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, has become the mainstream architecture for microprocessor chips and thus many new transactional memory implementations have been proposed recently (see (8) for an excellent survey).

A transactional memory (TM) receives requests from clients and issues responses. The requests are usually part of a *transaction* that is a sequence of operations starting with a request to *open* a transaction, followed by a sequence of read/write requests, followed by a request to *commit* (or *abort*). The TM responds to requests. When a transaction requests a successful "commit," all of its effects are stored in the memory. If a transaction is aborted (by either issuing an abort request or when TM detects that it should be aborted) all of its effects are removed. Thus, a transaction is a sequence of *atomic* operations, either all complete successfully and all its write operations update the memory, or none completes and its write operations do not alter the memory. In addition, committed transaction should be serializable – the sequence of operations belonging to successful transactions should be such that it can be reordered (preserving the order of operations in each transaction) so that the operation of each transaction appear consecutive, and a "read" from any memory location returns the value of the last "write" to that memory location.

Scott (12) wrote a widely-cited paper that was the first to characterize transactional memory in a way that captured and clarified the many semantic distinctions among the most popular implementations of TMs. His approach was to begin with classical notions of transactional histories and sequential specifications, and to introduce two important notions. The first is a *conflict function* which specifies when two overlapping (concurrent) transactions cannot both succeed (a safety condition). The second is an *arbitration function* which specifies which of two transactions must fail (a liveness condition). Scott's work went a long way towards clarification of the semantics of TMs, but did not facilitate mechanical verification of implementations.

In (2) we took a first step towards modeling TMs so to allow for mechanical verification of their implementations. We associated each conflict with its set *admissible* interchanges – a set of rules specifying when a pair of consecutive operations in a sequence of transactional operations can be safely swapped without introducing or removing the conflict. All the conflicts described in (12) can be cast as admissible sets. (Note, however, that here we restrict to rules that depend only on the history leading to the swap, and cannot express Scott's *mixed invalidation* conflict.) We then define a specification of a TM to be a fair transition systems whose input is a stream of transactional requests and whose output is a serializable sequence of the input requests and their responses. The fairness conditions guarantee that each transaction is eventually closed (committed or aborted) and, if committed, appears in the output.

To show that implementations (of TMs) satisfy their specifications, (2) introduces two proof rules. One that assumes a mapping between the states of the concrete (implementation) and the abstract (specification) system that has to be preserved with each step of the concrete system, and a more general rule that allows expressing the connection between the concrete and abstract states by a relation rather than a function (mapping). The more general rule also allows the abstract system to perform several steps for each concrete step. We also described how some well known TM implementations can be formally shown to satisfy their specifications (with the appropriate admissible sets). For sanity check, we automatically verified some small instantiations of each of those implementations with model checker TLC (7). However, since TLC cannot deal with relations between concrete and abstract states, we had to use the first proof rule mentioned above, and to (manually) identify, for each concrete step, an abstract "meta step" that consists of a sequence

of abstract steps as well as add some auxiliary variables to the concrete system to define the mapping.

As mentioned above, accesses to the memory are not necessarily all transactional. For various reasons (e.g., legacy code) memory accesses can be also *non-transactional*, which neither (12) nor (2) dealt with. Unlike transactional accesses, non-transactional accesses cannot be aborted. The atomicity and serializability requirements remain intact, where non-transaction operations are cast as a single operation, successfully committed, transaction.

We make here a strong assumption, under which the transactional memory is aware of non-transactional accesses, as soon as they occur. While the TM cannot abort such accesses, it may use them in order to abort transactions that are under its control. It is only with such or similar assumption that total consistency or coherence can be maintained.

In this paper we extend our previous work as follows:

- We extend the TM model to handle non-transactional memory accesses;
- We formalize the notion of admissible interchanges using temporal logic;
- We present a proof rule to verify that a concrete system implements an abstract one, where the correspondence between concrete and abstract states is a relation and the mapping between computations is modulo stuttering (of either system);
- We present a mechanical verification of the proof rule using TLPVS (11), a system that embeds temporal logic and its deductive frame-work within the theorem prover PVS (10);
- We present mechanical proofs that some TM implementations satisfy their specifications (given an admissible interchange) using TLPVS.

Using TLPVS entailed some modification to the formal model. We opted not to handle compassion (strong fairness) here since it is not used in any of the systems we studied.

To the best of our knowledge, the work presented here is the first to employ a theorem prover for verifying correctness of transactional memories and the first to formally verify an implementation that handles non-transactional memory accesses.

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

2. 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* (9). Under OPFS, a system $\mathcal{D} : \langle \Sigma, d_{\mathcal{O}}, \mathcal{O}, \Theta, \rho, \mathcal{F}, \mathcal{J}, \rangle$ consists of the following components:

- Σ — 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.
- $d_{\mathcal{O}}$ — A non-empty *observation domain* which is used for observing the external behavior of the states.
- $\mathcal{O}$ — An *observation functioin*. A mapping from Σ to $d_{\mathcal{O}}$.
- Θ — 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 $\mathcal{D}$-successor of s. We assume that every state has a ρ-successor.
- $\mathcal{F}$ — A non-empty *fairness domain* which is used for parameterizing the justice and compassion fairness requirements.
- $\mathcal{J}$ — A mapping from $\mathcal{F}$ to predicates. For each $f \in \mathcal{F}$, $\mathcal{J}(f)$ is a *justice* (*weak fairness*) requirement (predicate). For each $f \in \mathcal{F}$, a computation must include infinitely many states satisfying $\mathcal{J}(f)$.
- $\mathcal{C}$ — A mapping from $\mathcal{F}$ to pairs of predicates, called a *compassion (strong fairness)* requirement. For each $f \in \mathcal{F}$, let $\mathcal{C}(f) = \langle p, q \rangle$ 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 $\mathcal{D}$ is an infinite sequence of states $\sigma : s_0, s_1, \ldots$ satisfying the requirements:

- *Initiality* — s_0 is initial, i.e., $s_0 \models \Theta$.
- *Consecution* — For each $\ell = 0, 1, \ldots$, the state $s_{\ell+1}$ is an $\mathcal{D}$-successor of s_ℓ. That is $\rho(s_\ell, s_{\ell+1}) = \mathrm{T}$.

