Decomposing Opacity
(Appendix)

Mohsen Lesani Jens Palsberg
Computer Science Department
University of California, Los Angeles
{lesani, palsberg}@ucla.edu

Contents

1 Histories

Strings. We use $\|s\|$ to denote the size of the string s. 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, $Trans$ denote the set of transactions, 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 $Inv = \{inv(l \triangleright\ o.n_T(v)) \mid l \in Label, o \in O, n \in N, T \in Trans, v \in V\}$. 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 transaction 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. Let $Sequential$ be the set of sequences of completed method calls possibly followed by an invocation event. A transaction T is sequential in a sequence of events E if $E|T$ is sequential. **Execution history.** An execution history is a sequence of events where each invocation event has a unique label and every transaction is 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 $Trans(X)$ denote the set of transactions in X. As the labels are unique in a history, the following functions on $Labels(X)$ are defined. The functions obj_X, $name_X$, $trans_X$, $arg1_X$, $arg2_X$, $retv_X$ map labels to the receiving object, the method name, the transaction identifier, the first and the second argument, and the return value associated with the labels. Similary, iEv and rEv functions on $Labels(X)$ 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 $Labels(X)$ as follows: First, $l_1 \prec_X l_2$ iff $rEv(l_1) \lhd_X iEv(l_2)$. $l_1 \preceq_X l_2$ 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 $Sequential$ we have that $X \in Sequential$ iff $\forall l, l' \in X \colon l \preceq_X l' \ \vee\ l' \prec_X l$. For an execution history X, we define the real-time relations $\prec\!\!\prec_X$ and $\preceq\!\!\!\preceq_X$ as follows. First, $T \prec\!\!\prec_X T'$ iff $X|T \lhd_X X|T'$. Second, $T \preceq\!\!\!\preceq_X T'$ iff $T \prec\!\!\prec_X T' \ \vee\ T = T'$.

We now define shared memory and transaction histories.

Transactional Memory. The *transactional memory* is a singleton object mem that encapsulates a set of locations where each location, $i \in I$, $I = \{1, \ldots, m\}$ encapsulates a value v. The object mem has five methods $init_t()$, $read_t(i)$, $write_t(i, v)$, $commit_t()$ and $abort_t()$. The parameter t is the invoking transaction identifier. The method call $init_t()$ initializes t and returns ok. The method call $read_t(i)$ returns the value of location i or aborts t and returns $\mathbb{A}$. The method $write_t(i, v)$ writes v to location i and returns ok or aborts t and returns $\mathbb{A}$. The method $commit_t()$ tries to commit transaction t. If t is successfully committed, it returns $\mathbb{C}$; otherwise, it returns $\mathbb{A}$. The method $abort_t()$ aborts t and returns $\mathbb{A}$. The object mem can be implicit, that is $read_t(i)$ abbreviates $mem.read_T(i)$. The values ok, $\mathbb{A}$, $\mathbb{C}$ are reserved values that are used to denote successful completion of writes and abortion and commitment of transactions respectively.

Transaction History. A *transaction history* H is an execution history such that $H|mem = H_{Init} \cdot H'$ with the following conditions. H_{Init} is the following history that initializes every location to v_0. $H_{Init} =$

$l_{0i} \triangleright init_{T_0}() \cdot l_{00} \triangleright write_{T_0}(1, v_0){:}ok \cdot \ldots \cdot l_{0m} \triangleright write_{T_0}(m, v_0){:}ok \cdot l_{0c} \triangleright commit_{T_0}{:}\mathbb{C}$. For every $T \in H'$, the history $H'|T$ is a prefix of $e.e'$. The event sequence e is the initialization method call $l \triangleright init_T()$ (for some l), and then a sequence of reads $l \triangleright read_T(i){:}v$ and writes $l \triangleright write_T(i, v)$ (for some l, i, and v). The event sequence e' is one of the following sequences (for some l, i, and v): (1) $inv(l \triangleright read_T(i))$, $ret(l \triangleright \mathbb{A})$, (2) $inv(l \triangleright write_T(i, v))$, $ret(l \triangleright \mathbb{A})$, (3) $inv(l \triangleright commit_T())$, $ret(l \triangleright \mathbb{C})$, (4) $inv(l \triangleright commit_T())$, $ret(l \triangleright \mathbb{A})$, or (5) $inv(l \triangleright abort_T())$, $ret(l \triangleright \mathbb{A})$. Let $THistory$ denote the set of transaction histories. Let $Trans(H)$ denote the set of transactions of H. The projection of H on i, written $H|i$, denotes the subsequence of history H that contains exactly the events on location i. For a TM algorithm specification π, let $\mathbb{H}(\pi)$ denote the set of complete transaction histories that π results.

Now, we present a set of basic lemmas about execution orders.

Lemma 1 (XASym) *For every execution history X and method calls l and l', if $l \prec_X l'$ then $\neg(l' \prec_X l) \wedge \neg(l' \sim_X l) \wedge \neg(l' = l)$*

Lemma 2 (XTrans) *For every execution history X and method calls l, $l.$ and l'', if $l \prec_X l'$ and $l' \prec l''$ then $l \prec_X l''$*

Lemma 3 (XXTrans) *For every execution history X and method calls l_1, l_2, l_3, and l_4, if $l_1 \prec_X l_2$, $l_2 \precsim_X l_3$, and $l_3 \prec_X l_4$ then $l_1 \prec_X l_4$*

Lemma 4 (XTotal) *For every execution history X and method calls l and l', if $l \in X$, and $l' \in X$, then $(l \prec_X l') \vee (l' \prec_X l) \vee (l \sim_X l') \vee (l = l')$*

Lemma 5 (X2X) *For every execution history X and method calls l and l', if $l \prec_X l'$ then $l \in X$, and $l' \in X$.*

Lemma 6 (XI2X) *For every execution history X and method calls l, l', and l'' if $l \prec_X l'$ and $inv(l') \lhd_X inv(l'')$ then $l \prec_X l''$.*

Lemma 7 (RX2X) *For every execution history X and method calls l, l', and l'' if $ret(l) \lhd_X ret(l')$ and $l' \prec_X l''$ then $l \prec_X l''$.*

Proof Sketches.

