Write-observation and Read-preservation TM Correctness Invariants

Mohsen Lesani   Jens Palsberg
UCLA, University of California, Los Angeles
lesani@ucla.edu, palsberg@ucla.edu

1. Introduction

A transactional memory (TM) is a concurrent object with the three $read$, $write$ and $commit$ methods. The clients of a TM are transactions, a sequence of $read$ and $write$ invocations that are possibly succeeded by a $commit$ invocation. A transactional processing system is the composition of a TM and as set of clients. The clients issue the invocation events and the TM issues the response events. TM should guarantee that every concurrent execution of an arbitrary set of client transactions is indistinguishable from a sequential execution of them. Correctness conditions for TM such as opacity [7], VWC [11], and TMS1 and TMS2 [5] define the indistinguishably criterion and the set of correct histories.

Considering the promised safety properties, designing a correct TM is an art. TM algorithms are subtle programs that compose a TM from a set of base concurrent objects. Their subtly makes them vulnerable. Lesani et al [14] reported bugs and fixes for a couple of previously proposed TM algorithms. Verification of TM algorithm has been a topic of recent attention. Researchers have employed model checking, automatic invariant generation and theorem proving to verify the correctness of TM algorithms. Model checkers from Cohen et al. [1, 2], and Guerraoui et al. [8–10] are the pioneering approach to verification of TM. Model checking can automate the verification process but is either based on assumptions about the TM algorithm or only scalable to a finite number of threads and locations or simplified algorithms. Later, Emmi et al. [6] tried to automatically infer invariants that are strong enough to entail the correctness criterion. Their research tackled the very central part of the problem but reported resorting to simplified algorithms due to scalability issues. Later, Lesani et al. [12] presented a machine checked theorem proving framework and a full proof of NORec TM algorithm [3]. The framework can be employed to verify realistic algorithms but requires translation of the algorithm to a transition system and more importantly, the process involves coming up with non-trivial invariants.

Verification of TM algorithms has been a formidable problem in part because the target correctness criterion is a monolithic complicated condition. In an early work, Tasiran [15] presented a decomposition of the correctness condition for a specific class of algorithms. Can the correctness of TM be stated as a conjunction of simpler meaningful conditions? In other words, is there an intuitive functional decomposition of TM correctness conditions? What are the separate invariants that the TM designers should maintain?

We present intuitive invariants for the correctness of TM algorithms. We say that a history is markable if there is a specific ordering relation called marking such that three invariants are satisfied. These invariants are not only required but also sufficient for opacity. We prove the equivalence of markability and opacity. Roughly speaking, the first invariant called write-observation requires that each read operation returns the most current value and the second invariant called read-preservation requires that the location which is read is not overwritten in a certain interval and the third invariant is the well-known real-time-preservation property. We will look at these invariants more closely in the next section. Separation of concerns brings modularity in understanding, design and verification. Decomposition of the correctness condition showcases different aspects of it. Separate required conditions informs designers and helps them concentrate on maintaining one condition at a time. It also allows studying the effect of separate correctness aspects on performance. In addition, separation has obvious benefits of modularity and scalability for verification. The marking relation can be defined using the execution order or the linearization order of method calls on the used base objects. Thus, proofs of markability can be aided by and mirror design intuitions. Another important property of markability statement is that it can be proved using a program logic that we are developing. We are working on a machine checked proof of markability of the TL2 [4] algorithm.

2. Markability

In this section, we explain our main idea of markability. For clarity, we explain the crux of the problem by focusing on marking of complete histories with only global reads and writes. We present the formal and general definition of markability later.

A transaction history is markable if and only if there exists a marking of it that is write-observant, read-preserving, and real-time-preserving. We will explain each concept in turn.

A marking of a transaction history is a relation on the union of the transactions and the read operations in the history. We can think of the marking as the union of a collection of orders:

- The *effect order*: The effect order is a total order of the transactions.
- The *access orders*: Consider a read operation R that reads from a location i and doesn't abort. Let writers of i be the committed transactions that have write(s) to location i. For each such R, the access order is an antisymmetric relation that orders R and every writer of i.

The effect order represents the order in which the transactions appear to take effect, that is, the order that justifies the correctness of the history. The access order represents where R's access to location i has happened *between* the accesses by the writers of i. In other words, the access order identifies the writer whose written value the read operation has read.