A *computation* of $\mathcal{D}$ is a run that satisfies

- *Justice* — for every $f \in \mathcal{F}$, σ contains infinitely many occurrences of $\mathcal{J}(f)$-states.
- *Compassion* – for every $f \in \mathcal{F}$ such that $\mathcal{C}(f) = \langle p, q \rangle \in \mathcal{C}$, either σ contains only finitely many occurrences of p-states, or σ contains infinitely many occurrences of q-states.

The differences between the FDS model and the one of PFS is due to the need to embed LTL and its proof theory within PVS. This representation allows the verification of parameterized systems where the size of the system is kept symbolic. The change from a PFS to the OPFS model was caused by the need to verify a refinement relation between a concrete and an abstract system and compare their observables.

3. Transactional Sequences and Interchanges

The first subsection presents an extension of the (2) model so to support non-transactional memory accesses and separate each action to a request and a response. The second subsection defines the interchange sets using past temporal logic formulae.

3.1 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:

- $\iota R_p^{nt}(x)$ – A non-transactional request to read from address $x \in \mathbb{N}$.
- $\iota W_p^{nt}(y, v)$ – A non-transactional request to write value $v \in \mathbb{N}$ to address $y \in \mathbb{N}$.

Let the set of *transactional invocations by client* p consists of:

- $\iota \blacktriangleleft_p$ – An open transaction request.
- $\iota R_p^{t}(x)$ – A transactional read request from address $x \in \mathbb{N}$.

- $\iota W_p^t(y,v)$ – A transactional request to write the value $v \in \mathbb{N}$ to address $y \in \mathbb{N}$.
- $\iota \blacktriangleright_p$ – A commit transaction request.
- $\iota \not\blacktriangleright_p$ – An abort transaction request.

The memory provides a response for each invocation. Erroneous invocations (e.g., a $\iota \blacktriangleleft_p$ while client p has a pending transaction) are responded by the memory returning an error flag err. Non-erroneous invocations, except for ιR^t and ιR^{nt} are responded by the memory returning an acknowledgment ack. Finally, for non-erroneous $\iota R_p^t(x)$ and $\iota R_p^{nt}(x)$ the memory returns the (natural) value of the memory at location x. Let ρ_p denote the set of responses to client p. We assume that invocations and responses occur atomically and consecutively, i.e., there are no other operation that interleave an invocation and its response.

Let E_p^{nt}: $\{R_p^{nt}(x,u), W_p^{nt}(x,v)\}$ be the set of *non-transactional observable events*, E_p^t: $\{\blacktriangleleft_p, R_p^t(x,u), W_p^t(x,v), \blacktriangleright_p, \not\blacktriangleright_p\}$ be the set of *transactional observable events* and $E_p = E^{nt} \cup E^t$, i.e. all events associated with client p. We consider as observable events only requests that are accepted, and abbreviate the pair (*invocation*, *non-err response*) by omitting the ι-prefix of the invocation. Thus, $W_p^t(x,v)$ abbreviates $\iota W_p^t(x,v), ack_p$. For read actions, we include the value read, that is, $R_p^t(x,u)$ abbreviates $\iota R^t(x), \rho R(u)$. When the value written/read has no relevance, we write the above as $W_p^t(x)$ and $R_p^t(x)$. When both values and addresses are of no importance, we omit the addresses, thus obtaining W_p^t and R_p^t (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., $E = \bigcup_{p=1}^n E_p$ (similarly define E^{nt} and E^t to be the set of all non-transactional and the set of all transactional observable events, respectively).

Let σ: $e_0, e_1, \ldots, e_k$ be a finite sequence of observable E-events. We say that the sequence $\hat{\sigma}$ over E^t is *σ's transactional sequence*, where $\hat{\sigma}$ is obtained from σ by replacing each R_p^{nt} and W_p^{nt} by $\blacktriangleleft_p\ R_p^t\ \blacktriangleright_p$ and $\blacktriangleleft_p\ W_p^t\ \blacktriangleright_p$, respectively. That is, each non-transactional event of σ is transformed into a singleton committed transaction in $\hat{\sigma}$. The sequence σ is called a *well-formed transactional sequence* (TS for short) if the following all hold:

1. For every client p, let $\hat{\sigma}|_p$ be the sequence obtained by projecting $\hat{\sigma}$ onto E_p^t. Then $\hat{\sigma}|_p$ satisfies the regular expression T_p^*, where T_p is the regular expression $\blacktriangleleft_p\ (R_p^t + W_p^t)^*(\blacktriangleright_p + \not\blacktriangleright_p)$. For each occurrence of T_p in $\hat{\sigma}|_p$, we refer to its first and last elements as *matching*. The notion of matching is lifted to $\hat{\sigma}$ itself, where $\blacktriangleleft_p$ and $\blacktriangleright_p$ (or $\not\blacktriangleright_p$) are matching if they are matching in $\hat{\sigma}|_p$;
2. The sequence $\hat{\sigma}$ is *locally read-write consistent*: for any subsequence of $\hat{\sigma}$ of the form $W_p^t(x,v)\ \eta\ R_p^t(x,u)$, where η contains no event of the form $\blacktriangleright_p$, $\not\blacktriangleright_p$, or $W_p^t(x,u)$, we have $u = v$.