Lemma 1:

We Assume

(1) $l \prec_X l'$

From [1] and definition of $\sim_X$, we have

(2) $\neg(l' \sim_X l)$

From [1], we have

(3) $rEv(l) \lhd_X iEv(l')$

As X is a valid history, we have

(4) $iEv(l) \lhd_X rEv(l)$

(5) $iEv(l') \lhd_X rEv(l')$

From [4], [3], and [5], we have

(6) $iEv(l) \lhd_X rEv(l')$

From [6], we have

(7) $\neg(rEv(l') \lhd_X iEv(l))$

From [7], and definition of $\prec_X$, we have

(9) $\neg(l' \prec_X l)$

From [3] and [7], we have

(9) $\neg(l' = l)$

Lemma 2:

Straightforward from the definition of $\prec_X$.

Lemma 3:

We have

(1) $l_1 \prec_X l_2$
(2) $l_3 \prec_X l_4$
(3) $l_2 \sim_X l_3$

From [1], we have

(4) $rEv(l_1) \lhd_X iEv(l_2)$

From [2], we have

(5) $rEv(l_3) \lhd_X iEv(l_4)$

From [3], we have

(6) $\neg(l_3 \prec_X l_2)$

From [6], we have

(7) $\neg(rEv(l_3) \lhd_X iEv(l_2))$

From [7], we have

(8) $iEv(l_2) \lhd_X rEv(l_3)$

From [4], [8], and [5], we have

(9) $rEv(l_1) \lhd_X iEv(l_4)$

From [9], we have

$l_1 \prec_X l_4$

Lemma 4:

Straightforward from the definition of $\prec_X$ and $\sim_X$.

Lemma 5:

Straightforward from the definition of $\prec_X$.

Lemma 6:

Straightforward from the definition of $\prec_X$ and $\lhd_X$.

Lemma 7:

Straightforward from the definition of $\prec_X$ and $\lhd_X$.

2 Opacity

In this section, we present a formal definition of opacity. Opacity of a TM algorithm is defined in two steps. First, it is defined what it means for a transaction history to be opaque which is called final-state-opacity. Then, a TM algorithm is defined to be opaque if every transaction history of every source program running on top of that TM algorithm is final-state-opaque.

$FinalStateOpaque$ is defined in Figure 1. First, we present some preliminary definitions. We use T prefix before some of the terms for transactions to avoid confusion with the terms for concurrent objects. We say that a transaction history is *transaction sequential* if it is a sequence of transactions. A transaction T is *committed* or *aborted* in a transaction history H if there is respectively a commit or abort response event for T in H. A *completed* transaction is either committed or aborted. A *live* transaction is a transaction that is not completed. A transaction history is *complete* if all its transactions are completed. A *pending* transaction has a pending event and a *commit-pending* transaction has a commit pending event. An *extension* of a history is obtained by committing or aborting its commit-pending transactions and aborting the other live transactions. For a TM algorithm specification π, let $\mathbb{H}(\pi)$ denote the set of complete transaction histories that π results.

If H is a transaction history and p is a predicate on transaction identifiers, we define $filter(H, p)$ to be the subsequence of H that contains the events of transactions T for which $p(T)$ is true. The *visible history* for a transaction T in a sequential transaction history S, $Visible(S, T)$, is the sequence of committed transactions before T in S and T itself. The *sequential specification* of a location i, $SeqSpec(i)$, is the set of sequential histories of read and write method calls on location i where every read returns the value given as the argument to the latest preceding write (regardless of transaction identifiers). It is essentially the sequential specification of a register. *Transactional sequential specification* is the set of complete sequential transaction histories S that for every transaction T and location i, $Visible(S, T)|i$ is a member of the sequential specification of i. A transaction history H is *final-state-opaque* if there is an equivalent sequential transaction history S for an extension of H such that S is real-time-preserving and a member of transactional sequential specification. The sequential history S is called the justifying history. In other words, every correct concurrent execution is indistinguishable from a correct sequential execution.

$$
\begin{aligned}
&\mathit{TReads}(H) = \\
&\qquad \{R \mid R \in H \ \wedge\ \mathit{obj}_H(R) = \mathit{mem} \ \wedge\ \mathit{name}_H(R) = \mathit{read} \ \wedge\ \mathit{retv}_H(R) \neq \mathbb{A}\} \\
&\mathit{TWrites}(H) = \\
&\qquad \{W \mid W \in H \ \wedge\ \mathit{obj}_H(W) = \mathit{mem} \ \wedge\ \mathit{name}_H(W) = \mathit{write} \ \wedge\ \mathit{retv}_H(W) \neq \mathbb{A}\} \\
&\mathit{Commits}(H) = \\
&\qquad \{C \mid C \in H \ \wedge\ \mathit{obj}_H(C) = \mathit{mem} \ \wedge\ \mathit{name}_H(C) = \mathit{commit}\} \\
&\mathit{Trans}(H) = \\
&\qquad \{T \mid \exists l \in H \colon \mathit{trans}_H(l) = T\} \\
&\mathit{TSequential} = \\
&\qquad \{S \in \mathit{THistory} \mid \preccurlyeq_S \text{ is a total order of } \mathit{Trans}(S)\} \\
&\mathit{Committed}(H) = \\
&\qquad \{T \mid \exists l \in \mathit{Commits}(H) \ \wedge\ \mathit{retv}_H(l) = \mathbb{C}\} \\
&\mathit{Aborted}(H) = \\
&\qquad \{T \mid \exists l \in H \colon \mathit{obj}_H(l) = \mathit{mem} \ \wedge\ \mathit{trans}_H(l) = T \ \wedge\ \mathit{retv}_H(l) = \mathbb{A}\} \\
&\mathit{Completed}(H) = \\
&\qquad \mathit{Committed}(H) \cup \mathit{Aborted}(H) \\
&\mathit{Live}(H) = \\
&\qquad \mathit{Trans}(H) \setminus \mathit{Completed}(H) \\
&\mathit{TComplete} = \\
&\qquad \{H \in \mathit{THistory} \mid \forall T \in \mathit{Trans}(H) \colon T \in \mathit{Completed}(H)\} \\
&\mathit{CommitPending}(H) = \\
&\qquad \{T \in \mathit{Live}(H) \mid \exists l \in H \colon \mathit{obj}_H(l) = \mathit{mem} \ \wedge\ \mathit{trans}_H(l) = T \ \wedge\ \mathit{name}_H(l) = \mathit{commit}\} \\
&\mathit{TExtension}(H) = \\
&\qquad \{H' \in \mathit{THistory} \mid \exists H'' \colon H' = H \cdot H'' \\
&\qquad\qquad \mathit{Trans}(H'') \subseteq \mathit{Trans}(H) \ \wedge\ \forall T \colon \|H''|T\| \leq 1 \ \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) = \\
&\qquad \mathit{filter}\,(S, \lambda T'.(T' = T) \vee ((T' \llcurly_S T) \wedge T' \in \mathit{Committed}(S))) \\
&\mathit{NoWriteBetween}_S(W, R) = \\
&\qquad \forall W' \in \mathit{TWrites}(S) \colon W' \preceq_S W \ \vee\ R \prec_S W' \\
&\mathit{SeqSpec}(i) = \\
&\qquad \{S \in \mathit{Sequential} \mid \forall R \in \mathit{TReads}(S) \colon \exists W \in \mathit{TWrites}(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} = \\
&\qquad \{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} = \\
&\qquad \{H \in \mathit{THistory} \mid \exists H' \in \mathit{TExtension}(H) \colon \exists S \in \mathit{TSequential} \colon \\
&\qquad\qquad H' \equiv S \wedge\ \preccurlyeq_{H'} \subseteq \preccurlyeq_S \ \wedge S \in \mathit{TSeqSpec}\}
\end{aligned}
$$

