Verification of Transactional Memories that Support Non-Transactional Memory Accesses

Ariel Cohen
New York University
arielc@cs.nyu.edu

Amir Pnueli
New York University
amir@cs.nyu.edu

Lenore D. Zuck
University of Illinois at Chicago
lenore@cs.uic.edu

Abstract

A major challenge of Transactional memory implementations is dealing with memory accesses that occur outside of transactions. In previous work we showed how to specify transactional memory in terms of admissible interchanges of transaction operations, and gave proof rules for showing that an implementation satisfies its specification. However, we did not capture non-transactional memory accesses. In this work we show how to extend our previous model to handle non-transactional accesses. We apply our proof rules using a PVS-based theorem prover and produce a machine checkable, deductive proof for the correctness of implementations of transactional memory systems that handle non-transactional memory accesses.

Keywords Verification, Transactional Memory, Non-Transactional Accesses

1. Introduction

Transactional Memory (4) is a simple solution for coordinating and synchronizing concurrent threads that access the same memory locations. It transfers the burden of concurrency management from the programmers to the system designers and enables a safe composition of scalable applications. Multicore, which requires concurrent programs in order to gain a full advantage of the multiple number of processors, have became the mainstream architecture for microprocessor chips and thus many new transactional memory implementations have been proposed recently.

A transaction is a sequence of operations that are executed atomically – either all completely successfully (and the transaction commits), or none complete (and the transaction aborts). In addition, transactions also run in isolation – each transaction appears as if running alone on the system. Under strong isolation, isolation is guaranteed not only between transactions but also between transaction and non-transactional memory accesses. Under weak isolation, isolation is guaranteed only among transactions. Transactional memory allows transactions to run concurrently as long as the atomicity and isolation of each transaction are preserved. One of the major challenges of strong isolated implementations is dealing with memory accesses that occur outside of transactions.

In (3) we show how to specify transactional memory in terms of admissible interchange of transaction operations. Roughly speaking, we define a generic specification, where (transactional) operations are pushed into a queue-like structure. Operations in the queue are interchanged according to a fixed set of rules that depends on the correctness requirements (e.g., conflict resolution policy), expressed as admissible interchanges operations. When a complete transaction is in the head of the queue, it’s being popped. If every transaction eventually commits or aborts, then the sequence of popped committed transactions forms a possible atomic transaction sequence that is, in some precise sense, equivalent to the original sequence of operations.

There (3), we also give proof rules for showing that an implementation satisfies such a specification. We also use the model checker TLC (7) to verify several types of transactional memory implementations.

In this paper we add non-transactional accesses into the model. In order to accommodate the non-transactional access, we first transform them into singleton committed transactions. Then, it is the queue-like structure in our model that guarantees that the sequence of operation is correct.

While the proof rules for verifying that an implementation satisfies a specification (with an admissible interchange set) remain intact when adding non-transactional access expressed as singleton committed transaction, applying it to particular implementations is much harder. Moreover, proofs in TLC are restricted to small instantiations (of number of clients, number of events in a transaction, etc.) and cannot generalize. Adding the non-transactional access to this mix, it is virtually impossible to gain any assurance from successful TLC runs for this model.

TLPVS (11) is a system that embeds temporal logic and its deductive frame-work within the theorem prover PVS (10). It includes a set of theories for defining temporal logic, proof rules for proving soundness and response properties, and strategies which aid in conducting the proofs. Moreover, it has a special framework for verifying unbounded systems and theories, and unlike model checking it does not suffer from state-explosion. It also can be used to create general relations, rather than strict mappings, between states. The downside, of course, is that like any other similar of its ilk, some user intervention and expertise is required.

Here, we demonstrate how to use TLPVS to verify the correctness of a simple transactional memory implementation that handles non-transactional memory accesses. We also give a broad ground-work and a set of theories that can be used for future proofs of more involved implementations. To the best of our knowledge, the work presented here is the first to formally verify correctness of transactional memories that handle non-transactional memory accesses.

The rest of the paper is organized as follows: Section 2 presents the formal model of observable parameterized fair systems, provides preliminary definitions related to transactional memory, and
defines the concept of admissible interchanges. Section 3 provides a
specification model of a transactional memory. Section 4 discusses a
proof rule for verifying implementations. Section 5 presents a
simple implementation of transactional memory that handles non-
transactional memory accesses. Section 6 shows how to apply de-
ductive verification using TLPVS to verify this implementation.
Section 7 discusses related work, and Section 8 provides some con-
cclusions and open problems.

2. Background

2.1 Observable Parameterized Fair Systems

As a computational model we use observable parameterized fair
systems (OPFS). This extends the model parameterized fair systems
(PFS) (11) with an observation domain. PFS is a variation of FDS
which in turn is a slight variation of the model fair transition system
(8). Under OPFS, a system \( D : (\Sigma, d_C, O, \Theta, \rho, F, J, C) \) consists of
the following components:

- \( \Sigma \) — A non-empty set of system states. Typically, a state
  is structured as a record whose fields are typed system variables,
  \( V \). For a state \( s \) and a variable \( v \in V, s[v] \) denotes the value
  assigned to \( v \) by state \( s \).
- \( d_C \) — A non-empty observation domain which is used for
  observing the external behavior of the states.
- \( O \) — An observation function. This is a mapping from \( \Sigma \) to
  \( d_C \).
- \( \Theta \) — An initial condition: A predicate characterizing the initial
  states.
- \( \rho(s, s') \) — A transition relation: A bi-predicate, relating the
  state \( s \) to state \( s' \) — a \( D \)-successor of \( s \). We assume that every
  state has a \( D \)-successor.
- \( F \) — A non-empty fairness domain which is used for paramete-
  rizing the justice and compassion fairness requirements.
- \( J \) — A mapping from \( F \) to predicates. For each \( f \in F \),
  \( J(f) \) is a justice (weak fairness) requirement (predicate). For each
  \( f \in F \), a computation must include infinitely many states
  satisfying \( J(f) \).
- \( C \) — A mapping from \( F \) to pairs of predicates, called a com-
  passion (strong fairness) requirement. For each \( f \in F \), let
  \( C(f) = (p, q) \) be the compassion requirement corresponding to
  \( f \). It is required that if a computation contains infinitely many
  \( p \)-states, then it should also contain infinitely many \( q \)-states.