We denote by $\mathcal{T}$ the set of all well-formed transactional sequences, and by and *pref*$(\mathcal{T})$ the sets $\mathcal{T}$'s prefixes. Note 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 $R_p^t(x)$ following a $W_p^t(x)$, 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.

A TS σ is called *atomic* if:

1. $\hat{\sigma}$ satisfies the regular expression $(T_1 + \cdots + T_n)^*$. That is, there is no overlap between any two transactions;
2. $\hat{\sigma}$ is *globally read-write consistent*: namely, for any subsequence $W_p^t(x,v)\eta R_q^t(x,u)$ in $\hat{\sigma}$, where η contains $\blacktriangleright_p$, which is not preceded by $\not\blacktriangleright_p$, and contains no event $W_k^t(x)$ followed by an event $\blacktriangleright_k$, it is the case that $u = v$.

3.2 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 differs the update until 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 was written by a transaction that is later aborted, thus the value is not valid since the transaction is never committed.

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, 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.

3.3 Interchanging Events

The notion of a correct implementation is that every TS can be transformed into an atomic TS by a sequence of interchanges which swap two consecutive events. This definition is parameterized by the set $\mathcal{A}$ of *admissible interchanges* which may be used in the process of serialization. Rather than attempt to characterize $\mathcal{A}$, we choose to characterize its complement $\mathcal{F}$, the set of *forbidden interchanges*. The definition here differs from the one in (2) in two aspects: There, in order to characterize $\mathcal{F}$, we allowed arbitrary predicates over the TS, here, we restrict to temporal logic formulae. Also, while (2) allowed swaps that depend on future events, here we restrict to swaps whose soundness depends only on the history leading to them. This restriction simplifies the verification process, and is the one used in all TM systems we are aware of. Note that it does not allow to express (12)'s mixed invalidation conflict.

In all our discussions, we assume *strict serializability* that implies that while serializing a TS, the order of committed transactions has to be preserved.

Consider a temporal logic over E using the past operators $\ominus$ (previously), $\diamondsuit$ (sometimes in the past), and $\mathcal{S}$ (since). Let σ be a prefix of a well-formed TS over E^t (i.e., $\sigma = \hat{\sigma}$). We define a satisfiability relation $\models$ between σ and a temporal logic formula φ so that $\sigma \models \varphi$ if at the end of σ, φ holds. (The more standard notation is $(\sigma, |\sigma|-1) \models \varphi$, but since we always interpret formulae at the end of sequences we chose the simplified notation.)

Some of the restrictions we place in $\mathcal{F}$ are structural. For example, the formula $\blacktriangleright_i \wedge \ominus\ \blacktriangleright_j$, $i \neq j$. forbids the interchange of clo-

sures of transactions belonging to different clients. This guarantees the strictness of the serializability process. Similarly, the restriction $u_i \wedge \ominus v_i$, where $u_i, v_i \in E_i$, forbids the interchanges of two events belonging to the same client. Other formulas may guarantee the absence of certain conflicts. For example, following (12), a *lazy invalidation* conflict occurs when committing one transaction may invalidate a read of another. Formally, if for some i, j, and x, where $i \neq j$,

$$\sigma \models \blacktriangleright_j \ \wedge \ \ominus(R_i(x) \ \wedge \ (\neg \blacktriangleright_j) \mathcal{S} \ W_j(x)) \qquad (1)$$

the last two events in σ *cannot* be interchanged. This formula guards against the occurrence of a lazy invalidation conflict by guaranteeing that any such occurrence cannot be serialized away because the pattern $\blacktriangleleft_i \cdots R_i(x) \cdots \blacktriangleright_j$ cannot be eliminated by any interchange. Similarly, we express conflicts by TL formulae that determine, for any prefix of a TS (that includes only E^t events) whether the two last events in the sequence can be safely interchanged without removing the conflict. For a conflict c, the formula that forbids interchanges that may remove instances of this conflict is called *the maintaining formula for* c and is denoted by m_c. Thus, Formula 1 is the maintaining formula for the conflict of lazy invalidation. We next provide a formulae for each of the other conflicts defined by (12) but for mixed invalidation that requires future operators.

1. An *overlap* conflict occurs if for some transactions T_i and T_j, we have $\blacktriangleleft_i \prec \blacktriangleright_j$ and $\blacktriangleleft_j \prec \blacktriangleright_i$. The corresponding maintaining formula m_o is
$$\blacktriangleright_j \ \wedge \ \ominus \blacktriangleleft_i$$
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 $\blacktriangleleft_i \prec \ W_j \ \prec \blacktriangleright_i$ or $W_j \ \prec \blacktriangleleft_i \prec \blacktriangleright_j$. The corresponding maintaining formula m_{wo} is
$$(\blacktriangleleft_j \wedge \ominus W_i) \ \vee (\blacktriangleright_i \wedge \ominus W_j)$$
3. 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 $W_i(x) \prec R_j(x) \prec \blacktriangleright_i$. The corresponding maintaining formula m_{ewr} is
$$m_{li} \ \vee \ (R_j(x) \wedge \ominus W_i(x))$$
4. 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 $R_i(x) \prec W_j(x) \prec \blacktriangleright_i$. The corresponding maintaining formula m_{ei} is
$$\begin{array}{lll} m_{ewr} & \vee & (W_j(x) \wedge \ominus R_i(x)) \\ & \vee & (\blacktriangleright_i \wedge \ominus(W_j(x) \ \wedge \ (\neg \blacktriangleright_i) \mathcal{S} \ R_i(x)))) \end{array}$$

