrected a mistake in a definition or lemma, we could rerun our earlier proofs and examine only those, if any, that no longer succeeded. Correcting those proofs was typically straightforward.

One exception was that, after proving that `NOrecAtomicCommitValidate` implements `TMS2`, we defined a variant that refines the validation operation but still treats commit as a single atomic operation, and proved that it implements `NOrecAtomicCommitValidate`. However, when we attempted to refine this automaton further so that the commit operation was no longer atomic, we found that it was difficult to prove that this automaton implemented the version with the atomic commit, and that it was easier to prove that it implemented `NOrecAtomicCommitValidate` directly. This problem was with our proof approach, and would have occurred in a hand proof as well.

Another exception was in the proof that `NOrecPaperPseudocode` implements `NOrec` (the abstract NOrec algorithm, in which some "local" actions happen atomically together with an action that accesses shared state), which was much more difficult than we expected: it required great care to correctly express the state correspondence (i.e., the forward simulation). Again, this problem would exist for a hand proof as well, but hand proofs are rarely done to that level of detail, and indeed, we had not initially intended to do so in our verification.

While working on our proofs, we discovered several small mistakes we had made in specifying the automata involved. Because we had done all the proofs with PVS, we could simply rerun them after fixing the mistakes, thereby identifying within minutes which proofs had been broken by the fixes. Of course we had to construct proofs to address the cases missed due to the fixed mistakes, but otherwise proofs were typically broken in straightforward and predictable ways, and could be quickly and easily repaired.

5.3 Formal Proofs Are Still Harder Than Typical Hand Proofs

Machine-checked PVS proofs are still harder to write than hand proofs. First, there is the difficulty of specifying automata and related properties in the PVS language. In a hand proof, we use whatever notation and mathematical definitions are most convenient. However, a formal language is more limited. For example, the `Automaton` theory (Figure 1) defines an execution fragment using a sequence of actions and a sequence of states that is exactly one longer than the sequence of actions, rather than as an alternating sequence of states and actions, which is more natural but would require a common supertype for actions and states that would pollute our proofs with many inconvenient conversions.

Second, in PVS, we need to prove that our definitions are type-correct, even for cases that are never used. For example, the `effect` function we use to determine the poststate of a transition must be well defined even when the precondition does not hold, even though its value in that case is unimportant. To address this, we define the poststate to be an arbitrary state in case the precondition does not hold, requiring extra steps in every proof that deals with the `effect` function. This issue is mitigated by the use of automated strategies. This kind of problem seems to be inherent in formal machine-checked proofs. Although annoying, such issues are usually manageable once one becomes familiar with PVS.

Third, "obvious" facts that would usually be used implicitly in a hand proof must be proved and cited. Associativity of concatenation is an example. Developing and verifying richer theories that assert these obvious facts and using auto-rewrite rules to avoid the need to cite them explicitly helps.

Fourth, we prove results about automata and simulations only when we need them. This disrupts our work when it happens, but will happen less as our framework matures. For example, in proving that `NOrecPaperPseudocode` implements `NOrec`, we needed an invariant of `NOrec` that would be somewhat involved to prove. However, we had already proved (an abstract version of) this invariant for `NOrecAtomicCommitValidate`, and refinements from `NOrec` to `NOrecDerived`, and from `NOrecDerived` to `NOrecAtomicCommitValidate`.

Rather than proving the invariant directly, we proved two new "metatheorems" for this purpose: one shows that the composition of two refinements is a refinement, and the other allows us to derive an invariant of one automaton from an invariant of another automaton and a refinement from the first automaton to the second. This approach has several advantages: (1) There is no need to replicate the proof. (2) The proof in the abstract automaton is simpler than the direct proof in `NOrec` would have been because the abstract automaton is simpler than `NOrec`. (3) We can use these metatheorems in future proofs.

6 Related Work

Cohen et al. [1] verified small instances of some simple TM algorithms directly using a model checker. This approach cannot verify larger instances, especially for more complex algorithms, and is limited to finite instances regardless. Others have attempted to overcome these limitations using more complex techniques.

Guerraoui et al. [7] showed that TM algorithms satisfying certain structural properties can be verifed by model checking small instances of them. To our knowledge, these structural properties have not been formally verified for any TM algorithm, so this work does not yield fully machine checked proofs. Emmi et al. [6] used techniques to automatically generate and check parameterized invariants. However, limitations of their approach forced them to use abstract models that assume away complex concurrency-related aspects of the practical TM algorithms considered. Overall, while model checking approaches can be valuable for testing hypotheses and finding bugs, we do not believe that they will be sufficient to fully verify practical TM algorithms any time soon.