Next, we will explain write-observation and read-preservation. Note that as $read$ is the only method in the transactional memory interface that exposes the state of the TM object, both of these two invariants constraint the return values of read operations.

Write-observation means that each read operation should read the most current value. Let us explain this idea in more detail.

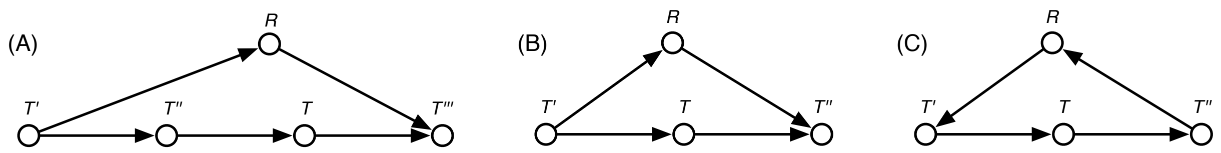

Figure 1. Illustrations of (A) write-observation, (B) read-preservation, and (C) a violation of read-preservation

Consider a read operation R from the location i. The *pre-accessors* are the writers of i that come before R in the access order for R. We can use the effect order to determine the *last* pre-accessor that is, the pre-accessor that is greatest in the effect order. Write-observation requires that the value that R has read is the same as the value written by the last pre-accessor. Figure 1.A illustrates the idea: The figure shows the marking relation $\sqsubseteq$ that is both the effect order $T' \sqsubseteq T'' \sqsubseteq T \sqsubseteq T'''$ and the access order $T' \sqsubseteq R \sqsubseteq T'''$. T is the transaction that performs R. Additionally, T' and T''' are writers of i while T'' is not a writer of i. T' is the last pre-accessor for R. Thus, R is expected to return the value that T' writes.

Read-preservation means that the location read by a read operation is not overwritten between the read accesses the location and the transaction takes effect. Let us explain this idea in more detail. Consider a read operation R by transaction T from location i. Intuitively, read-preservation requires that no writer of i should come between R and T in the marking relation. More precisely, read-preservation requires that there is no writer T' of i that accesses i *after* R and takes effect *before* T and there is no write T' of i that takes effect *after* T and accesses i *before* R. (Note that depending on whether a transaction takes effect earlier or later in its lifetime, one of these two conditions is usually trivially true.)

In other words, read-preservation requires the writers to both access i and take effect on the same "side" of R and T. More precisely, if a writer T' accesses i *before* R (T' is marked before R in the access order), then T' takes effect *before* T (T' is marked before T in the effect order) too. Similarly, read-preservation requires that if T' accesses i *after* R, it takes effect *after* T too.

Figures 1.B and 1.C illustrate read-preservation. R is a read from i by T. Additionally, T' and T'' are writers of i. The arrows show the marking relation. Figure 1.B shows a marking that is read-preserving. There is no writer between R and T. T' accesses i before R and takes effect before T too. T'' accesses i after R and takes effect after T too. Figure 1.C shows a marking where read-preservation is violated by both T' and T''. T' is between R and T and T'' is between T and R.

The real-time-preservation condition requires that if all the events of a transaction T happen before all the events of another transaction T', then T is less than T' in the effect order.

When a transaction history H is markable, we can pick a marking and construct a justifying history by ordering the transactions in the effect order. To see why the effect order makes a justifying history, consider an arbitrary read R from i by T. We call the writers of i that take effect before T, pre-effectors. Let T' be the last pre-effector in the effect order. We need to show that the value that R returns is the value that T' writes. We remind that we call the writers that access i before R, pre-accessors. First, we argue that pre-accessors are exactly pre-effectors. If a writer accesses before R, by read-preservation, it takes effect before T. If a writer takes effect before T, by antisymmetry of effect order, it does not take effect after T. Thus, by read-preservation, it does not access after R. Thus, by antisymmetry of access order, it accesses before R. Second, from write-observation, we have that R returns the value written by the last pre-accessor in the effect order. Thus from the two above statements, we have that R returns the value written by the last pre-effector in the effect order. This means that R returns the value written by T'. This is the essence of the condition needed to prove opacity. We will formalize this intuition in section 4.