Let $\mathcal{F}$ be a set of forbidden formulae characterizing all the forbidden interchanges, and let $\mathcal{A}$ denote the set of interchanges which do not satisfy any of the formulas in $\mathcal{F}$. Assume that $\sigma = a_0, \ldots, a_k$. Let σ' be obtained from σ by interchanging two elements, say a_{i-1} and a_i. We then say that σ' is *1-derivable from* σ *with respect to* $\mathcal{A}$ if $(a_0, \ldots, a_i) \not\models \bigwedge \mathcal{F}$. Similarly, we say that σ' is *derivable from* σ *with respect to* $\mathcal{A}$ if there exist $\sigma = \sigma_0, \ldots, \sigma_\ell = \sigma'$ such that for every $i < \ell$, σ_{i+1} is 1-derivable from σ_i with respect to $\mathcal{A}$.

A TS is *serializable with respect to* $\mathcal{A}$ if there exists an atomic TS that is derivable from it with respect to $\mathcal{A}$. The sequence $\breve{\sigma}$ is called the *purified version* of TS σ if $\breve{\sigma}$ is obtained by removing from $\hat{\sigma}$ 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.

4. Specification and Implementation

Let $\mathcal{A}$ be a set of admissible interchanges which we fix for the remainder of this section. We next describe *Spec*$_{\mathcal{A}}$ – a specification of transactional memory that generates all sequences whose corresponding TSs are serializable with respect to $\mathcal{A}$. The description here is somewhat informal, and we refer the reader to (2) for a formal description of similar (and simpler) specifications.

Spec$_{\mathcal{A}}$ is a process that is described as a fair transition system. In every step, it outputs an element in $E_\perp = E \cup \{\perp\}$. The sequence of outputs it generates, once the $\perp$ elements are projected away, is the set of TSs that are admissible with respect to $\mathcal{A}$. *Spec*$_{\mathcal{A}}$ maintains a queue-like structure $\mathcal{Q}$ to which elements are appended, interchanged, deleted, and removed. The sequence of elements removed from this queue-like structure defines an atomic TS that can be obtained by serialization of *Spec*$_{\mathcal{A}}$'s output with respect to $\mathcal{A}$. *Spec*$_{\mathcal{A}}$ uses the following data structures:

- *spec_mem*: $\mathbb{N} \to \mathbb{N}$ — A persistent memory. Initially, for all $i \in \mathbb{N}$, *spec_mem*$[i] = 0$;
- $\mathcal{Q}$: **list over** $E^t \cup \{mark_p\}$ initially empty;
- *spec_out*: **scalar in** $E_\perp = E \cup \{\perp\}$ — an output variable, initially $\perp$;
- *spec_doomed*: **array** $[1..n]$ **of booleans** — An array recording which pending transactions are doomed to be aborted. Initially *imp_doomed*$[p] =$ F for every p.

Fig. 1 summarized the steps taken by *Spec*$_{\mathcal{A}}$. The first column describes the value of *spec_out* with each step – it is assumed that every step produces an output. The second column describes the effects of the step on the other variables. The third column describes the conditions under which the step can be taken. We use the following abbreviations in Fig. 1:

- A client p is *pending* if *spec_doomed*$[p] =$ T or if $\mathcal{Q}|_p$ is not empty and does not terminate with $\blacktriangleright_p$;
- a client p is *marked* if $\mathcal{Q}|_p$ terminates with $mark_p$; otherwise, it is *unmarked*;
- a p action a is *locally consistent with* $\mathcal{Q}$ if $\mathcal{Q}|_p, a$ is a prefix of some locally consistent p-transaction;
- a transaction is *consistent with* spec_mem if every $R^t(x, v)$ in it is either preceded by some $W^t(x, v)$, or $v =$ *spec_mem*$[x]$;
- the *update of* spec_mem *by a (consistent) transaction* is *spec_mem*$'$ where for every location x for which the transaction has no $W^t(x, v)$ actions, *spec_mem*$'[x] =$ *spec_mem*$[x]$, and for every memory location x such that the transaction has some $W^t(x, v)$ actions, *spec_mem*$'[x]$ is the value written in the last such action in the transaction;
- an $\mathcal{A}$*-valid transformation to* $\mathcal{Q}$ is a sequence of interchanges of consecutive $\mathcal{Q}$ entries that each is consistent with $\mathcal{A}$. To apply the transformations, each $mark_p$ is treated as if it is $\blacktriangleright_p$.