Cohen et al. [2] used PVS to verify another simple TM algorithm. Like us, they used PVS to model algorithms and specifications, and used the PVS theorem prover to verify that a TM algorithm satisfies the specification. While this work is similar in spirit to ours, there are two notable differences. First, we have used correctness conditions that ensure aborted transactions cannot observe inconsistent behavior, which is critical in some contexts (see Section 2). Other than [7], all other work mentioned above use specifications that do not constrain the behavior of aborted transactions. Second, in contrast to the other work mentioned above, we have have modeled a practical TM algorithm in faithful detail, and have proved it correct in a hierarchical manner that can be leveraged to significantly reduce the effort required to verify other TM algorithms.

Finally, other frameworks exist for specifying and verifying relationships between I/O automata in PVS, analogous to the non-TM-specific foundations of our framework. To our knowledge, the most mature of these is TAME [13]. However TAME is not generally available, so we developed our own framework so that we could make it available for others to use and extend.

7 Concluding Remarks

We have built a framework for formally verifying transactional memory (TM) algorithms using the PVS theorem prover. To demonstrate its utility, we have used it to complete what we believe is the first fully formal, machine-checked correctness proof of a practical TM algorithm (NOrec). Our framework is available so that others may use and extend it, for example to verify other TM algorithms.

We continue to improve our framework, and we plan to extend it with proofs of additional TM algorithms. We are particularly interested in verifying an algorithm—such as TL2 [4]—that requires a backward simulation to prove that it implements TMS2. As discussed in Section 4.2, we expect to be able to prove that TL2 implements TMS2 by proving that it implements `ReservationTMS2`, thus avoiding the need for a backward simulation.

Acknowledgments. We thank Sam Owre for PVS assistance, Andy Lewis for machines, and Simon Doherty for contributions to earlier versions of our framework.

References

1. Cohen, A., O'Leary, J., Pnueli, A., Tuttle, M., Zuck, L.: Verifying correctness of transactional memories. In: FMCAD 2007: Proceedings of Formal Methods in Computer Aided Design, pp. 37–44 (2007)
2. Cohen, A., Pnueli, A., Zuck, L.D.: Mechanical Verification of Transactional Memories with Non-transactional Memory Accesses. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 121–134. Springer, Heidelberg (2008)
3. Dalessandro, L., Spear, M., Scott, M.: NOrec: Streamlining STM by abolishing ownership records. In: PPoPP 2010: Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (January 2010)
4. Dice, D., Shalev, O., Shavit, N.: Transactional locking II. In: International Symposium on Distributed Computing, pp. 194–208 (2006)
5. Doherty, S., Groves, L., Luchangco, V., Moir, M.: Towards formally specifying and verifying transactional memory. Formal Aspects of Computing (2012), `http://labs.oracle.com/projects/scalable/pubs/Doherty-FAC-2012.pdf`
6. Emmi, M., Majumdar, R., Manevich, R.: Parameterized verification of transactional memories. In: Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, pp. 134–145. ACM, New York (2010)
7. Guerraoui, R., Henzinger, T., Jobstmann, B., Singh, V.: Model checking transactional memories. In: PLDI 2008: Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 372–382 (2008)
8. Guerraoui, R., Kapalka, M.: On the correctness of transactional memory. In: PPoPP 2008: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 175–184 (2008)
9. Herlihy, M., Moss, J.E.B.: Transactional memory: Architectural support for lock-free data structures. In: Proceedings of the 20th Annual International Symposium on Computer Architecture (1993)
10. Lesani, M., Luchangco, V., Moir, M.: Putting opacity in its place (May 2012), `http://labs.oracle.com/projects/scalable/pubs/OpacityInPlace.pdf`
11. Lynch, N., Tuttle, M.: Hierarchical correctness proofs for distributed algorithms. In: Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing, pp. 137–151 (August 1987)
12. Lynch, N., Vaandrager, F.: Forward and backward simulations, I: Untimed systems. Information and Computation 121(2), 214–233 (1995)
13. Mitra, S., Archer, M.: PVS strategies for proving abstraction properties of automata. Electron. Notes Theor. Comput. Sci. 125(2), 45–65 (2005)
14. Owre, S., Shankar, N., Rushby, J.: PVS: A Prototype Verification System. In: Kapur, D. (ed.) CADE 1992. LNCS, vol. 607, pp. 748–752. Springer, Heidelberg (1992)
15. Papadimitriou, C.: The serializability of concurrent database updates. J. ACM 26, 631–653 (1979)
16. The PVS Specification and Verification System, `http://pvs.csl.sri.com/`