Our marking theorem says that a history is opaque if and only if it is markable. So, to prove that a history is opaque we can focus on proving that it is markable. The algorithm designer can usually define the marking order readily from the guarantees (such as linearization orders) of the base objects.

The goal of this extended abstract is to introduce the notion of marking. We are working on machine checked proofs of markability of TL2 and DSTM (visible reads) algorithms. We conjecture marking relations for these two algorithms in the appendix [13]. In TL2, write-observation is satisfied by the the following: In the commit procedure, writes to a location are mutual exclusive by acquiring a lock. In the read procedure, it is checked that the version is unchanged while the value is being read and the lock is released. In DSTM (visible reads), write-observation is satisfied by aborting the tentative writer in the write procedure. In TL2, Read-preservation is satisfied by validation checks in the read procedure and also validation checks in the commit procedure. In DSTM (visible reads), read-preservation is satisfied by aborting the last writer in the read procedure and aborting the previous readers in the write procedure.

3. Histories

Strings. If s_1 and s_2 are strings, we write $s_1 \Subset s_2$ iff s_1 is a subsequence of s_2. For example, $bd \Subset abcde$. Let s be an isogram (i.e. contains no repeating occurrence of the alphabet.) For any $s_1, s_2 \in s$, we write $s_1 \lhd_s s_2$ iff the last element of s_1 occurs before the first element of s_2 in s. For example $ab \lhd_{abcde} de$. We use $s(i)$ to denote the i^{th} element of s.

Method calls and events. Let O denote the set of objects, n denote the set of method names, $Thread$ denote the set of threads, V denote the set of values and $Label$ denote the set of labels. We use l, R and W as labels. The set of invocation events is $Inv = \{inv(l \triangleright o.n_T(v)) \mid l \in Label, o \in O, n \in N, T \in Thread, v \in V\}$. The set of response events is $Res = \{ret(l \triangleright v) \mid l \in Label, v \in V \cup \{\mathbb{A}, \mathbb{C}\}\}$. ($\mathbb{A}$ and $\mathbb{C}$ are used later to denote abortion and commitment of transactions.) The set of events is $Ev = Inv \cup Res$. We will use the term completed method call to denote a sequence of an invocation event followed by the matching response event (with the same label). We use $l \triangleright o.n_T(v){:}v$ to denote the completed method call $inv(l \triangleright o.n_T(v)) \cdot ret(l \triangleright v)$.

Operations on event sequences. Let E and E' be event sequences. We use $E \cdot E'$ to denote the concatenation of E and E'. For a thread T, we use $E|T$ to denote the subsequence of all events of T in E. For an object o, we use $E|o$ to denote the subsequence of all events of o in E. $Sequential$ is the set of sequences of completed method calls possibly followed by an invocation event.

Execution history. An execution history X is a sequence of events where each invocation event has a unique label and every thread T is sequential (i.e. $X|T \in Sequential$). Let $History$ denote the set of execution histories. We say label l is in X and write $l \in X$ if there is an invocation event with label l in X. Let $Labels(X)$ denote the set of labels in X. Let $Threads(X)$ denote the set of threads in X. As the labels are unique in a history, the following functions on $Labels(X)$ are defined. The functions obj_X, $name_X$, $thread_X$, $arg1_X$, $arg2_X$, $retv_X$ map labels to