A run of an OPFS \( D \) is an infinite sequence of states \( \sigma : s_0, s_1, \ldots \)

- Initiality — \( s_0 \) is initial, i.e., \( s_0 \models \Theta \).
- Consecution — For each \( \ell = 0, 1, \ldots \), the state \( s_{\ell+1} \) is a \( D \)-
  successor of \( s_\ell \). That is \( \rho(s_\ell, s_{\ell+1}) = \top \).

A computation of \( D \) is a run that satisfies

- Justice — for every \( f \in F \), \( \sigma \) contains infinitely many occu-
  rrences of \( J(f) \)-states.
- Compassion — for every \( f \in F \) such that \( C(f) = (p, q) \in C \),
  either \( \sigma \) contains only finitely many occurrences of \( p \)-states, or
  \( \sigma \) contains infinitely many occurrences of \( q \)-states.

2.2 Transactional Sequences

Assume \( n \) clients that direct requests to a memory system, denoted
by memory. For every client \( p \), let the set of non-transactional
invocations by client \( p \) consists of:

- \( iR^t_p(x) \) — A non-transactional request to read from address
  \( x \in \mathbb{N} \).
- \( iW^t_p(y, v) \) — A non-transactional request to write the value
  \( v \in \mathbb{N} \) to address \( y \in \mathbb{N} \).

Let the set of transactional invocations by client \( p \) consists of:

- \( t \) — An open transaction request.
- \( tR^t_p(x) \) — A transactional read request from address \( x \in \mathbb{N} \).
- \( iW^t_p(y, v) \) — A transactional request to write the value \( v \in \mathbb{N} \) to
  address \( y \in \mathbb{N} \).
- \( t \) — A commit transaction request.
- \( t \) — An abort transaction request.

The memory provides a response for each invocation. Erro-
neous invocations (e.g., a \( t \) while client \( p \) has a pending transaction)
are responded by the memory returning an error flag \( err \). Non-
erroneous invocations, except for \( iR^t \) and \( iW^t \) are responded by the
memory returning an acknowledgment \( ack \). Finally, for non-
erroneous \( iR^t_p(x) \) and \( iW^t_p(x, v) \) the memory returns the (nat-
ural) value of the memory at location \( x \). Let \( p_0 \) denote the set of re-
ponses to client \( p \). We assume that invocations and responses oc-
cur atomically and consecutively, i.e., there are no other operation
that interleave an invocation and its response.

Let \( E^o_p : \{ R^o_p(x, u), W^o_p(x, v) \} \) be the set of non-
transactional observable events, \( E^t_p : \{ R^t_p(x, u), W^t_p(x, v), t \} \) be the set of trans-
tactional observable events and \( E_p = E^o \uplus E^t \), all asso-
ciated with client \( p \). We consider as observable only requests that
are accepted, and abbreviate the pair (invitation, non-err response)
by omitting the \( i \)-prefix of the invocation. Thus, \( W^o_p(x, v) \) abbrevi-
ates \( iW^o_p(x, v) \), \( ack \). For read actions, we include the value read,
that is, \( R^o_p(x, u) \) abbreviates \( iR^o_p(x) \), \( p \). When the value writ-
ten/read of no relevance, we write the above as \( W^o_p(x) \) and \( R^o_p(x) \).
When both values and addresses are of no importance, we omit
the addresses, thus obtaining \( W^p_p(x, v) \) and \( R^p_p(x) \) (symmetric abbreviations
and shortcuts are used for the non-transactional observable events).

The output of each action is its relevant observable event when the
invocation is accepted and undefined otherwise. Let \( E \) be the set of
all observable events over all clients, i.e., \( E = \bigcup_{p=m} E_p \) (similarly
define \( E^o \) and \( E^t \) to be the sets of all non-transactional and all trans-
tactional observable events, respectively).

Let \( \sigma : e_0, e_1, \ldots, e_n \) be a finite sequence of observable \( E \)-
events. Let \( \sigma \) be its corresponding transactional sequence of ob-
vitable \( E^t \)-events obtained from \( \sigma \) by replacing each \( R^t_p(x, v) \)
and \( W^t_p(x, v) \) by \( \{ R^o_p(x, v) \} \) and \( \{ W^o_p(x, v) \} \), respectively.

The transactional sequence \( \sigma \) is called a well-formed transactional sequence
(TS for short) if the following conditions hold:

1. For every client \( p \), let \( \sigma |_p \) be the sequence obtained by pro-
jecting \( \sigma \) onto \( E^o_p \). Then \( \sigma |_p \) satisfies the regular expression \( T^p \),
where \( T^p \) is the regular expression \( \{ R^o_p(x, v) \} \) and \( \{ W^o_p(x, v) \} \).

2. The sequence \( \sigma \) is locally read-write consistent: namely, for any
subsequence \( W^o_p(x, v) R^o_p(x, u) \) in \( \sigma \), where \( \eta \) contains
no event of the form \( \{ W^p_p(x, v), \) or \( \{ R^p_p(x, u) \}, we have \( u = v \).

The sequence \( \sigma \) is called a well-formed sequence (TS for short) if
its corresponding transactional sequence \( \sigma \) is well-formed trans-
actional sequence. We denote by \( S \) the set of all well-formed se-
quences, and by \( T \) the set of all well-formed transactional se-
quences. We respectively denote by \( \text{pref}(S) \) and \( \text{pref}(T) \) the sets of prefixes of such sequences.

Notice that the requirement of local read-write consistency can be enforced by each client locally. To build on this observation, we assume that, within a single transaction, there is no \( R^e_i(x) \) following a \( W^e_i(x) \), and there are no two reads or two writes to the same address. With these assumptions, the requirement of local read-write consistency is always (vacuously) satisfied.

The TS \( \dot{\sigma} \) is called atomic if:

1. It satisfies the regular expression \((T_1 + \cdots + T_n)^*\). That is, there is no overlap between any two transactions.

2. The sequence \( \dot{\sigma} \) is globally read-write consistent: namely, for any subsequence \( W^e_i(x, v) R^e_i(x, u) \) in \( \dot{\sigma} \), where \( \eta \) contains \( \succ_p \), which is not preceded by \( \succ_p \), and contains no event \( W^e_i(x) \) followed by an event \( \succ_p \), it is the case that \( u = v \).