The role of *spec_doomed* is to allow *Spec*$_{\mathcal{A}}$ to be implemented with various arbitration policies. It can, at will, schedule a pending transaction to be aborted by setting *spec_doomed*$[p]$, by so "dooming" p's pending transaction to be aborted. The variable *spec_doomed*$[p]$ is reset once the transaction is actually aborted (when *Spec*$_{\mathcal{A}}$ outputs $\not\blacktriangleright_p$). Note that actions of doomed transactions are not recorded on $\mathcal{Q}$.

spec_out	other updates	conditions
$\blacktriangleleft_p$	append $\blacktriangleleft_p$ to $\mathcal{Q}$	p is not pending
$R_p^t(x,v)$	append $R_p^t(x,v)$ to $\mathcal{Q}$	p is pending, unmarked and *spec_doomed*$[p] = \text{F}$; $R(x,v)$ is locally consistent with $\mathcal{Q}$
$\bar{R}_p^t(x,v)$	none	p is pending, unmarked and *spec_doomed*$[p] = \text{T}$
$W_p^t(x,v)$	append $W_p^t(x,v)$ to $\mathcal{Q}$	p is pending, unmarked and *spec_doomed*$[p] = \text{F}$
$\bar{W}_p^t(x,v)$	none	p is pending, unmarked and *spec_doomed*$[p] = \text{T}$
$\blacktriangleleft\!\!\!/_p$	delete p's pending transaction from $\mathcal{Q}$; set *spec_doomed*$[p]$ to F	p is pending
$\blacktriangleright_p$	update *spec_mem* by p's pending transaction; remove p-pending transaction from $\mathcal{Q}$	p has a consistent transaction at the front of $\mathcal{Q}$ that ends with $mark_p$ (p is pending and marked)
$R_p^{nt}(x,v)$	append $\blacktriangleleft_p, R_p^t(x,v), \blacktriangleright_p$ to $\mathcal{Q}$	p is not pending
$W_p^{nt}(x,v)$	append $\blacktriangleleft_p, W_p^t(x,v), \blacktriangleright_p$ to $\mathcal{Q}$	p is not pending
$\bot$	set *spec_doomed*$[p]$ to T; delete all pending p-events from $\mathcal{Q}$	p is pending and *spec_doomed*$[p] = \text{F}$
$\bot$	apply a $\mathcal{A}$-valid transformation to $\mathcal{Q}$	none
$\bot$	append $mark_p$ to $\mathcal{Q}$	p is pending and unmarked
$\bot$	none	none

Figure 1. Steps of $Spec_{\mathcal{A}}$

We assume a fairness requirement, namely, that for every client $p = 1, \ldots, n$, there are infinitely many states of $Spec_{\mathcal{A}}$ where $\mathcal{Q}|_p$ is empty. This implies that every transaction eventually terminates (commits or aborts). It also guarantees that the sequence of outputs is indeed serializable. Note that unlike the specification described in (2) where progress can always be guaranteed by aborting transactions, here, because of the non-transactional accesses, there are cases where $\mathcal{Q}$ cannot be emptied.

A sequence σ over E is *compatible with* $\text{Spec}_{\mathcal{A}}$ if it can be obtained by the sequence of *spec_out* that $Spec_{\mathcal{A}}$ outputs once all the $\bot$'s are removed. We then have:

CLAIM 1. *For every sequence σ over E, σ is compatible with* $\text{Spec}_{\mathcal{A}}$ *iff $\hat{\sigma}$ is serializable with respect to $\mathcal{A}$.*

An *implementation TM*: $(read, commit)$ of a transactional memory consists of a pair of functions

$$\begin{array}{rcll} read & : & pref(TS) \times [1..n] \times \mathbb{N} \to \mathbb{N} & \text{and} \\ commit & : & pref(TS) \times [1..n] \to \{ack, err\} & \end{array}$$

For a prefix σ of a TS, $read(\sigma, p, x)$ is the response (value) of the memory to an accepted $\iota R_p^{nt}(x)/\iota R_p^t(x)$ request immediately following σ, and $commit(\sigma, p)$ is the response (*ack* or *err*) of the memory to a $\iota \blacktriangleright_p$ request immediately following σ.

A TS σ is said to be *compatible* with the memory *TM* if:

1. For every prefix $\eta R_p^{nt}(x,u)$ or $\eta R_p^t(x,u)$ of σ, $read(\eta, p, x) = u$.
2. For every prefix $\eta \blacktriangleright_p$ of σ, $commit(\eta, p) = ack$.

An implementation *TM*: $(read, commit)$ is a *correct implementation of a transactional memory with respect to* $\mathcal{A}$ if every TS compatible with *TM* is also compatible with $Spec_{\mathcal{A}}$.

5. Verifying Implementation Correctness

We present a proof rule for verifying that an implementation satisfies the specification *Spec*. The rule is adapted version of the rule provided in (5), which was based on the *abstraction mapping* of (1). It is similar to the second rule of (2) with the following differences: The rule here cast in terms of a different formal framework (the one used in TLPVS), it captures general stuttering equivalence as opposed of assuming stuttering only on the part of the specification, and we omitted dealing with compassion (strong fairness) here since we believe compassion is not used in the TM framework.

To apply the underlying theory, we assume that both the implementation and the specifications are represented as OPFSs. 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 $\mathcal{O}(s) = s[O]$. 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 η: $e_0, e_1, \ldots$ be an infinite sequence of $E_\bot$-values. The $E_\bot$-sequence $\widetilde{\eta}$ is called a *stuttering variant* of the sequence η if it can be obtained by removing or inserting finite strings of the form $\bot, \ldots, \bot$ at (potentially infinitely many) different positions within η.

Let σ: $s_0, s_1, \ldots$ be a computation of OPFS $\mathcal{D}$, that is, a sequence of states where s_0 satisfies the initial condition, each state is a successor of the previous one, and for every justice (weak fairness) requirement, σ has infinitely many states that satisfy the requirement. The *observation* corresponding to σ (i.e., $\mathcal{O}(\sigma)$) 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 $Obs(\mathcal{D})$ the set of all observations of system $\mathcal{D}$.