$$
\begin{aligned}
\mathit{Reads}(H) &= \{R \mid R \in H \ \wedge\ \mathit{obj}_H(R) = \mathit{this}\ \wedge \\
&\qquad\qquad \mathit{name}_H(R) = \mathit{read}\ \wedge\ \mathit{retv}_H(R) \neq \mathbb{A}\} \\
\mathit{Writes}(H) &= \{W \mid W \in H \ \wedge\ \mathit{obj}_H(W) = \mathit{this}\ \wedge \\
&\qquad\qquad \mathit{name}_H(W) = \mathit{write}\ \wedge\ \mathit{retv}_H(W) \neq \mathbb{A}\} \\
\mathit{Trans}(H) &= \{T \mid \exists l \in H \colon \mathit{thread}_H(l) = T\} \\
\mathit{TSequential} &= \{S \in \mathit{THistory} \mid \underline{\lessdot}_S \text{ is a total order of } \mathit{Trans}(S)\} \\
\mathit{Committed}(H) &= \{T \mid \exists l \in H \colon \mathit{thread}_H(l) = T\ \wedge\ \mathit{retv}_H(l) = \mathbb{C}\} \\
\mathit{Aborted}(H) &= \{T \mid \exists l \in H \colon \mathit{thread}_H(l) = T\ \wedge\ \mathit{retv}_H(l) = \mathbb{A}\} \\
\mathit{Completed}(H) &= \mathit{Committed}(H) \cup \mathit{Aborted}(H) \\
\mathit{Live}(H) &= \mathit{Trans}(H) \setminus \mathit{Completed}(H) \\
\mathit{TComplete} &= \{H \in \mathit{THistory} \mid \forall T \in \mathit{Trans}(H) \colon T \in \mathit{Completed}(H)\} \\
\mathit{CommitPending}(H) &= \{T \in \mathit{Live}(H) \mid \exists l \in H \colon \mathit{thread}_H(l) = T\ \wedge\ \mathit{name}_H(l) = \mathit{commit} \\
&\qquad\qquad \mathit{iEv}(l) \Subset H\ \wedge\ \neg(\mathit{rEv}(l) \Subset H)\} \\
\mathit{TExtension}(H) &= \{H' \in \mathit{THistory} \mid H \text{ is a prefix of } H'\ \wedge\ \forall T \in \mathit{Trans}(H') \Rightarrow T \in \mathit{Trans}(H)\ \wedge \\
&\qquad\qquad \mathit{Live}(H) \setminus \mathit{CommitPending}(H) \subseteq \mathit{Aborted}(H')\ \wedge \\
&\qquad\qquad \mathit{CommitPending}(H) \subseteq \mathit{Completed}(H')\} \\
\mathit{Visible}(S, T) &= \mathit{filter}\left(S, \lambda T'.(T' = T) \vee \left((T' \lessdot_S T) \wedge T' \in \mathit{Committed}(S)\right)\right) \\
\mathit{NoWriteBetween}_S(W, R) &= \forall W' \in \mathit{Writes}(S) \colon W' \preceq_S W\ \vee\ R \prec_S W' \\
\mathit{SeqSpec}(i) &= \{S \in \mathit{Sequential} \mid \forall R \in \mathit{Reads}(S) \colon \exists W \in \mathit{Writes}(S) \colon \\
&\qquad\qquad W \prec_S R\ \wedge\ \mathit{NoWriteBetween}_S(W, R)\ \wedge \\
&\qquad\qquad \mathit{retv}_S(R) = \mathit{arg2}_S(W)\} \\
\mathit{TSeqSpec} &= \{S \in \mathit{TSequential} \cap \mathit{TComplete} \mid \forall T \in S \colon \forall i \in I \colon \\
&\qquad\qquad (\mathit{Visible}(S, T) \mid i) \in \mathit{SeqSpec}(i)\} \\
\mathit{FinalStateOpaque} &= \{H \in \mathit{THistory} \mid \exists H' \in \mathit{TExtension}(H) \colon \exists S \in \mathit{TSequential} \colon \\
&\qquad\qquad H' \equiv S\ \wedge\ \underline{\lessdot}_{H'} \subseteq \underline{\lessdot}_S\ \wedge\ S \in \mathit{TSeqSpec}\}
\end{aligned}
$$

Figure 2. $\mathit{FinalStateOpaque}$

the receiving object, the method name, the thread identifier, the first and the second argument, and the return value associated with the labels. Similary, iEv and rEv functions on $\mathit{Labels}(X)$ map labels to the invocation and the response events associated with the labels.

A history X is equivalent to a history X', $X \equiv X'$, if one is a permutation of the other one that is only the events are reordered but the components of the events (including the argument and return values) are preserved.