### 2.3 Characterizations of Transactional Memory

Transactional memory implementations are classified using a number of parameters. Some of the variations may not be applied when the transactional memory supports non-transactional memory accesses.

An implementation has an eager version management if it updates the memory immediately when a transactional write occurs. It has a lazy version management if it updates the memory when the transaction commits. Under lazy version management responses to transactional read operations must be re-validated upon a commit invocation, and aborts require no further ado. Under eager version management aborts may require rolling-back the memory to hold its old values. This implies that such implementations can not handle non-transactional operations, since a non-transactional read may obtain a value which is written by a transaction that is later aborted, thus the value is not valid since the memory is rolled-back to hold old values.

A conflict occurs when two overlapping transactions (each begins before the other ends), or a non-transactional operation and an overlapping transaction, access the same location and at least one writes to it. For the latter case, it is always the transaction that is aborted. An implementation has an eager conflict detection if it detects conflicts as soon as they occur, and it has a lazy conflict detection if it delays the detection until a transaction requests to commit. When the conflict occurs between two transactions, eager conflict detection helps to avoid worthless work by a transaction that is eventually aborted. Yet, deciding to keep one transaction (and to cause the other to abort) does not guarantee that the “surviving” transaction can commit since it may conflict with a third transaction. Consequently, lazy conflict detection may allow doomed transactions to perform worthless work.

### 2.4 Interchanging Events

We use strict serializability as the correctness condition for transactional memory. Intuitively, \( \sigma \) is correct if its corresponding \( \dot{\sigma} \) can be transformed into an atomic TS by first removing from it all events that belong to aborted transactions, then interchanging adjacent events that belong to committed transactions but keeping the transactions in the order in which they were committed.

We will describe conflicts by restricting which events can be exchanged during serialization. Let \( \dot{\sigma} \) be a well-formed TS. For an event \( e \in \dot{\sigma} \), let \( T(e) \) be the transaction to which \( e \) belongs, and let \( k(e) \) be such that \( \dot{\sigma}[k(e) = e] \). Suppose that \( T_i \) is a committed transaction. Thus, for every \( e \in T_i \), \( k(e) \geq 1 \), we may need to determine whether \( e \) can be interchanged with \( e(k(e) - 1) \). Let \( \beta(\dot{\sigma}, e) \) be the projection of \( \dot{\sigma} \) onto the events of \( T_i \) that \( T(\dot{\sigma}(k(e) - 1)) \) until the commit event of \( T_i \). Note that \( \beta(\dot{\sigma}, e) \) is a finite sequence of events over \( T(e) \) and \( T(\dot{\sigma}(k(e) - 1)) \).

A forbidding formula is an LTL formula that uses no future operators. Such formulas are interpreted at given points in a computation and reason only on the prefix of computations leading to them. Let \( \tilde{\sigma} : e_1, e_2, \ldots \) be a well-formed transactional sequence, and assume that \( e = e_{k(\sigma)} \) (where \( k \geq 2 \)). Then \( e_{k-1} \) and \( e_{k} \) are forbidden from being interchanged w.r.t. the forbidding formula \( \phi \) if \( \{\beta(\tilde{\sigma}, e), k(\tilde{\sigma}(e_{k-1}))\} = \emptyset \). That is, we look at the subsequence of \( \tilde{\sigma} \) that consists of the events of the transactions that \( e \) and its predecessor belong to, up to the closing event of \( T(e) \), and check whether \( \phi \) is satisfied at the position where the original \( e \) is relative to the subsequence.

In (12), Scott describes six classes of conflicts. Our definition is not equivalent to his, which, in some cases, may imply interchanges that are not admissible. We next provide Scott’s classes of conflicts and illustrate how the definitions for disallowing them are expressed using forbidding formulas. Note that \( e_i \prec e_j \) denotes that \( e_i \) precedes \( e_j \) and \( \land \) and \( \lor \) are the temporal operators for “previously” (in the previous state) and “sometimes in the past,” respectively.

1. An overlap conflict occurs if for some transactions \( T_i \) and \( T_j \), we have \( e_i \prec e_j \) and \( e_j \prec e_i \). The corresponding forbidden formula \( \phi_o \) is

\[
\bigl( \land \bigl( x \land \land J_i \bigl) \lor \bigl( J_i \land \land x \land \land W_j(x) \bigl)
\]

2. A writer overlap conflict occurs if two transactions overlap and one performs a write before the other terminates, i.e., for some \( T_i \) and \( T_j \), we have \( e_i \prec W_j \prec e_j \) or \( W_j \prec e_i \prec e_j \). The corresponding forbidden formula \( \phi_{wo} \) is

\[
\bigl( \land \bigl( \land \bigl( x \land \land J_i \bigl) \lor \bigl( J_i \land \land x \land \land W_j(x) \bigl) \bigl)
\]

3. A lazy invalidation conflict occurs if the commitment of one transaction may invalidate a read of the other, i.e., if for some transaction \( T_i \) and \( T_j \) and some memory address \( x \), we have \( R_i(x), W_j(x) \prec e_j \prec e_i \). The corresponding forbidden formula \( \phi_{li} \) is

\[
\bigl( \land \bigl( \land \bigl( x \land \land J_i \bigl) \lor \bigl( J_i \land \land x \land \land W_j(x) \bigl) \bigl)
\]

4. An eager W-R conflict occurs if for some transactions \( T_i \) and \( T_j \) a lazy invalidation conflict occurs, or if for some memory address \( x \), we have \( W_i(x) \prec R_j(x) \prec e_i \). The corresponding forbidden formula \( \phi_{ewr} \) is

\[
\phi_{ewr} \lor \bigl( \land \bigl( x \land \land J_i \bigl) \lor \bigl( J_i \land \land x \land \land W_j(x) \bigl) \bigl)
\]

5. A mixed invalidation conflict occurs if for some transactions \( T_i \) and \( T_j \) a lazy invalidation conflict occurs, or if for some memory address \( x \), we have \( R_i(x) \prec W_j(x) \prec e_i \). There is no corresponding forbidden formula for this conflict.