Figure 1: $\mathit{FinalStateOpaque}$

3 Markability

In this section, we define markability for general histories.

First, we present some preliminary definitions in Figure 2. (We use T prefix before some of the terms for transactions to avoid confusion with similar terms that used for concurrent objects.) A transaction T is *committed* or *aborted* in a transaction history H if there is respectively a commit or abort response event for T in H. A *completed* transaction is either committed or aborted. A *live* transaction is a transaction that is not completed. A *pending* transaction has a pending event and a *commit-pending* transaction has a commit pending event. An *extension* of a history is obtained by committing or aborting its commit-pending transactions and aborting the other live transactions.

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 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 and hence the name. A *global write* is a write that is not local. The *writers of* i are the committed transactions that write to location i.

Markability is defined in Figure 3. A *marking* $\sqsubseteq$ of a transaction history is the union of the following relations on the set of transactions and the global reads.

- The *effect order*: The set of transactions is totally ordered by $\sqsubseteq$. In other words, $\sqsubseteq$ is total, antisymmetric and transitive on the set of transactions.
- The *access orders*: For each global read R from a location i, R and every writer of i are ordered by $\sqsubseteq$. In other words, $\sqsubseteq$ totally orders every global read R from a location i with respect to writers of i and is antisymmetric.

The *write-observation* property is comprised of the two properties: *local write-observation* and *global write-observation*. Local write-observation requires that every local read R from a location i returns the value written by the last write before it in the same transaction to i. Global write-observation requires that the value that every global read R from a location i returns is the value written by the global write of the last pre-accessor transaction to i. We remind that pre-accessors of R are the writers of i that are ordered before R in the access order and the last pre-accessor is the one that is greatest in the effect order.

The *Read-preservation* property requires that for every global read R from location i by transaction T, there is no writer transaction T' of i such that T' is marked between T and R (i.e. T' accesses i after R and takes effect before T), or similarly, T' is marked between R and T (i.e. T' takes effect after T and accesses i before R).

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

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

$$
\begin{array}{rcl}
\mathit{Committed}(H) & = & \{T \mid \exists l \in H \colon \mathit{obj}_H(l) = \mathit{mem} \ \wedge \ \mathit{trans}_H(l) = T \ \wedge \ \mathit{retv}_H(l) = \mathbb{C}\} \\
\mathit{Aborted}(H) & = & \{T \mid \exists l \in H \colon \mathit{obj}_H(l) = \mathit{mem} \ \wedge \ \mathit{trans}_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{CommitPending}(H) & = & \{T \in \mathit{Live}(H) \mid \exists l \in H \colon \mathit{obj}_H(l) = \mathit{mem} \ \wedge \ \mathit{trans}_H(l) = T \ \wedge \\
 & & \qquad \mathit{name}_H(l) = \mathit{commit}\} \\
\mathit{TExtension}(H) & = & \{H' \in \mathit{THistory} \mid \exists H'' \colon H' = H \cdot H'' \\
 & & \qquad \mathit{Trans}(H'') \subseteq \mathit{Trans}(H) \ \wedge \ \forall T \colon \|H''|T\| \leq 1 \ \wedge \\
 & & \qquad \mathit{Live}(H) \setminus \mathit{CommitPending}(H) \subseteq \mathit{Aborted}(H') \ \wedge \\
 & & \qquad \mathit{CommitPending}(H) \subseteq \mathit{Completed}(H')\} \\
\mathit{TReads}(H) & = & \{R \mid R \in H \ \wedge \ \mathit{obj}_H(R) = \mathit{mem} \ \wedge \ \mathit{name}_H(R) = \mathit{read} \ \wedge \ \mathit{retv}_H(R) \neq \mathbb{A}\} \\
\mathit{TWrites}(H) & = & \{W \mid W \in H \ \wedge \ \mathit{obj}_H(W) = \mathit{mem} \ \wedge \ \mathit{name}_H(W) = \mathit{write} \ \wedge \ \mathit{retv}_H(W) \neq \mathbb{A}\} \\
\mathit{LocalTReads}(H) & = & \{R \mid R \in \mathit{TReads}(H) \ \wedge \ \exists W \in \mathit{TWrites}(H) \colon \\
 & & \qquad \mathit{trans}_H(R) = \mathit{trans}_H(W) \ \wedge \ \mathit{arg1}_H(R) = \mathit{arg1}_H(W) \ \wedge \ W \prec_H R\} \\
\mathit{GlobalTReads}(H) & = & \mathit{TReads}(H) \setminus \mathit{LocalTReads}(H) \\
\mathit{LocalTWrites}(H) & = & \{W \mid W \in \mathit{TWrites}(H) \ \wedge \ \exists W' \in \mathit{TWrites}(H) \colon \\
 & & \qquad \mathit{trans}_H(W) = \mathit{trans}_H(W') \ \wedge \ \mathit{arg1}_H(W) = \mathit{arg1}_H(W') \ \wedge \ W \prec_H W'\} \\
\mathit{GlobalTWrites}(H) & = & \mathit{TWrites}(H) \setminus \mathit{LocalTWrites}(H) \\
\mathit{Writers}_H(i) & = & \{T \in \mathit{Trans}(H) \mid \exists l \in \mathit{TWrites}(H) \colon \mathit{arg1}_H(l) = i \ \wedge \\
 & & \qquad\qquad \mathit{trans}_H(l) = T \ \wedge \ T \in \mathit{Committed}(H)\}
\end{array}
$$