Real-time relations. For an execution history X, we define the real-time relations $\prec_X$, $\preceq_X$, $\sim_X$, $\precsim_X$ on $\mathit{Labels}(X)$ as follows: First, $l_1 \prec_X l_2$ iff $\mathit{rEv}(l_1) \lhd_X \mathit{iEv}(l_2)$. $l_1 \preceq_X l_2$ iff $l_1 \prec_X l_2 \ \vee\ l_1 = l_2$. Second, $l_1 \sim_X l_2$ iff $l_1 \not\prec_X l_2 \ \wedge\ l_2 \not\prec_X l_1$. Third, $l_1 \precsim_X l_2$ iff $l_1 \prec_X l_2 \ \vee\ l_1 \sim_X l_2$.

From the definition of $\mathit{Sequential}$ we have that $X \in \mathit{Sequential}$ iff $\forall l, l' \in X \colon l \preceq_X l' \ \vee\ l' \prec_X l$. For an execution history X, we define the thread real-time relations $\lessdot_X$ and $\underline{\lessdot}_X$ as follows. First, $T \lessdot_X T'$ iff $X|T \lhd_X X|T'$. Second, $T \underline{\lessdot}_X T'$ iff $T \lessdot_X T' \ \vee\ T = T'$.

We now define shared memory and transaction histories.

Shared Memory. The shared memory is a singleton object mem that encapsulates the set of locations Loc where each location, $i \in I$, $I = \{1, \ldots, m\}$ stores a value $v \in V$. The object mem has three methods $\mathit{read}_T(i)$, $\mathit{write}_T(i, v)$ and commit_T. The method call $\mathit{read}_T(i)$ returns the value of location i or $\mathbb{A}$ (if the transaction is aborted). The method $\mathit{write}_T(i, v)$ writes v to location i and returns ok or returns $\mathbb{A}$. The method commit_T tries to commit transaction T and returns $\mathbb{C}$ (if the transaction is successfully committed) or returns $\mathbb{A}$ (if it is aborted). The object mem (or this) can be implicit, that is, $\mathit{read}_T(i)$ abbreviates $\mathit{this}.\mathit{read}_T(i)$.

Transaction History. A transaction history H is $\mathit{Init} \cdot H'$, where Init is the transaction $\mathit{write}_{T_0}(1, v_0), \ldots, \mathit{write}_{T_0}(m, v_0)$, $\mathit{commit}_{T_0}{:}C$ that initializes every location to v_0, and for all $T \in H'$: $H'|T$ is a prefix of $O.F$ where O is a sequence of reads $l \rhd \mathit{read}_T(i){:}v$ and writes $l \rhd \mathit{write}_T(i, v)$ (for some $l \in \mathit{Label}$, $i \in I$, and $v \in V$) and F is one of the following sequences: (1) $\mathit{inv}(l \rhd \mathit{read}_T(i)), \mathit{ret}(l \rhd \mathbb{A})$ (for some $l \in \mathit{Label}$ and $i \in I$), (2) $\mathit{inv}(l \rhd \mathit{write}_T(i, v)), \mathit{ret}(l \rhd \mathbb{A})$ (for some $l \in \mathit{Label}$, $i \in I$, and $v \in V$), (3) $\mathit{inv}(l \rhd \mathit{commit}_T), \mathit{ret}(l \rhd \mathbb{C})$, or (4) $\mathit{inv}(l \rhd \mathit{commit}_T)$, $\mathit{ret}(l \rhd \mathbb{A})$ (for some $l \in \mathit{Label}$). Let $\mathit{THistory}$ denote the set of transaction histories. The projection of H on i, written $H|i$, denotes the subsequence of history H that contains exactly the events on location i.

4. The Marking Theorem

In this section, we present a formal definition of opacity, define markability for general histories and state the marking theorem. $\mathit{FinalStateOpaque}$ is defined in Figure 2. As the definitions are self-descriptive, we narrate the definitions only in the appendix [13].