6. A n eager invalidation conflict occurs if for some transactions \( T_i \) and \( T_j \) an eager W-R conflict occurs, or if for some memory address \( x \), we have \( R_i(x) \prec W_j(x) \prec e_i \). The corresponding forbidden formula \( \phi_{ews} \) is

\[
\phi_{ews} \lor \bigl( \land \bigl( x \land \land J_i \bigl) \lor \bigl( J_i \land \land x \land \land W_j(x) \bigl) \bigl)
\]

Given a conflict \( c \) and the forbidden formula \( \phi_c \) that corresponds to it, a TS \( \dot{\sigma} \) is said to be serializable with respect to \( \phi_c \), if \( \phi_c \) does not hold at any point of \( \dot{\sigma} \).

The sequence \( \dot{\sigma} \) is called the purified version of TS \( \sigma \) if \( \dot{\sigma} \) is obtained by removing from \( \dot{\sigma} \) all aborted transactions, i.e., removing the opening and closing events for such a transaction and all the read-write events by the same client that occurred between the opening and closing events. When we specify the correctness of a transactional memory implementation, only the purified versions of
the implementation’s transaction sequences will have to be serializable.

3. Specification and Implementation

Let \( \phi \) be a forbidden formula which we fix for the remainder of this section. We now describe \( \text{Spec}_\phi \), a specification of transactional memory that generates all SSs that their corresponding TSs are serializable with respect to \( \phi \) and a definition of a correct implementation of \( \text{Spec}_\phi \).

The specification \( \text{Spec}_\phi \) uses the following data structures:

- \( \text{spec mem}: \mathbb{N} \rightarrow \mathbb{N} \) — A persistent memory, represented as an array of naturals. For simplicity, we represent it as an infinite array. Initially, for every \( i \geq 0 \), \( \text{spec mem}[i] = 0 \);
- \( Q \): queue of \( E \) \( \cup \bigcup_{j=1}^{n} \{ \text{mark}_p \} \) — A queue of pending transactional events, initially empty;
- \( \text{spec out} \): scalar in \( E \) \( \cup \{ \bot \} \) — an output variable recording responses to clients, initially \( \bot \);
- \( \text{spec doomed} \): array \( [1..n] \) of booleans — An array recording which transactions are doomed to be aborted. Initially \( \text{spec doomed}[p] = F \) for every \( p \).

Consider the following transaction, \( T_r \):

\[
\begin{align*}
\text{mark}_p & \rightarrow R_p^1(x_1, u_1), \ldots, R_p^{m}(x, u), W_p^1(y_1, v_1), \ldots, W_p^n(y, w) \\
\end{align*}
\]

We say that \( T_r \) is consistent with \( \text{spec mem} \) if, for each \( j \in [1..r] \), \( \text{spec mem}[x_j] = u_j \). The update of \( \text{spec mem} \) by \( T_r \) is defined to be the memory \( \text{spec mem}' \) such that, for each \( j \in [1..r] \), \( \text{spec mem}'[y_j] = v_j \) and, for all \( k \notin \{ y_1, \ldots, y_r \} \), \( \text{spec mem}'[k] = \text{spec mem}[k] \).

Intuitively, the stream of \( \text{spec out} \)'s is the sequence of observable events. Pending transactions are partitioned into two categories. Transaction \( T_p \) is pending-active, if it has at least one event in \( Q \), it is not committed nor aborted (does not terminate with \( \text{mark}_p \rightarrow W_p \) in \( Q \)) and \( \text{spec doomed}[p] = F \). Its events are maintained in \( Q \) in the order they are issued. Transaction \( T_p \) is pending-doomed if it has no events in \( Q \) and \( \text{spec doomed}[p] = T \). A client is pending if it has a pending-active or a pending-doomed transaction. A client is not-pending if it is not pending and \( \text{spec doomed}[p] = F \). Note however, that a not-pending client is allowed to have a committed transaction in \( Q \) (aborted transactions are removed from \( Q \) as soon as they get aborted).

For every pending-active transaction \( T_p \), we allow the queue \( Q \) to include a special symbol, \( \text{mark}_p \). The symbol \( \text{mark}_p \) is added to the queue just before \( T_p \) issues a commit request, and some tests are done to determine whether it can safely commit. If the tests are successful, \( \text{spec out} \) is set to \( \text{mark}_p \) and then \( \text{mark}_p \) as well as all the \( E_p \)'s events are removed from the queue. We say that \( Q \) is marked (unmarked) if it has some (no) \( \text{mark}_p \) symbol.

Non-transactional read and write events are modeled by committed singleton transactions, i.e., by transactions satisfying the regular expression \( \text{mark}_p \rightarrow (R_p^0 + W_p^0) \rightarrow \text{mark}_p \). Their events are added to \( Q \) consecutively in one step. Forming these special singleton transactions, for non-transactional operations, allows using \( Q \) exactly as defined in (3) since it continues to be over \( E \). Furthermore, it allows using the same forbidden formulas, which were design to prohibit conflicts between transactions, to prevent the same conflicts between transactions and non-transactional operations. Since non-transactional operations cannot be aborted, their corresponding singleton transactions are always committed (have \( \text{mark}_p \) already when added to \( Q \)), and therefore when conflicting with other transactions, the latter are always those that are doomed to be aborted. We only allow not-pending clients to issue non-transactional operations.

Actions \( a_1 \)–\( a_7 \) are applicable only when \( Q \) is unmarked. \( a_1 \) and \( a_2 \) correspond to non-transactional read and write.

\( a_1 \). For some \( p \in [1..n] \), and \( x, u \in \mathbb{N} \), if client \( p \) is not-pending, write \( R_p^{m}(x, u) \) to \( \text{spec out} \) and append \( \text{mark}_p \rightarrow R_p^{m}(x, u) \rightarrow \text{mark}_p \) to the end of \( Q \).

\( a_2 \). For some \( p \in [1..n] \), and \( x, u \in \mathbb{N} \), if client \( p \) is not-pending, write \( W_p^{m}(x, u) \) to \( \text{spec out} \) and append \( \text{mark}_p \rightarrow W_p^{m}(x, u) \rightarrow \text{mark}_p \) to the end of \( Q \).

Actions \( a_3 \)–\( a_7 \) correspond to transactional operations. Note that \( a_6 \) and \( a_7 \) do not set \( \text{spec out} \) to a value. For such cases we assume that \( \text{spec out} \) is set to \( \bot \).

\( a_3 \). For some \( p \in [1..n] \), if client \( p \) is not-pending, write \( \text{mark}_p \rightarrow \text{spec out} \) and append it to the end of the queue \( Q \).