Let $\mathcal{D}_C$ be a *concrete* system whose set of states is Σ_C, set of observations is $E_\bot$, observation function is $\mathcal{O}_C$, initial condition is Θ_C, transition relation is ρ_C, and justice requirements are $\cup_{f \in \mathcal{F}_C} \mathcal{J}(f)$. Similarly, let $\mathcal{D}_A$ be an *abstract* system whose set of states, set of observations, observation function, initial condition, transition relation, and justice requirements are Σ_A, $E_\bot$, $\mathcal{O}_A$, Θ_A, ρ_A, and $\cup_{f \in \mathcal{F}_A} \mathcal{J}(f)$ respectively. (For simplicity, we assume that neither system contains compassion requirements.) Note that we assume that both systems share the same observations domain $E_\bot$. We say that system $\mathcal{D}_A$ *abstracts* system $\mathcal{D}_C$ (equivalently $\mathcal{D}_C$ *refines* $\mathcal{D}_A$), denoted $\mathcal{D}_C \sqsubseteq \mathcal{D}_A$ if, for every observation $\eta \in Obs(\mathcal{D}_C)$, there exists $\widetilde{\eta} \in Obs(\mathcal{D}_A)$, such that $\widetilde{\eta}$ is a stuttering variant of η. In other words, modulo stuttering, $Obs(\mathcal{D}_C)$ is a subset of $Obs(\mathcal{D}_A)$. We denote by s and S the states of $\mathcal{D}_C$ and $\mathcal{D}_A$, respectively.

$$
\begin{array}{lllll}
\mathbf{R1}. & \Theta_C(s) & \rightarrow & \exists S : R(s,S) \ \wedge \ \Theta_A(S) & \\
\mathbf{R2}. & \mathcal{D}_C & \models & \Box(R(s,S) \ \wedge \ \rho_C(s,s') \ \rightarrow \ \exists S' : R(s',S') \ \wedge \ \rho_A(S,S')) & \\
\mathbf{R3}. & \mathcal{D}_C & \models & \Box(R(s,S) \ \rightarrow \ \mathcal{O}_C(s) = \mathcal{O}_A(S)) & \\
\mathbf{R4}. & \mathcal{D}_C & \models & \Box \Diamond(\forall S : R(s,S) \rightarrow \mathcal{J}(f)(S)), & \text{for every } f \in \mathcal{F}_A \\
\hline
 & & & \mathcal{D}_C \sqsubseteq \mathcal{D}_A &
\end{array}
$$

Figure 2. Rule ABS-REL.

Rule ABS-REL in Fig. 2 is a proof rule to establish that $\mathcal{D}_C \sqsubseteq \mathcal{D}_A$. The method advocated by the rule assumes the identification of an *abstraction relation* $R(s,S) \subseteq \Sigma_C \times \Sigma_A$. 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 $R(s,S) = \text{T}$.

Premise R2 states that for every pair of concrete states, s and s', such that s' is a ρ_C-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 ρ_A-successor of S. Together, R1 and R2 guarantee that, for every run $s_0, s_1, \ldots$ of $\mathcal{D}_C$ there exists a run $S_0, S_1, \ldots$, of $\mathcal{D}_A$, 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 σ a run of $\mathcal{D}_C$ there exists a corresponding run of $\mathcal{D}_A$ which has the same observation as σ. 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, $\Box$ is the (linear time) temporal operator for "henceforth", $\Diamond$ the temporal operator for "eventually", thus, $\Box \Diamond$ means "infinitely often." It follows that every sequence of abstract states which is R-related to a concrete computation σ and is obtained by applications of premises R1 and R2 is an abstract computation whose observables match the observables of σ. This leads to the following claim which was proved using TLPVS (see Section 7):

CLAIM 2. *If the premises of rule* ABS-REL *are valid for some choice of R, then $\mathcal{D}_A$ is an abstraction of $\mathcal{D}_C$.*

6. Example: TCC with non-transactional accesses

We demonstrate our proof method by verifying a TM implementation which is essentially TCC (3) augmented with non-transactional accesses. Its specifications is given by $\mathit{Spec}_{\mathcal{A}}$ where $\mathcal{A}$ is the admissible set of events corresponding to the lazy invalidation conflict described in Subsection 3.3.

In the implementation, transactions execute speculatively in the clients' caches. When a transaction commits, all pending transactions that contain some read events from addresses written to by the committed transaction are "doomed." Similarly, non-transactional writes cause pending transactions that read from the same location to be "doomed." A doomed transactions may execute new read and write events in its cache, but it must eventually abort.

Here we present the implementation, and in Section 7 explain how we can verify that it refines its specification using the proof rule ABS-REL in TLPVS. We refer to the implementation as *TM*. It uses the following data structures:

- *imp_mem*: $\mathbb{N} \rightarrow \mathbb{N}$ — A persistent memory. Initially, for all $i \in \mathbb{N}$, *imp_mem*$[i] = 0$;
- *caches*: **array**$[1..n]$ **of list of** E^t — Caches of clients. For each $p \in [1..n]$, *caches*$[p]$, initially empty, is a sequence over E_p^t that records the actions of p's pending transaction;
- *imp_out*: **scalar in** $E_\perp = E \cup \{\perp\}$ — an output variable recording responses to clients, initially $\perp$;
- *imp_doomed*: **array** $[1..n]$ **of booleans** — An array recording which transactions are doomed to be aborted. Initially *imp_doomed*$[p] = \text{F}$ for every p.