First, we present some preliminary definitions in the upper part of Figure 3. A local read is a read that is preceded by a write by the same transaction to the same location. Intuitively, a local read should read a value that is previously written by the same transaction and hence the name. A global read is a read that is not

$$\begin{array}{rcl}
\mathit{LocalReads}(H) & = & \{R \mid R \in \mathit{Reads}(H) \ \wedge\ \exists W \in \mathit{Writes}(H)\colon \\
& & \quad \mathit{thread}_H(R) = \mathit{thread}_H(W) \ \wedge\ \mathit{arg1}_H(R) = \mathit{arg1}_H(W) \ \wedge\ W \prec_H R\} \\
\mathit{GlobalReads}(H) & = & \mathit{Reads}(H) \setminus \mathit{LocalReads}(H) \\
\mathit{LocalWrites}(H) & = & \{W \mid W \in \mathit{Writes}(H) \ \wedge\ \exists W' \in \mathit{Writes}(H)\colon \\
& & \quad \mathit{thread}_H(W) = \mathit{thread}_H(W') \ \wedge\ \mathit{arg1}_H(W) = \mathit{arg1}_H(W') \ \wedge\ W \prec_H W'\} \\
\mathit{GlobalWrites}(H) & = & \mathit{Writes}(H) \setminus \mathit{LocalWrites}(H) \\
\mathit{LocalTSeqSpec} & = & \{H \in \mathit{THistory} \mid \forall R \in \mathit{LocalReads}(H)\colon \mathit{Let}\ T = \mathit{thread}_H(R), i = \mathit{arg1}_H(R)\colon \\
& & \quad \exists W \in \mathit{Writes}(H|T|i)\colon \\
& & \quad W \prec_{H|T|i} R \ \wedge\ \mathit{NoWriteBetween}_{H|T|i}(W, R) \ \wedge \\
& & \quad \mathit{retv}_H(R) = \mathit{arg2}_H(W)\} \\
\mathit{Writers}_H(i) & = & \{T \in \mathit{Trans}(H) \mid \exists l \in \mathit{Writes}(H)\colon \mathit{arg1}_H(l) = i \ \wedge \\
& & \quad \mathit{thread}_H(l) = T \ \wedge\ T \in \mathit{Committed}(H)\}
\end{array}$$

$$\begin{array}{l}
\mathit{NoWriterBetween}_{H,i}(q_1, \sqsubseteq, q_2) \iff \\
\qquad \forall T \in \mathit{Writers}_H(i)\colon T \sqsubseteq q_1 \ \vee\ q_2 \sqsubseteq T \\
\mathit{LastPreAccessor}_H(R, T') \iff \\
\qquad \mathit{Let}\ i = \mathit{arg1}_H(R), T = \mathit{thread}_H(R)\colon \\
\qquad T' \in \mathit{Writers}_H(i) \ \wedge\ T' \sqsubset R \ \wedge\ T' \neq T \ \wedge \\
\qquad \mathit{NoWriterBetween}_{H,i}(T', \sqsubseteq, R) \\
\sqsubseteq\ \in \mathit{WriteObs}(H) \iff \\
\qquad H \in \mathit{LocalTSeqSpec} \wedge \\
\qquad \forall R \in \mathit{GlobalReads}(H)\colon \exists W \in \mathit{GlobalWrites}(H)\colon \mathit{Let}\ T' = \mathit{thread}_H(W)\colon \\
\qquad \mathit{LastPreAccessor}_H(R, T') \wedge \\
\qquad \mathit{arg1}_H(R) = \mathit{arg1}_H(W) \ \wedge\ \mathit{retv}_H(R) = \mathit{arg2}_H(W) \\
\sqsubseteq\ \in \mathit{ReadPres}(H) \iff \\
\qquad \forall R \in \mathit{GlobalReads}(H)\colon \\
\qquad \mathit{Let}\ i = \mathit{arg1}_H(R), T = \mathit{thread}_H(R)\colon \\
\qquad \mathit{NoWriterBetween}_{H,i}(R, \sqsubseteq, T) \ \wedge\ \mathit{NoWriterBetween}_{H,i}(T, \sqsubseteq, R) \\
\sqsubseteq\ \in \mathit{RealTimePres}(H) \iff \\
\qquad \preceq\!\!\!\prec_H \subseteq \sqsubseteq \\
\mathit{Markable} = \{H \in \mathit{THistory} \mid \exists H' \in \mathit{TExtension}(H)\colon \exists \sqsubseteq\ \in \mathit{Marking}(H')\colon \\
\qquad\qquad \sqsubseteq\ \in \mathit{ReadPres}(H') \cap \mathit{WriteObs}(H') \cap \mathit{RealTimePres}(H')\}
\end{array}$$

Figure 3. $\mathit{Markable}$

local. A local write is a write that precedes a write by the same transaction to the same location. A local write is overwritten by the same transaction. A global write is a read that is not local. A history is in the local transactional sequential specification if each local read returns the value written by the last write of the same transaction before the read. The writers of i are the committed transactions that write to location i.