Figure 2: Basic Definitions

$$
\begin{array}{l}
\mathit{Marking}(H) = \{\sqsubseteq \ \mid \\
\qquad \forall T1, T2, T3 \in \mathit{Trans}(H)\colon \\
\qquad\quad (T1 \sqsubseteq T2 \ \vee \ T2 \sqsubseteq T1) \wedge \\
\qquad\quad (T1 \sqsubseteq T2 \ \wedge \ T2 \sqsubseteq T1) \Rightarrow (T1 = T2) \wedge \\
\qquad\quad (T1 \sqsubseteq T2) \ \wedge \ (T2 \sqsubseteq T3) \Rightarrow (T1 \sqsubseteq T3) \wedge \\
\qquad \forall R, T\colon \mathit{Let}\ i = \mathit{arg1}_H(R)\colon (R \in \mathit{GlobalTRead}(H) \ \wedge \ T \in \mathit{Writers}_H(i)) \Rightarrow \\
\qquad\quad (R \sqsubseteq T \ \vee \ T \sqsubseteq R) \wedge \\
\qquad\quad (R \sqsubseteq T \Rightarrow \neg T \sqsubseteq R) \ \wedge \ (T \sqsubseteq R \Rightarrow \neg R \sqsubseteq T)\} \\
\mathit{NoWriteBetween}_H(W, R) \Leftrightarrow \\
\qquad \forall W' \in \mathit{TWrites}(H)\colon W' \preceq_H W \ \vee \ R \prec_H W' \\
\mathit{LocalWriteObs}(H) \Leftrightarrow \\
\qquad \forall R \in \mathit{LocalTReads}(H)\colon \mathit{Let}\ T = \mathit{trans}_H(R), i = \mathit{arg1}_H(R), H' = H|T|i\colon \\
\qquad \exists W \in \mathit{TWrites}(H')\colon W \prec_{H'} R \ \wedge \ \mathit{NoWriteBetween}_{H'}(W, R) \ \wedge \ \mathit{retv}_{H'}(R) = \mathit{arg2}_{H'}(W) \\
\mathit{NoWriterBetween}_{H,i}(x, \sqsubseteq, x') \Leftrightarrow \\
\qquad \forall T \in \mathit{Writers}_H(i)\colon T \sqsubseteq x \ \vee \ x' \sqsubseteq T \\
\mathit{LastPreAccessor}_{H,\sqsubseteq}(T', R) \Leftrightarrow \mathit{Let}\ i = \mathit{arg1}_H(R), T = \mathit{trans}_H(R)\colon \\
\qquad T' \in \mathit{Writers}_H(i) \ \wedge \ T' \neq T \ \wedge \ T' \sqsubset R \ \wedge \ \mathit{NoWriterBetween}_{H,i}(T', \sqsubseteq, R) \\
\mathit{GlobalWriteObs}(H, \sqsubseteq) \Leftrightarrow \\
\qquad \forall R \in \mathit{GlobalTReads}(H)\colon \exists W \in \mathit{GlobalTWrites}(H)\colon \mathit{Let}\ T' = \mathit{trans}_H(W)\colon \\
\qquad \mathit{LastPreAccessor}_{H,\sqsubseteq}(T', R) \ \wedge \ \mathit{arg1}_H(R) = \mathit{arg1}_H(W) \ \wedge \ \mathit{retv}_H(R) = \mathit{arg2}_H(W) \\
\mathit{WriteObs}(H, \sqsubseteq) \Leftrightarrow \\
\qquad \mathit{LocalWriteObs}(H) \ \wedge \ \mathit{GlobalWriteObs}(H, \sqsubseteq) \\
\mathit{ReadPres}(H, \sqsubseteq) \Leftrightarrow \\
\qquad \forall R \in \mathit{GlobalTReads}(H)\colon \mathit{Let}\ i = \mathit{arg1}_H(R), T = \mathit{trans}_H(R)\colon \\
\qquad \mathit{NoWriterBetween}_{H,i}(R, \sqsubseteq, T) \ \wedge \ \mathit{NoWriterBetween}_{H,i}(T, \sqsubseteq, R) \\
\mathit{RealTimePres}(H, \sqsubseteq) \Leftrightarrow \\
\qquad \underline{\ll}_H \subseteq \sqsubseteq \\
\mathit{FinalStateMarkable} = \{H \in \mathit{THistory} \mid \exists H' \in \mathit{TExtension}(H)\colon \exists \sqsubseteq \in \mathit{Marking}(H')\colon \\
\qquad\qquad\qquad\qquad \mathit{ReadPres}(H', \sqsubseteq) \ \wedge \ \mathit{WriteObs}(H', \sqsubseteq) \ \wedge \ \mathit{RealTimePres}(H', \sqsubseteq)\}
\end{array}
$$

Figure 3: *FinalStateMarkable*

4 Marking Theorem

In this section, we prove the marking theorem.

For the sake of brevity, we use the shorthand notation
$\exists l = o.n_T(v_1){:}v_2 \in X$
for
$\exists l \in X\colon obj_X(l) = o \ \wedge \ name_X(l) = n \ \wedge \ trans_X(l) = T \ \wedge \ arg1_X(l) = v_1 \ \wedge \ retv_X(l) = v_2$
and similarly for universal quantification.

We also use W, R to denote labels.

Lemma 8 *For all $S \in TSequential$, $T \in S$, $S' = Visible(S,T)$, and $T', T'' \in S'$, we have $T' \mathrel{\underline{\prec\!\!\prec}}_{S'} T'' \iff T' \mathrel{\underline{\prec\!\!\prec}}_S T''$.*

Proof.

$$\begin{aligned}
& T' \mathrel{\underline{\prec\!\!\prec}}_{S'} T'' \\
\iff\ & S'|T' \lhd_{S'} S'|T'' \ \vee \ T' = T'' \\
\iff\ & S|T' \lhd_{S'} S|T'' \ \vee \ T' = T'' \\
\iff\ & S|T' \lhd_{S} S|T'' \ \vee \ T' = T'' \\
\iff\ & T' \mathrel{\underline{\prec\!\!\prec}}_S T''
\end{aligned}$$

In these four steps we apply:
1) the definition of $\mathrel{\underline{\prec\!\!\prec}}_{S'}$,
2) that the definition of $Visible(S,T)$ implies both $S'|T' = S|T'$ and $S'|T'' = S|T''$,
3) $S' \Subset S$, and
4) the definition of $\mathrel{\underline{\prec\!\!\prec}}_S$. □

Lemma 9 *For all $S \in TSequential$, $T \in S$, $i \in I$, $v, v' \in V$, $R = read_T(i){:}v \in GlobalReads(S)$, $S' = Visible(S,T)$, $T' \in S'$, and $W' = write_{T'}(i, v') \in GlobalWrites(S)$, we have*

$$NoWriteBetween_{(S'|i)}(W', R) \iff NoWriterBetween_{S,i}(T', \underline{\prec\!\!\prec}_S, T)$$