\( a_4 \). For some \( p \in [1..n] \), and \( x, u \in \mathbb{N} \), if client \( p \) is pending (has a pending-active or pending-doomed \( T_p \)), then write \( W_p^{m}(x, u) \) to \( \text{spec out} \). If \( T_p \) is pending-active also append it to the end of the queue \( Q \).

\( a_5 \). For some \( p \in [1..n] \), and \( x, u \in \mathbb{N} \), if client \( p \) is pending, then write \( R_p^{m}(x, u) \) to \( \text{spec out} \). If \( T_p \) is pending-active also append it to \( Q \). We also require that the events of \( T_p \) are locally consistent.

\( a_6 \). For some \( p \in [1..n] \) such that \( T_p \) is pending-active, remove all of \( T_p \)'s events from \( Q \) and set \( \text{spec doomed}[p] \) to \( T \).

\( a_7 \). For some \( p \in [1..n] \) such that \( T_p \) is pending-active, add \( \text{mark}_p \) to the end of \( Q \).

Actions \( a_8 \)–\( a_{11} \) deal with commits and aborts. It is \( a_9 \) that determines whether a transaction marked for commit can indeed commit.

\( a_8 \). For some \( p \in [1..n] \) such that client \( p \) is pending, write \( \text{mark}_p \rightarrow \text{spec out} \), and remove all of \( T_p \)'s- and \( \text{mark}_p \)-events from \( Q \), and set \( \text{spec doomed}[p] \) to \( F \).

\( a_9 \). For some \( p \in [1..n] \) such that \( T_p \) is pending-active, if \( T_p \) is consistent with \( \text{spec mem} \), all of its events appear consecutively in the front of \( Q \), and \( \text{mark}_p \) is in \( Q \), then write \( \text{mark}_p \rightarrow \text{spec out} \) and replace \( \text{mark}_p \) with \( \text{mark}_p \).

\( a_{10} \). For some \( p \in [1..n] \), if \( T_p \) is consistent with \( \text{spec mem} \), all of its events appear consecutively in the front of \( Q \) and it terminates with \( \text{mark}_p \) (i.e. committed), update \( \text{spec mem} \) according to \( T_p \), and remove all \( E_p \)'s events from \( Q \).

\( a_{11} \). Interchange the order of two contiguous events \( e_p, e_q \) in \( Q \) belonging to different transactions \( T_p, T_q \), respectively, if \( e_p, e_q \) are not compatible with \( \phi \). We treat \( e_p, e_q \) as if it is a \( \text{mark}_p \).

\( a_{12} \). An idling transition which does not modify \( \text{spec mem} \), \( Q \) or \( \text{spec doomed} \).

Note that the updates of the queue in \( a_8 \) and \( a_{11} \) are not standard queue operations.

The specification has \( n \) associated justice requirements, namely, for every \( p = 1, \ldots, n \):

there are infinitely many states in which \( Q[p] \) is empty.

A sequence \( \sigma \) over \( E \) is compatible with \( \text{Spec}_\phi \) if it can be obtained by the sequence of \( \text{spec out} \) that \( \text{Spec}_\phi \) outputs once all the \( \bot \)'s are removed. We then have:

CLAIM 1. For every sequence \( \sigma \) over \( E \), \( \sigma \) is compatible with \( \text{Spec}_\phi \) iff its corresponding transactional sequence \( \hat{\sigma} \) is serializable with respect to \( \phi \).
An implementation $TM$: $(read, commit)$ of a transactional memory consists of a pair of functions

$$read: \text{ pref}(S) \times [1..n] \times N \rightarrow N \quad \text{and}$$

$$commit: \text{ pref}(S) \times [1..n] \rightarrow \{ack, err\}$$

For a prefix $\sigma$ of a $S$, $read(\sigma, p, x)$ is the response (value) of the memory to an accepted $\iota R^n_{\text{opf}}(x)$ request immediately following $\sigma$, and $commit(\sigma, p)$ is the response (ack or err) of the memory to a $\iota \triangleright_p$ request immediately following $\sigma$.

An $S \sigma \in S$ is said to be compatible with the memory $TM$ if:

1. For every prefix $\eta R^n_{\text{opf}}(x, u)$ of $\sigma$, $read(\eta, p, x) = u$.
2. For every prefix $\eta R^n_{\text{opf}}(x, u)$ of $\sigma$, $read(\eta, p, x) = u$.
3. For every prefix $\eta \triangleright_p$ of $\sigma$, $commit(\eta, p) = ack$.

An implementation $TM$: $(read, commit)$ is a correct implementation of a transactional memory with respect to $\phi$ if every $S$ compatible with $TM$ is also compatible with $Spec$.

4. Verifying Implementation Correctness

In this section we present a proof rule for verifying that an implementation satisfies the specification $Spec$. The approach is an adapted version of the rule provided in (5), which was based on the abstract mapping of (1), and was given in (3).

To apply the underlying theory, we assume that both the implementation and the specifications are represented as OPFS.

In the current application, we prefer to adopt an event-based view of reactive systems, by which the observed behavior of a system is a (potentially infinite) set of events. Technically, this implies that one of the system variables $O$ is designated as an output variable.

The observation function is then defined by $O(s) = s[O]$. It is also required that the observation domain always includes the value $\bot$, implying no observable event. In our case, the observation domain of the output variable is $E_\bot = E \cup \{\bot\}$.

Let $\eta : e_0, e_1, \ldots$ be an infinite sequence of $E_\bot$-values. The $E_\bot$-sequence $\eta$ is called a stuttering variant of the sequence $\eta$ if it can be obtained by removing or inserting finite strings of the form $\bot, \ldots, \bot$ at (potentially infinitely) many different positions within $\eta$.

Let $\sigma : s_0, s_1, \ldots$ be a computation of OPFS $D$. The observation $\text{output} \sigma$ of $\sigma$ is the $E_\bot$ sequence $s_0[O], s_1[O], \ldots$ obtained by listing the values of the output variable $O$ in each of the states. We denote by $\text{Obs}(D)$ the set of all observations of system $D$.

Let $D_C : (\Sigma_C, E_C, \Theta_C, \rho_C, \mathcal{F}_C, J_C)$ be a concrete system, and let $D_A : (\Sigma_A, E_A, \Theta_A, \rho_A, \mathcal{F}_A, J_A)$ be an abstract system. For simplicity, we assume that neither system contains compression requirements. Note that we assume that both systems share the same observation domain $E_\bot$.