TM receives requests from clients, to each it updates its state, including updating the output variable *imp_out*, and issues a response to the requesting client. The responses are either a value in $\mathbb{N}$ (for a ιR^t or ιR^{nt} requests), an error err (for $\iota\blacktriangleright$ requests that cannot be performed), or an acknowledgment ack for all other cases. Fig. 3 describes the actions of *TM*, where for each request we describe the new output value, the other updates to *TM*'s state, the conditions under which the updates occur, and the response to the client that issues the request. For now, ignore the comments in the square brackets under the "conditions" column. The last line represents the idle step where no actions occurs and the output is $\perp$.

Comment: For simplicity of exposition, we assume that clients only issue reads for locations they had not written to in the pending transaction.

The specification of Section 4 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, *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 *Clients*(n) is that eventually, every pending transaction issues an ack-ed $\iota\blacktriangleright$ or an $\iota\not\blacktriangleright_p$.

Combining modules *TM* and *Clients*(n) we obtain the complete implementation, defined by:

$$\mathit{Imp}: \quad \mathit{TM} \ ||| \ \mathit{Clients}(n)$$

where $|||$ 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 actions of *Imp* can be described similarly to the one given by Fig. 3, where the first and last column are ignored, the conditions in the brackets are added. The justice requirements of *Clients*(n), together with the observation that both $\iota\not\blacktriangleright$ and an ack-ed $\iota\blacktriangleright$ cause the cache of the issuing client to be emptied, imply that *Imp*'s justice requirement is that for every $p = 1, \ldots, n$, *caches*$[p]$ is empty infinitely many times.

The application of rule ABS-REL requires the identification of a relation R which holds between concrete and abstract states. In (2), we used the relation defined by:

$$\begin{array}{c}
\mathit{spec_out} = \mathit{imp_out} \ \wedge \ \mathit{spec_mem} = \mathit{imp_mem} \\
\wedge \ \mathit{spec_doomed} = \mathit{imp_doomed} \\
\wedge \ \bigwedge_{p=1}^{n} \mathit{imp_doomed}[\text{p}] \longrightarrow (\mathcal{Q}|_p = \mathit{caches}[p])
\end{array}$$

however, there the implementation did not support non-transactional accesses. In Section 7 we provide the relation that was applied when proving the augmented implementation using TLPVS.

Request	*imp_out*	Other Updates	Conditions	Response
$\iota\blacktriangleleft_p$	$\blacktriangleleft_p$	append $\blacktriangleleft_p$ to *caches*$[p]$	[*caches*$[p]$ is empty]	ack
$\iota R_p^t(x)$	$R_p^t(x,v)$	append $R_p(x,v)$ to *caches*$[p]$	$v =$ *imp_mem*$[x]$; [*caches*$[p]$ is empty] (see comment)	*imp_mem*$[x]$
$\iota W_p^t(x,v)$	$W_p^t(x,v)$	append $W_p(x,v)$ to *caches*$[p]$	[*caches*$[p]$ is not empty]	ack
$\iota\not\blacktriangleleft_p$	$\not\blacktriangleleft_p$	set *caches*$[p]$ to empty	[*caches*$[p]$ is not empty]	ack
$\iota\blacktriangleright_p$	$\blacktriangleright_p$	set *caches*$[p]$ to empty; for every x and $q \neq p$ such that $W_p^t(x) \in$ *caches*$[p]$ and $R_p^t(x) \in$ *caches*$[q]$ set *spec_doomed*$[q]$ to T; update *imp_mem* by *caches*$[p]$	*imp_doomed*$[p] =$ F; [*caches*$[p]$ is not empty]	ack
$\iota\blacktriangleright_p$	$\perp$	none	*imp_doomed*$[p] =$ T; [*caches*$[p]$ is not empty]	err
$\iota R_p^{nt}(x)$	$R_p^{nt}(x,v)$	none	$v =$ *imp_mem*$[x]$; [*caches*$[p]$ is empty]	*imp_mem*$[x]$
$\iota W_p^{nt}(x,v)$	$W_p^{nt}(x,v)$	set *imp_mem*$[x]$ to v; for every q such that $R^t(x) \in$ *caches*$[q]$ set *spec_doomed*$[q]$ to T	[*caches*$[p]$ is empty]	ack
none	$\perp$	none	none	none

Figure 3. The actions of *TM*

7. Deductive Verification in TLPVS

In this section we describe how we used TLPVS (11) to verify the correctness of the implementation provided in Section 6. TLPVS was developed to reduce the substantial manual effort required to complete deductive temporal proofs of reactive systems. It embeds temporal logic and its deductive framework within the high-order theorem prover, PVS (10). It includes a set of theories defining linear temporal logic (LTL), 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 (10) and (11) for thorough discussions for proving with PVS and TLPVS, respectively.

In (2) we described the verification of three known transactional memory implementations with the (explicit-state) model checker TLC. This verification involved TLA$^+$ (7) modules for both the specification and implementation, and abstraction *mapping* associating each of the specification's variables with an expression over the implementation's variables.