Proof.

$$\begin{aligned}
& NoWriteBetween_{(S'|i)}(W', R) \\
\iff\ & \forall W'' \in Writes(S'|i)\colon W'' \preceq_{(S'|i)} W' \vee R \preceq_{(S'|i)} W'' \\
\iff\ & \forall T'' \in S'|i\colon \forall i' \in I\colon \forall v'' \in V\colon \forall W'' = write_{T''}(i', v'') \in S'|i\colon W'' \preceq_{(S'|i)} W' \vee R \preceq_{(S'|i)} W'' \\
\iff\ & \forall T'' \in S'|i\colon \forall v'' \in V\colon \forall W'' = write_{T''}(i, v'') \in S'|i\colon W'' \preceq_{(S'|i)} W' \vee R \preceq_{(S'|i)} W'' \\
\iff\ & \forall T'' \in S'\colon \forall v'' \in V\colon \forall W'' = write_{T''}(i, v'') \in S'\colon W'' \preceq_{S'} W' \vee R \preceq_{S'} W'' \\
\iff\ & \forall T'' \in S'\colon \forall v'' \in V\colon \forall W'' = write_{T''}(i, v'') \in S'\colon T'' \underline{\prec\!\!\prec}_{S'} T' \vee T \underline{\prec\!\!\prec}_{S'} T'' \\
\iff\ & \forall T'' \in S'\colon \forall v'' \in V\colon \forall W'' = write_{T''}(i, v'') \in S'\colon T'' \underline{\prec\!\!\prec}_{S} T' \vee T \underline{\prec\!\!\prec}_{S} T'' \\
\iff\ & \forall T'' \in S'\colon \forall v'' \in V\colon \forall W'' = write_{T''}(i, v'') \in S'\colon T'' \prec\!\!\prec_S T \Rightarrow T'' \underline{\prec\!\!\prec}_{S} T' \\
\iff\ & \forall T'' \in S\colon \forall v'' \in V\colon \forall W'' = write_{T''}(i, v'') \in S\colon \\
& \qquad \big[[(T'' = T) \vee (T'' \prec\!\!\prec_S T \wedge T'' \in Committed(S))] \wedge [T'' \prec\!\!\prec_S T]\big] \Rightarrow T'' \underline{\prec\!\!\prec}_{S} T' \\
\iff\ & \forall T'' \in S\colon \forall v'' \in V\colon \forall W'' = write_{T''}(i, v'') \in S\colon \\
& \qquad (T'' \in Committed(S) \wedge\ T'' \prec\!\!\prec_S T) \Rightarrow T'' \underline{\prec\!\!\prec}_{S} T' \\
\iff\ & \forall T'' \in Writers_S(i)\colon T'' \prec\!\!\prec_S T \Rightarrow T'' \underline{\prec\!\!\prec}_{S} T' \\
\iff\ & \forall T'' \in Writers_S(i)\colon T'' \underline{\prec\!\!\prec}_{S} T' \vee T \underline{\prec\!\!\prec}_{S} T'' \\
\iff\ & NoWriterBetween_{S,i}(T', \underline{\prec\!\!\prec}_S, T)
\end{aligned}$$

In these twelve steps, we apply:
1) the definition of $NoWriteBetween$,
2) the definition of $Writes$,
3) the definition of projection $S'|i$,
4) R, W' and W'' access location i,
5) $S' \in TSequential$ and $R \in GlobalReads(S')$ and $W' \in GlobalWrites(S')$ (that are concluded from $S \in TSequential$, $R \in GlobalReads(S)$, $W' \in GlobalWrites(S)$ and $S' = Visible(S,T)$.),
6) Lemma 8,
7) Boolean logic and that $\underline{\prec\!\!\prec}_S$ is total,
8) the definition of $Visible$,
9) logical simplification,
10) the definition of $Writers$,
11) Boolean logic and that $\underline{\prec\!\!\prec}_S$ is total, and
12) the definition of $NoWriterBetween$. □

Lemma 10 $TSequential \subset Sequential$

Proof. Straightforward from definitions of $TSequential$, $THistory$ and $Sequential$. $\square$

Lemma 11 *$\forall i \in I$: $\forall v, v' \in V$: $\forall T, T' \in Trans$: if $R = read_T(i){:}v$, $W = write_{T'}(i, v)$, $W' = write_T(i, v')$, $S \in TSequential$, $W \prec_S R$, $NoWriteBetween_S(W, R)$ and $W' \prec_S R$, then $T = T'$.*

Proof. Suppose (1) $S \in TSequential$, (2) $W \prec_S R$, (3) $NoWriteBetween_S(W, R)$ and (4) $W' \prec_S R$. From [1] and Lemma 10, we have (5) $S \in Sequential$. From [4] and [5], we have (6) $\neg(R \prec_S W')$. From [3] we have (7) $W' \preceq_S W \vee R \prec_S W'$. From [6] and [7], we have (8) $W' \preceq_S W$. From [2] and [8], we have (9) $W' \preceq_S W \preceq_S R$. From [9], [1], and that W' and R are by T and W is by T', we have $T = T'$. $\square$

Lemma 12 *Suppose $S \in TSequential$. We have:*

$$\begin{aligned}
& \forall T \in S \colon \forall i \in I \colon \forall v \in V \colon \forall R = read_T(i){:}v \in LocalReads(S) \colon \\
& \qquad \exists T' \in Visible(S,T) \colon \exists W = write_{T'}(i,v) \in Visible(S,T) \colon \\
& \qquad W \prec_{(Visible(S,T) \mid i)} R \wedge NoWriteBetween_{(Visible(S,T) \mid i)}(W,R) \\
\iff & S \in LocalTSeqSpec
\end{aligned}$$

Proof. Suppose $S \in TSequential$. Thus, from Lemma 10, we have $S \in Sequential$. Let $S' = Visible(S,T)$. From $S \in TSequential$ and Lemma 8, we have $S' \in TSequential$. Thus, from Lemma 10, we have $S' \in Sequential$. From the definition of $Visible$, we have $S'|T = S|T$.