We say that system $D_A$ abstracts system $D_C$ (equivalently, $D_C$ refines $D_A$), denoted $D_C \subseteq D_A$ if, for every observation $\eta \in \text{Obs}(D_C)$, there exists $\hat{\eta} \in \text{Obs}(D_A)$, such that $\eta$ is a stuttering variant of $\hat{\eta}$. In other words, modulo stuttering, $\text{Obs}(D_C)$ is a subset of $\text{Obs}(D_A)$. We denote by $s$ and $S$ the states of $D_C$ and $D_A$, respectively.

Rule $\text{ABS-REL}$ in Fig. 1 is a proof rule to establish that $D_C \subseteq D_A$. The method advocated by the rule assumes the identification of an abstraction relation $R(s, S)$ between the concrete and abstract states. If the relation $R(s, S)$ holds, we say that the abstract state $S$ is an $R$-image of the concrete state $s$.

Premise R1 of the rule states that for every initial concrete state $s$, it is possible to find an initial abstract state $S \models \Theta_A$, such that $R(s, S) = T$.

Premise R2 states that for every pair of concrete states, $s$ and $s'$, such that $s'$ is a $\rho_A$-successor of $s$, and an abstract state $S$ which is a $R$-related to $s$, there exists an abstract state $S'$ such that $S'$ is $R$-related to $s'$ and is also a $\rho_A$-successor of $S$. Together, R1 and R2 guarantee that, for every run $s_0, s_1, \ldots$ of $D_A$, there exists a run $S_0, S_1, \ldots$ of $D_A$, such that for every $j \geq 0$, $S_j$ is $R$-related to $s_j$.

Premise R3 states that if abstract state $S$ is $R$-related to the concrete state $s$, then the two states agree on the values of their observables.

Together with the previous premises, we conclude that for every $s \in D_C$, there exists a corresponding run of $D_A$, which has the same observation as $s$. Premise R4 ensures that the abstract justice requirements hold in any abstract state sequence which is a (point-wise) $R$-related to a concrete computation. Here, $\square$ is the (linear time) temporal operator for “henceforth”, $\Diamond$ the temporal operator for “eventually”, thus, $\square \Diamond$ means “infinitely often.” It follows that every sequence of abstract states which is $R$-related to a concrete computation $\sigma$ and is obtained by applications of premises R1 and R2 is an abstract computation whose observables match the observables of $\sigma$. This leads to the following claim:

**Claim 2.** If the premises of rule $\text{ABS-REL}$ are valid for some choice of $R$, then $D_A$ is an abstraction of $D_C$.

5. Trivial Implementation

We now demonstrate how our framework can be used to verify a trivial transactional memory implementation that handles non-transactional memory accesses and employs lazy version management. In this implementation transactions execute speculatively in the clients’ caches and once a transaction commits, all open transactions that contain some read events from addresses written to by the committed transaction are “doomed.” In case of non-transactional write, all open transactions that read from the same location are also “doomed.” A doomed transactions may execute new read and write events in its cache, but it must eventually abort. Note that non-transactional memory accesses may interfere with transactions of other clients, but not during atomic operation (e.g. not during a commit).

The forbidden conflict that may be identified with this implementation is lazy invalidation. As explained in Subsection 2.4 this conflict occurs when the commitment of one transaction may invalidate a read of another, or when a non-transactional write invalidates a read of an overlapping transaction. Thus the corresponding forbidden formula of the specification is $\phi_{\text{iva}}$, as provided in Subsection 2.4.

We next give a simple description of the implementation, and later in Section 6 explain how we can verify that it refines its specification using the proof rule $\text{ABS-REL}$. We refer to the implementation as $TM$. It uses the following data structures:

- $\text{imp.mem}$: $N \rightarrow N$ — A persistent memory. Initially, $\text{imp.mem}[i] = 0$ for all $i \in N$;
- $\text{caches}$: array$[1..n]$ of list of $E^i$ — Caches of clients. For each $p \in [1..n]$, $\text{caches}[p]$ is a sequence over $E^i_p$ that lists the events of the currently pending transaction of client $p$, if such exists. Initially, $\text{caches}[p]$ is empty for every $p$;
- $\text{imp.out}$: scalar in $E_\bot = E \cup \{\bot\}$ — An output variable recording responses to clients, initially $\bot$;
- $\text{imp.doomed}$: array$[1..n]$ of $\text{bools}$ — An array recording transactions that are doomed to be aborted. Initially $\text{imp.doomed}[p] = \perp$ for every $p$.

The implementation reacts to possible requests by the clients. It accepts a request of $\iota \triangleright_p$ (“open transaction”), and rejects any other transactional request if $\text{caches}[p]$ is empty. An accepted $\iota R^n_{\text{opf}}(x)$ request is responded by $\text{imp.mem}[x]$ (a result of the simplifying assumptions of Subsection 2.2) and added to $\text{caches}[p]$; A $\iota \eta^\delta_{\text{opf}}(x, v)$ request is simply acknowledged and as well added to $\text{caches}[p]$. Upon an invocation of a $\iota \triangleright_p$, $TM$ checks whether $\text{caches}[p]$ is
consistent with \( \text{imp\_mem} \) and if \( \text{imp\_doomed}[p] = F \). If they are, \( \text{TM} \) acknowledges it, updates \( \text{mem\_according\_to\_caches}[p] \), resets \( \text{caches}[p] \) to be empty, and sets \( \text{imp\_doomed}[q] = T \) for all clients \( q \) such that \( \text{caches}[q] \) consists of \( R^q_0(x) \) that is violated by a \( W^p_0(x) \) of \( \text{caches}[p] \). If \( \text{caches}[p] \) is not consistent with \( \text{imp\_mem} \), \( \text{TM} \) returns an “err”. When client \( p \) invokes a \( \triangleright_{p} \), \( \text{TM} \) acknowledges it, sets \( \text{imp\_doomed}[p] \) to \( F \), and resets \( \text{caches}[p] \) to be empty, \( R^p_0(x) \) request is responded by \( \text{imp\_mem}[x] \). For \( W^p_0(x, v) \) request, \( \text{mem\_according\_to\_caches}[p] \) is updated to \( v \), and for all clients \( q \) such that \( \text{caches}[q] \) consists of \( R^q_0(x) \), \( \text{imp\_doomed}[q] \) is set to \( T \).