This effort has several drawbacks: The mapping does not allow to specify abstraction relations between states, but rather mappings between variables and we therefore used a proof rule that is weaker than ABS-REL. For example, the relation $\mathcal{Q}|_p =$ *caches*$[p]$, cannot be expressed in TLA$^+$. We consequently used an auxiliary queue to record the order in which events are invoked in the implementation and mapped it to Q. Also, TLC can only verify finite instances of the system, with rather small numbers chosen for memory size, number of clients, bound on the size pending transactions, etc. As of yet, we have no "small number" properties of such systems, and the positive results could only be served as a sanity check. Hence, to obtain full mechanical verification we have to use a tool that is based on deductive techniques.

Our tool of choice is TLPVS. Since, however, TLPVS only supports the model of PFS, we formulate OPFS in the PVS specification language. We then defined a new theory that uses two OPFSs, one for the abstract system (specification) and another for the concrete system (implementation), and proved, in a rather straightforward manner, that the rule ABS-REL is sound.

We then defined a theory for the queue-like structures used in both specification and implementation. This theory required, in addition to the regular queue operations, the definition of the projection (|) and deletion of projected elements, which, in turn, required the proofs of several auxiliary lemmas.

Next all the components of both OPFSs defining the abstract and concrete systems were defined. To simplify the TLPVS proofs, some of the abstract steps were combined. For example, when a *Spec*$_{\mathcal{A}}$ commits a transaction, we combined the steps of interchanging events, removing them from $\mathcal{Q}$, and setting *spec_out* to $\blacktriangleright$. This restricts the set of *Spec*$_{\mathcal{A}}$'s runs but retains soundness. Formally, *TM* $\sqsubseteq \widetilde{Spec}_{\phi_{li}}$ implies that *TM* $\sqsubseteq$ *Spec*$_{\phi_{li}}$, where $\widetilde{Spec}_{\phi_{li}}$ is the restricted specification

The abstraction relation R between concrete and abstract states was defined as:

```
rel: RELATION =
   (LAMBDA s_c, s_a:
      s_c`out = s_a`out                          AND
      s_c`mem = s_a`mem                          AND
      s_c`doomed = s_a`doomed                    AND
      FORALL (id: ID):
        (NOT s_c`doomed(id)) IMPLIES
          project(id,s_a`Q) = s_c`caches(id)     AND
      FORALL (id: ID):
        (empty(s_c`caches(id))) IMPLIES
          empty(project(id,s_a`Q))               AND
```

Here, `s_c` is a concrete state and `s_a` is an abstract state. The relation R equates the values of `out`, `mem` and `doomed` in the two systems. It also states that if the transaction of a client is not doomed, then its projection on the abstract $\mathcal{Q}$ equals to the concrete client's cache, and if the concrete cache is empty then so is the projection of the abstract $\mathcal{Q}$ on the client's current transaction. An additional invariant, stating that each value read by a non-doomed client is consistent with the memory was also added.

In order to prove that *TM* $\sqsubseteq$ *Spec*$_{\phi_{li}}$, $\mathcal{D}_C$ and $\mathcal{D}_A$ of ABS-REL were instantiated with *TM* and *Spec*$_{\phi_{li}}$, respectively, and all the

premises were verified. The proofs may be found in $http://cs.nyu.edu/acsys/tlpvs/tm.html$.

8. Conclusion and Future Work

We extended our previous work on a verification framework for transactional memory implementations against their specifications, to accommodate non-transactional memory accesses. We also developed a methodology for verifying transactional memory implementations based on the theorem prover TLPVS that provides a framework for verifying parameterized systems against temporal specifications. We obtained mechanical verifications of both the soundness of the method and the correctness of an implementation which is based on TCC augmented with non-transactional accesses.

Our extension for supporting non-transactional accesses is based on the assumption that an implementation can detect such accesses. We are currently working on weakening this assumption. We are also planning to study liveness properties of transactional memory.

References

[1] M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, May 1991.

[2] A. Cohen, J. W. O'Leary, A. Pnueli M. R. Tuttle, and L. D. Zuck. Verifying correctness of transactional memories. In *Proceedings of FMCAD 2007*, 11 2007.

[3] L. Hammond, W.Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Herzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *Proc. 31^{s}t annu. Int. Symp. on COmputer Architecture*, June 2004.

[4] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. In *ISCA '93: Proceedings of the 20th annual international symposium on Computer architecture*, pages 289–300, New York, NY, USA, 1993. ACM Press.

[5] Y. Kesten, A. Pnueli, E. Shahar, and L. D. Zuck. Network invariants in action. In *13th International Conference on Concurrency Theory (CONCUR02)*, volume 2421 of *Lect. Notes in Comp. Sci.*, pages 101–115. Springer-Verlag, 2002.

[6] Yonit Kesten and Amir Pnueli. Verification by augmented finitary abstraction. *Inf. Comput.*, 163(1):203–243, 2000.

[7] L. Lamport. *Specifying Systems: The* TLA$^+$ *Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[8] James R. Larus and Ravi Rajwar. *Transactional Memory*. Morgan & Claypool Publishers, 2007.

[9] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, New York, 1995.

[10] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS System Guide*. Computer Science Laboratory, SRI International, Menlo Park, CA, September 1999.

[11] A. Pnueli and T. Arons. Tlpvs: A pvs-based ltl verification system. In *Verification–Theory and Practice: Proceedings of an International Symposium in Honor of Zohar Manna's 64th Birthday*, Lect. Notes in Comp. Sci., pages 84–98. Springer-Verlag, 2003.

[12] M.L. Scott. Sequential specification of transactional memory semantics. In *Proc. TRANSACT the First ACM SIGPLAN Workshop on Languages, Compiler, and Hardware Suppport for Transactional Computing*, Ottawa, 2006.