$$\begin{aligned}
& \forall T \in S \colon \forall i \in I \colon \forall v \in V \colon \forall R = read_T(i){:}v \in LocalReads(S) \colon \\
& \qquad \exists T' \in S' \colon \exists W = write_{T'}(i,v) \in S' \colon \\
& \qquad\qquad W \prec_{(S' \mid i)} R \wedge NoWriteBetween_{(S' \mid i)}(W,R) \\
\iff & \forall T \in S \colon \forall i \in I \colon \forall v \in V \colon \forall R = read_T(i){:}v \in LocalReads(S) \colon \\
& \qquad \exists v' \in V \colon \exists W' = write_T(i,v') \in S \colon W' \prec_S R \wedge \\
& \qquad \exists T' \in S' \colon \exists W = write_{T'}(i,v) \in S' \colon \\
& \qquad\qquad W \prec_{(S' \mid i)} R \wedge NoWriteBetween_{(S' \mid i)}(W,R) \\
\iff & \forall T \in S \colon \forall i \in I \colon \forall v \in V \colon \forall R = read_T(i){:}v \in LocalReads(S) \colon \\
& \qquad \exists v' \in V \colon \exists W' = write_T(i,v') \in S' \colon W' \prec_S R \wedge \\
& \qquad \exists T' \in S' \colon \exists W = write_{T'}(i,v) \in S' \colon \\
& \qquad\qquad W \prec_{(S' \mid i)} R \wedge NoWriteBetween_{(S' \mid i)}(W,R) \\
\iff & \forall T \in S \colon \forall i \in I \colon \forall v \in V \colon \forall R = read_T(i){:}v \in LocalReads(S) \colon \\
& \qquad \exists v' \in V \colon \exists W' = write_T(i,v') \in S' \colon W' \prec_{S'} R \wedge \\
& \qquad \exists T' \in S' \colon \exists W = write_{T'}(i,v) \in S' \colon \\
& \qquad\qquad W \prec_{(S' \mid i)} R \wedge NoWriteBetween_{(S' \mid i)}(W,R) \\
\iff & \forall T \in S \colon \forall i \in I \colon \forall v \in V \colon \forall R = read_T(i){:}v \in LocalReads(S) \colon \\
& \qquad \exists v' \in V \colon \exists W' = write_T(i,v') \in S' \colon W' \prec_{(S' \mid i)} R \wedge \\
& \qquad \exists T' \in S' \colon \exists W = write_{T'}(i,v) \in S' \colon \\
& \qquad\qquad W \prec_{(S' \mid i)} R \wedge NoWriteBetween_{(S' \mid i)}(W,R) \\
\iff & \forall T \in S \colon \forall i \in I \colon \forall v \in V \colon \forall R = read_T(i){:}v \in LocalReads(S) \colon \\
& \qquad \exists v' \in V \colon \exists W' = write_T(i,v') \in S' \colon W' \prec_{(S' \mid i)} R \wedge \\
& \qquad \exists W = write_T(i,v) \in S' \colon \\
& \qquad\qquad W \prec_{(S' \mid i)} R \wedge NoWriteBetween_{(S' \mid i)}(W,R) \\
\iff & \forall T \in S \colon \forall i \in I \colon \forall v \in V \colon \forall R = read_T(i){:}v \in LocalReads(S) \colon \\
& \qquad \exists W = write_T(i,v) \in S' \colon \\
& \qquad\qquad W \prec_{(S' \mid i)} R \wedge NoWriteBetween_{(S' \mid i)}(W,R) \\
\iff & \forall T \in S \colon \forall i \in I \colon \forall v \in V \colon \forall R = read_T(i){:}v \in LocalReads(S) \colon \\
& \qquad \exists W = write_T(i,v) \in S \colon \\
& \qquad\qquad W \prec_{(S' \mid i)} R \wedge NoWriteBetween_{(S' \mid i)}(W,R)
\end{aligned}$$