Finally, the events corresponding to accepted requests are written to \( \text{imp\_out} \), which is set to \( \bot \) with steps that are not acknowledged or don’t produce a response. Each of the transactional events (with the exception of \( \triangleright_{p} \) and \( \triangleright_{q} \)), is appended to the appropriate \( \text{caches}[p] \).

The specification, described in Section 3, specifies not only the behavior of the Transactional Memory but also the combined behavior of the memory when coupled with a typical clients module. A generic clients module, \( \text{Clients}(n) \), may, at any step, invoke the next request for client \( p, p \in [1..n] \), provided the sequence of \( E^p \), events issued so far (including the current one) forms a prefix of a well-formed sequence. The justice requirement of \( \text{Clients}(n) \) is that eventually, every open transaction must be closed by issuing a successful \( \triangleright_{p} \) or \( \triangleright_{q} \).

Combining modules \( \text{TM} \) and \( \text{Clients}(n) \) we obtain the complete implementation, defined by:

\[
\text{Imp} : \quad \text{TM} \parallel \text{Clients}(n)
\]

where \( \parallel \) denote the synchronous composition operator defined in (6). We interpret this composition in a way that combines several of the actions of each of the modules into one.

The possible non-transactional actions of \( \text{Imp} \) are the following:

\( t_1 \). Set \( \text{imp\_out} \) to \( W^p_0(x, w) \) where \( w = \text{mem\_according\_to\_caches}[x] \), if \( \text{caches}[p] \) is empty;

\( t_2 \). Set \( \text{imp\_out} \) to \( W^p_0(y, v) \), update \( \text{mem\_according\_to\_caches}[y] \) to \( v \), and set \( \text{imp\_doomed}[q] \) to \( T \) if exist \( R^q_0(y) \in \text{caches}[q] \), if \( \text{caches}[p] \) is empty;

The possible transactional actions are the following:

\( t_3 \). Set \( \text{imp\_out} = \text{caches}[p] = \bot_{q} \) if \( \text{caches}[p] \) is empty;

\( t_4 \). Set \( \text{imp\_out} \) to \( W^p_0(x, w) \) and append it to \( \text{caches}[p] \), where \( w = \text{mem\_according\_to\_caches}[x] \), if \( \text{caches}[p] \) is non-empty, and \( \text{caches}[p] \) contains no \( W^p_0 \) events;

\( t_5 \). Set \( \text{imp\_out} \) to \( W^p_0(y, v) \) and append it to the end of \( \text{caches}[p] \) if \( \text{caches}[p] \) is non-empty;

\( t_6 \). Set \( \text{imp\_out} \) to \( \triangleright_{p} \), update \( \text{mem\_according\_to\_caches} \) according to \( \text{caches}[p] \), set \( \text{imp\_doomed}[q] \) to \( T \) if exist \( R^q_0(x) \in \text{caches}[q] \) and \( W^p_0(y) \in \text{caches}[p] \) where \( x = y \), and reset \( \text{caches}[p] \) to empty if \( \text{caches}[p] \) is non-empty, consistent with \( \text{imp\_mem} \) and \( \text{imp\_doomed}[p] = F \);

\( t_7 \). Set \( \text{imp\_out} \) to \( \triangleright_{q} \), set \( \text{caches}[p] \) to empty and \( \text{imp\_doomed}[p] \) to \( F \);

\( t_8 \). Set \( \text{imp\_out} \) to \( \bot \) and preserve all other variables;

Since \( \text{Clients}(n) \)'s justice requires every transaction to eventually issue a successful \( \triangleright_{p} \) or \( \triangleright_{q} \), and since \( t_6 \) and \( t_7 \) guarantee that \( \triangleright_{p} \) and \( \triangleright_{q} \) empty the corresponding \( \text{caches}[p] \), it follows that module \( \text{Imp} \) has a justice requirement: for each \( p = 1, \ldots, n \), \( \text{caches}[p] \) is empty infinitely many times.

6. Deductive Verification in TLPVS

In this section we describe how we used TLPVS (11) to verify the correctness of the implementation provided in Section 5. TLPVS was developed to reduce the substantial manual effort required to complete deductive proofs. It embeds temporal logic and its deductive framework within the high-order theorem prover. PVS (10). It includes a set of theories defining a temporal logic, proof rules for proving soundness and response properties, and strategies which aid in conducting the proofs. In particular, it has a special framework for verifying unbounded systems and theories. See (13) and (11) for thorough discussions for proving with PVS and TLPVS, respectively.

6.1 Observable Parameterized Fair Systems

Since TLPVS only supports the model of PFS, we formulate OPFS in PVS specification language as follows:

\[
\text{OPFS: TYPE} = \left[ \#\text{ initial: } \text{PREDICATE}, \right. \\
\text{rho: } \text{BI\_PREDICATE}, \\
\text{obs: } \text{OBSERVABLE}, \\
\text{justice: } \text{COMPASSION\_TYPE}, \\
\text{compassion: } \text{COMPASSION\_TYPE} #\right]
\]

where

\[
\text{PREDICATE: TYPE} = [\text{STATE} \rightarrow \text{bool}] \\
\text{BI\_PREDICATE: TYPE} = [\text{STATE, STATE} \rightarrow \text{bool}] \\
\text{OBSERVABLE: TYPE} = [\text{STATE} \rightarrow \text{OBSERVABLE\_DOMAIN}] \\
\text{JUSTICE\_TYPE: TYPE} = [\text{FAIRNESS\_DOMAIN} \rightarrow \text{PREDICATE}] \\
\text{COMPASSION\_PAIR: TYPE} = [\# p: \text{PREDICATE}, q: \text{PREDICATE}] \\
\text{COMPASSION\_TYPE: TYPE} = [\text{FAIRNESS\_DOMAIN} \rightarrow \text{COMPASSION\_PAIR}]
\]

When importing the OPFS theory, the user must instantiate it with \( \text{STATE, FAIRNESS\_DOMAIN} \) and \( \text{OBSERVABLE\_DOMAIN} \) parameters. These are uninterpreted types that must be defined by the user for each system. The rest of the module’s components are very similar to those provided for PFS, and we refer the reader to (11) for further reading.