Let $H \in \mathit{THistory}$. A marking of H, $\sqsubseteq$, is the union of the following relations on $\mathit{Trans}(H) \cup \mathit{Reads}(H)$

- The *effect order*: $\mathit{Trans}(H)$ is totally ordered by $\sqsubseteq$
- The *access orders*: For each $R \in \mathit{Reads}(H)$ where $i = \mathit{arg1}_H(R)$, R and every member of $\mathit{Writers}_H(i)$ are ordered by $\sqsubseteq$. Access order is antisymmetric.

$\mathit{Marking}(H)$ is the set of markings of H.

Write-observation requires the history to be in the local transactional sequential specification. Each local read should return the value written by the last write in the same transaction. Also, it requires that the value that every global read returns is the value written by the global write of the last pre-accessor.

Read-preservation requires that the location read by a global read operation is not overwritten between the read accesses the location and the transaction takes effect.

The real-time-preservation condition requires that if T is before T' in the real-time order, then T takes effect before T' as well.

A transaction history is markable if and only if there exists a marking for an extension of it that is write-observant, read-preserving, and real-time-preserving.

The marking theorem states that a transaction history is final-state-opaque if and only if it is markable.

Theorem 1. ***(Marking)*** $\mathit{FinalStateOpaque} = \mathit{Markable}$.

Please see the appendix [13] for the proofs.

References

[1] Ariel Cohen, John W. O'Leary, Amir Pnueli, Mark R. Tuttle, and Lenore D. Zuck. Verifying correctness of transactional memories. In *FMCAD*, 2007.

[2] Ariel Cohen, Amir Pnueli, and Lenore D. Zuck. Mechanical verification of transactional memories with non-transactional memory accesses. In *CAV*, 2008.

[3] Luke Dalessandro, Michael F. Spear, and Michael L. Scott. Norec: streamlining stm by abolishing ownership records. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 67–78, New York, NY, USA, 2010. ACM.

[4] Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking ii. In *In Proc. of the 20th Intl. Symp. on Distributed Computing*, 2006.

[5] S. Doherty, L. Groves, V. Luchangco, and M. Moir. Towards formally specifying and verifying transactional memory. *Formal Aspects of Computing*, 2012.

[6] Michael Emmi, Rupak Majumdar, and Roman Manevich. Parameterized verification of transactional memories. In *Proceedings of PLDI'10, ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 134–145, June 2010.

[7] R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *PPOPP*, pages 175–184, 2008.

[8] Rachid Guerraoui, Thomas A. Henzinger, Barbara Jobstmann, and Vasu Singh. Model checking transactional memories. In *ACM SIGPLAN Conference on Programming Languages Design and Implemen tation*, pages 372–382, 2008.

[9] Rachid Guerraoui, Thomas A. Henzinger, and Vasu Singh. Software transactional memory on relaxed memory models. In *Proceedings of CAV'09, Seventh International Conference on Computer Aided Verification*, pages 321–336, 2009.

[10] Rachid Guerraoui, Thomas A. Henzinger, and Vasu Singh. Model checking transactional memories. *Distributed Computing*, 2010.

[11] Damien Imbs, José Ramon de Mendivil, and Michel Raynal. Brief announcement: virtual world consistency: a new condition for stm systems. In *Proceedings of the 28th ACM symposium on Principles of distributed computing*, PODC '09, pages 280–281, New York, NY, USA, 2009. ACM.

[12] Mohsen Lesani, Victor Luchangco, and Mark Moir. A framework for formally verifying software transactional memory algorithms. In *CONCUR*, 2012.

[13] Mohsen Lesani and Jens Palsberg. The appendices. `http://www.cs.ucla.edu/~lesani/submission/wttm/`.

[14] Mohsen Lesani and Jens Palsberg. Putting non-opacity. In *Transact'13*, 2013.

[15] Serdar Tasiran. A compositional method for verifying software transactional memory implementations. Technical Report MSR-TR-2008-56, Microsoft Research, apr 2008.