$$
\begin{aligned}
\iff\ & \forall T \in S \colon \forall i \in I \colon \forall v \in V \colon \forall R = \mathit{read}_T(i){:}v \in \mathit{LocalReads}(S) \colon \\
& \qquad \exists W = \mathit{write}_T(i,v) \in S \colon \\
& \qquad\qquad W \prec_{S'} R \ \wedge\ \mathit{NoWriteBetween}_{(S' \mid i)}(W,R) \\
\iff\ & \forall T \in S \colon \forall i \in I \colon \forall v \in V \colon \forall R = \mathit{read}_T(i){:}v \in \mathit{LocalReads}(S) \colon \\
& \qquad \exists W = \mathit{write}_T(i,v) \in S \colon \\
& \qquad\qquad W \prec_{S} R \ \wedge\ \mathit{NoWriteBetween}_{(S' \mid i)}(W,R) \\
\iff\ & \forall T \in S \colon \forall i \in I \colon \forall v \in V \colon \forall R = \mathit{read}_T(i){:}v \in \mathit{LocalReads}(S) \colon \\
& \qquad \exists W = \mathit{write}_T(i,v) \in S \colon \\
& \qquad\qquad W \prec_{S} R \ \wedge\ \forall W' \in \mathit{Writes}(S' \mid i) \colon W' \preceq_{(S' \mid i)} W \vee R \prec_{(S' \mid i)} W' \\
\iff\ & \forall T \in S \colon \forall i \in I \colon \forall v \in V \colon \forall R = \mathit{read}_T(i){:}v \in \mathit{LocalReads}(S) \colon \\
& \qquad \exists W = \mathit{write}_T(i,v) \in S \colon \\
& \qquad\qquad W \prec_{S} R \ \wedge\ \neg\exists W' \in \mathit{Writes}(S' \mid i) \colon \neg(W' \preceq_{(S' \mid i)} W) \ \wedge\ \neg(R \prec_{(S' \mid i)} W') \\
\iff\ & \forall T \in S \colon \forall i \in I \colon \forall v \in V \colon \forall R = \mathit{read}_T(i){:}v \in \mathit{LocalReads}(S) \colon \\
& \qquad \exists W = \mathit{write}_T(i,v) \in S \colon \\
& \qquad\qquad W \prec_{S} R \ \wedge\ \neg\exists W' \in \mathit{Writes}(S' \mid i) \colon W \prec_{(S' \mid i)} W' \prec_{(S' \mid i)} R \\
\iff\ & \forall T \in S \colon \forall i \in I \colon \forall v \in V \colon \forall R = \mathit{read}_T(i){:}v \in \mathit{LocalReads}(S) \colon \\
& \qquad \exists W = \mathit{write}_T(i,v) \in S \colon \\
& \qquad\qquad W \prec_{S} R \ \wedge\ \neg\exists v' \in V \colon \exists W' = \mathit{write}_T(i,v') \colon W \prec_{(S' \mid i)} W' \prec_{(S' \mid i)} R \\
\iff\ & \forall T \in S \colon \forall i \in I \colon \forall v \in V \colon \forall R = \mathit{read}_T(i){:}v \in \mathit{LocalReads}(S) \colon \\
& \qquad \exists W = \mathit{write}_T(i,v) \in S \colon \\
& \qquad\qquad W \prec_{S} R \ \wedge\ \neg\exists v' \in V : \exists W' = \mathit{write}_T(i,v') \colon W \prec_{(S \mid i)} W' \prec_{(S \mid i)} R \\
\iff\ & \forall T \in S \colon \forall i \in I \colon \forall v \in V \colon \forall R = \mathit{read}_T(i){:}v \in \mathit{LocalReads}(S) \colon \\
& \qquad \exists W = \mathit{write}_T(i,v) \in S \colon \\
& \qquad\qquad W \prec_{S} R \ \wedge\ \neg\exists W' \in \mathit{Writes}(S \mid i) \colon W \prec_{(S \mid i)} W' \prec_{(S \mid i)} R \\
\iff\ & \forall T \in S \colon \forall i \in I \colon \forall v \in V \colon \forall R = \mathit{read}_T(i){:}v \in \mathit{LocalReads}(S) \colon \\
& \qquad \exists W = \mathit{write}_T(i,v) \in S \colon \\
& \qquad\qquad W \prec_{S} R \ \wedge\ \forall W' \in \mathit{Writes}(S \mid i) \colon \neg(W \prec_{(S \mid i)} W') \ \vee\ \neg(W' \prec_{(S \mid i)} R) \\
\iff\ & \forall T \in S \colon \forall i \in I \colon \forall v \in V \colon \forall R = \mathit{read}_T(i){:}v \in \mathit{LocalReads}(S) \colon \\
& \qquad \exists W = \mathit{write}_T(i,v) \in S \colon W \prec_{S} R \ \wedge \\
& \qquad\qquad \forall W' \in \mathit{Writes}(S \mid i) \colon W' \preceq_{(S \mid i)} W \ \vee\ R \prec_{(S \mid i)} W' \\
\iff\ & \forall T \in S \colon \forall i \in I \colon \forall v \in V \colon \forall R = \mathit{read}_T(i){:}v \in \mathit{LocalReads}(S) \colon \\
& \qquad \exists W = \mathit{write}_T(i,v) \in S|T|i \colon W \prec_{S|T|i} R \ \wedge \\
& \qquad\qquad \forall W' \in \mathit{Writes}(S|T|i) \colon W' \preceq_{(S|T|i)} W \ \vee\ R \prec_{(S|T|i)} W' \\
\iff\ & \forall T \in S \colon \forall i \in I \colon \forall v \in V \colon \forall R = \mathit{read}_T(i){:}v \in \mathit{LocalReads}(S) \colon \\
& \qquad \exists W = \mathit{write}_T(i,v) \in S|T|i \colon \\
& \qquad\qquad W \prec_{S|T|i} R \ \wedge\ \mathit{NoWriteBetween}_{(S|T|i)}(W,R) \\
\iff\ & S \in \mathit{LocalTSeqSpec}
\end{aligned}
$$

In these twenty steps, we apply: 1) the definition of *LocalReads*,

2) the definition of $Visible$,
3) $S'|T = S|T$ and that both W' and R are by T,
4) that both W' and R are on i,
5) Lemma 11,
6) duplicate conjunction,
7) the definition of $Visible$,
8) that both R and W are on i,
9) $S'|T = S|T$ and that both R and W are by T,
10) the definition of $NoWriteBetween$,
11) first-order logic,
12) $(S' \mid i) \in Sequential$,
13) from $(S' \mid i) \in TSequential$, R and W are by transaction T and W' is between them, we have W' is by T,
14) $S'|T = S|T$,
15) from $(S \mid i) \in TSequential$, R and W are by transaction T and W' is between them, we have W' is by T.
16) first-order logic,
17) $(S \mid i) \in Sequential$,
18) $(S \mid i) \in Sequential$, $trans_H(R) = trans_H(W) = T$ and $arg1_H(R) = arg1_H(W) = i$,
19) the definition of $NoWriteBetween$,
20) the definition of $LocalTSeqSpec$.

□

Lemma 13 *Suppose* $S \in \mathit{TSequential} \cap \mathit{TComplete}$*. We have:*

$$\begin{aligned}
& S \in \mathit{TSeqSpec} \\
\iff\ & S \in \mathit{LocalTSeqSpec}\ \wedge \\
& \forall T \in S \colon \forall i \in I \colon \forall v \in V \colon \forall R = \mathit{read}_T(i){:}v \in \mathit{GlobalReads}(S) \colon \\
& \exists T' \in \mathit{Committed}(S) \colon \exists W = \mathit{write}_{T'}(i, v) \in \mathit{GlobalWrites}(S) \colon \\
& \quad (T' \prec\!\!\prec_S T)\ \wedge\ \mathit{NoWriterBetween}_{S,i}(T', \underline{\prec\!\!\prec}_S, T)
\end{aligned}$$

Proof. Suppose $S \in \mathit{TSequential} \cap \mathit{TComplete}$. From $S \in \mathit{TSequential}$ and Lemma 8, we have $\mathit{Visible}(S, T) \in \mathit{TSequential}$.