6.2 Verifying the Trivial Implementation

After constructing the module of OPFS, we proved that the rule ABS-REL is correct. This required the definition of a new theory that imports two OPFSs, one for the abstract system (specification) and another for the concrete system (implementation). The proof itself was straightforward and did not require defining and proving additional lemmas.

Once ABS-REL was proved, we defined a none-standard theory for queues, to be used by the specification’s \( Q \) and the \( \text{caches} \) of the implementation. This theory required the definition of several
non-standard queue functions, in particular \texttt{project}, which is used for projecting \texttt{Q} on the events of a certain client. It also required to define and prove several side lemmas.

After having all the data structures constructed, the components of the OPFSs for the abstract and concrete systems are defined, namely, an initial condition, a transition relation, an observation function and fairness requirements for both systems. Note that in order to simplify the TLPS proof, some of the specification’s actions are combined together. For example, when committing a transaction, its events are interchanged, removed from \texttt{Q} and \texttt{spec\_out} is set to \texttt{s}, all in one step. This adjustment essentially restricts the allowed runs of the specification, yet sound. Namely, let \textit{Spec\_\textit{d}} denote the restricted specification, if \texttt{TM} \texttt{\subseteq \textit{Spec\_\textit{d}}} then \texttt{TM} \texttt{\subseteq \textit{Spec\_\textit{d}}.}

\textbf{The OBSERVABLE\_DOMAIN is identical for the two OPFSs:}

\texttt{OBSERVABLE\_DOMAIN: TYPE = \{out: EVENT\}}

where \texttt{out} stands for \texttt{spec\_out} and \texttt{imp\_out} that were defined in Section 3 and Section 5, respectively.

Applying rule \texttt{ABS-REL} requires the identification of an abstraction relation \textit{R} which holds between the concrete and abstract states. The following abstract relation was selected:

\[
rel: \text{RELATION} = \\
(\text{LAMBDA s_c, s_a:} \\
s_c \text{’out} = s_a \text{’out} \text{ AND} \\
s_c \text{’mem} = s_a \text{’mem} \text{ AND} \\
s_c \text{’doomed} = s_a \text{’doomed} \text{ AND} \\
\text{FORALL (id: ID):} \\
(\text{NOT s_c \text{’doomed(id)}) IMPLIES} \\
\text{project(id, s_c’Q}) = s_c \text{’caches[id] AND} \\
\text{FORALL (id: ID):} \\
(\text{empty(s_c’caches[id])}) IMPLIES} \\
\text{empty(project(id, s_c’Q})/) \text{ AND} \\
\text{INV s_c, s_a)}
\]

\textit{rel} defines a relation between the concrete state \texttt{s_c} and the abstract state \texttt{s_a}. It requires that the values of variables \texttt{out, mem} and \texttt{doomed} of the two systems are equal. It also defines the relation between the specification’s \texttt{Q} and the implementation’s \texttt{caches} by using the projection function \textit{project}. If the transaction of a client is not marked doomed, projecting \texttt{Q} on the client’s events is equal to the client’s local cache \texttt{caches[id]}. If the client’s transaction is empty, the projection of \texttt{Q} on its events should be empty as well.

\textit{INV} is the following inductive invariant:

\[
\text{INV(s_c, s_a): bool =} \\
\text{FORALL (id: ID):} \\
\text{memory\_consistency(project(id, s_a’Q), s_a’mem) AND} \\
\text{FORALL (id: ID):} \\
\text{empty(s_c’caches[id]) IMPLIES} \\
\text{NOT s_c \text{’doomed(id)}}
\]

Essentially \textit{INV} ensures that for each client, its read events are consistent with the memory at every step of the computation. The necessity for this invariant rises when committing a transaction, which is only permitted if the consistency is held. The invariant also guarantees that in the implementation a client without a pending transaction cannot be marked doomed. In order to certify that the invariant is inductive, it must be proved after each possible transition, which is performed as part of the proof for ABS-REL’s second premise.

In order to prove that \texttt{TM} \texttt{\subseteq Spec\_\textit{d}, D_C} and \texttt{D_A} of ABS-REL were instantiated with \texttt{TM} and \texttt{Spec\_\textit{d}}, respectively, and all the premises were proved to hold. The proof is relatively long and can be found at \texttt{http://eecs.nyu.edu/\texttt{arivel/TLPVS/tm.html}}.

7. \textbf{Related Work}

Scott (12) was the first to characterize transactional memory in a way that captured and clarified the many semantic distinctions among the most popular implementations. His approach was to begin with classical notions of transactional histories and sequential specifications, and to introduce two important notions: \textit{conflict function} and \textit{arbitration function}. The first specifies when two transactions cannot both succeed were the latter specifies which of two transactions must fail. Scott’s work was purely semantic and did not immediately facilitate mechanical verification of implementations.

In (3) we presented an abstract model for specifying transactional memory semantics, and proof rules for verifying that an implementation satisfies a transactional memory specification. We demonstrated the method by modeling in TLA\textsuperscript{+} three well-known transactional memory implementations and proved their correctness with the model checker TLC. The main contribution of that paper was the notion of the admissible interchange that were used to model the approaches to conflict detection and resolution characterized by Scott. Being able to model only finite-state systems is the main weakness of using model checkers, and what prevented from the proofs in (3) to be sound. In this work we use PVS-based theorem prover, therefore our proof of the correctness of Trivial Implementation is sound.

8. \textbf{Conclusion and Future Work}

We extended our previous work (3) that proposes a verification framework for transactional memory implementations against their specifications to accommodate non-transactional memory accesses. We also instrumented TLPVS to verify the correctness of implementations that have both transactional and non-transactional memory accesses.

Our work is based on the assumption that an implementation can detect non-transactional accesses, and is thus able to determine easily when such access conflicts with a pending transaction. We are currently working on extending our framework to enable the verification of implementations that cannot distinguish between transactional and non-transactional accesses, which will allow verifying implementations that have non-transactional accesses from a legacy code.

We are also working towards formal verification of implementations that employ more complex policies. Examples are LogTM (9) that has eager conflict detection and eager version management, and LTM (2) that has eager conflict detection and lazy version management, both were model checked (for restricted size instantiations in (3)). Finally, we are planning to study liveliness properties of transactional memory.

\textbf{References}