$$\begin{aligned}
& S \in \mathit{TSeqSpec} \\
\iff\ & \forall T \in S \colon \forall i \in I \colon (\mathit{Visible}(S, T) \mid i) \in \mathit{SeqSpec}(i) \\
\iff\ & \forall T \in S \colon \forall i \in I \colon \\
& \qquad \forall T'' \in (\mathit{Visible}(S, T) \mid i) \colon \forall v \in V \colon \forall R = \mathit{read}_{T''}(i){:}v \in (\mathit{Visible}(S, T) \mid i) \colon \\
& \qquad \exists T' \in (\mathit{Visible}(S, T) \mid i) \colon \exists W = \mathit{write}_{T'}(i, v) \in (\mathit{Visible}(S, T) \mid i) \colon \\
& \qquad W \prec_{(\mathit{Visible}(S,T)\ \mid\ i)} R\ \wedge\ \mathit{NoWriteBetween}_{(\mathit{Visible}(S,T)\ \mid\ i)}(W, R) \\
\iff\ & \forall T \in S \colon \forall i \in I \colon \\
& \qquad \forall T'' \in \mathit{Visible}(S, T) \colon \forall v \in V \colon \forall R = \mathit{read}_{T''}(i){:}v \in \mathit{Visible}(S, T) \colon \\
& \qquad \exists T' \in \mathit{Visible}(S, T) \colon \exists W = \mathit{write}_{T'}(i, v) \in \mathit{Visible}(S, T) \colon \\
& \qquad W \prec_{(\mathit{Visible}(S,T)\ \mid\ i)} R\ \wedge\ \mathit{NoWriteBetween}_{(\mathit{Visible}(S,T)\ \mid\ i)}(W, R) \\
\iff\ & \forall T \in S \colon \forall i \in I \colon \forall v \in V \colon \forall R = \mathit{read}_T(i){:}v \in S \colon \\
& \qquad \exists T' \in \mathit{Visible}(S, T) \colon \exists W = \mathit{write}_{T'}(i, v) \in \mathit{Visible}(S, T) \colon \\
& \qquad W \prec_{(\mathit{Visible}(S,T)\ \mid\ i)} R\ \wedge\ \mathit{NoWriteBetween}_{(\mathit{Visible}(S,T)\ \mid\ i)}(W, R) \\
\iff\ & \forall T \in S \colon \forall i \in I \colon \forall v \in V \colon \forall R = \mathit{read}_T(i){:}v \in \mathit{LocalReads}(S) \colon \\
& \qquad \exists T' \in \mathit{Visible}(S, T) \colon \exists W = \mathit{write}_{T'}(i, v) \in \mathit{Visible}(S, T) \colon \\
& \qquad W \prec_{(\mathit{Visible}(S,T)\ \mid\ i)} R\ \wedge\ \mathit{NoWriteBetween}_{(\mathit{Visible}(S,T)\ \mid\ i)}(W, R) \\
& \wedge \\
& \forall T \in S \colon \forall i \in I \colon \forall v \in V \colon \forall R = \mathit{read}_T(i){:}v \in \mathit{GlobalReads}(S) \colon \\
& \qquad \exists T' \in \mathit{Visible}(S, T) \colon \exists W = \mathit{write}_{T'}(i, v) \in \mathit{Visible}(S, T) \colon \\
& \qquad W \prec_{(\mathit{Visible}(S,T)\ \mid\ i)} R\ \wedge\ \mathit{NoWriteBetween}_{(\mathit{Visible}(S,T)\ \mid\ i)}(W, R) \\
\iff\ & S \in \mathit{LocalTSeqSpec}\ \wedge \\
& \forall T \in S \colon \forall i \in I \colon \forall v \in V \colon \forall R = \mathit{read}_T(i){:}v \in \mathit{GlobalReads}(S) \colon \\
& \qquad \exists T' \in \mathit{Visible}(S, T) \colon \exists W = \mathit{write}_{T'}(i, v) \in \mathit{Visible}(S, T) \colon \\
& \qquad W \prec_{(\mathit{Visible}(S,T)\ \mid\ i)} R\ \wedge\ \mathit{NoWriteBetween}_{(\mathit{Visible}(S,T)\ \mid\ i)}(W, R)
\end{aligned}$$

$$
\begin{array}{ll}
\iff & S \in \mathit{LocalTSeqSpec}\ \wedge \\
& \forall T \in S\colon \forall i \in I\colon \forall v \in V\colon \forall R = \mathit{read}_T(i){:}v \in \mathit{GlobalReads}(S)\colon \\
& \qquad \exists T' \in \mathit{Visible}(S,T)\colon \exists W = \mathit{write}_{T'}(i,v) \in \mathit{Visible}(S,T)\colon \\
& \qquad W \prec_{\mathit{Visible}(S,T)} R\ \wedge\ \mathit{NoWriteBetween}_{(\mathit{Visible}(S,T)\ |\ i)}(W,R) \\
\iff & S \in \mathit{LocalTSeqSpec}\ \wedge \\
& \forall T \in S\colon \forall i \in I\colon \forall v \in V\colon \forall R = \mathit{read}_T(i){:}v \in \mathit{GlobalReads}(S)\colon \\
& \qquad \exists T' \in \mathit{Visible}(S,T)\colon \exists W = \mathit{write}_{T'}(i,v) \in \mathit{Visible}(S,T)\colon \\
& \qquad T' \prec\!\!\prec_{\mathit{Visible}(S,T)} T\ \wedge\ \mathit{NoWriteBetween}_{(\mathit{Visible}(S,T)\ |\ i)}(W,R) \\
\iff & S \in \mathit{LocalTSeqSpec}\ \wedge \\
& \forall T \in S\colon \forall i \in I\colon \forall v \in V\colon \forall R = \mathit{read}_T(i){:}v \in \mathit{GlobalReads}(S)\colon \\
& \qquad \exists T' \in \mathit{Visible}(S,T)\colon \exists W = \mathit{write}_{T'}(i,v) \in \mathit{Visible}(S,T)\colon \\
& \qquad T' \prec\!\!\prec_S T\ \wedge\ \mathit{NoWriteBetween}_{(\mathit{Visible}(S,T)\ |\ i)}(W,R) \\
\iff & S \in \mathit{LocalTSeqSpec}\ \wedge \\
& \forall T \in S\colon \forall i \in I\colon \forall v \in V\colon \forall R = \mathit{read}_T(i){:}v \in \mathit{GlobalReads}(S)\colon \\
& \qquad \exists T' \in \mathit{Visible}(S,T)\colon \exists W = \mathit{write}_{T'}(i,v) \in \mathit{GlobalWrites}(S)\colon \\
& \qquad T' \prec\!\!\prec_S T\ \wedge\ \mathit{NoWriteBetween}_{(\mathit{Visible}(S,T)\ |\ i)}(W,R) \\
\iff & S \in \mathit{LocalTSeqSpec}\ \wedge \\
& \forall T \in S\colon \forall i \in I\colon \forall v \in V\colon \forall R = \mathit{read}_T(i){:}v \in \mathit{GlobalReads}(S)\colon \\
& \qquad \exists T' \in \mathit{Visible}(S,T)\colon \exists W = \mathit{write}_{T'}(i,v) \in \mathit{GlobalWrites}(S)\colon \\
& \qquad T' \prec\!\!\prec_S T\ \wedge\ \mathit{NoWriterBetween}_{S,i}(T', \underline{\prec\!\!\prec}_S, T) \\
\iff & S \in \mathit{LocalTSeqSpec}\ \wedge \\
& \forall T \in S\colon \forall i \in I\colon \forall v \in V\colon \forall R = \mathit{read}_T(i){:}v \in \mathit{GlobalReads}(S)\colon \\
& \qquad \exists T' \in \mathit{Visible}(S,T)\colon \exists W = \mathit{write}_{T'}(i,v) \in \mathit{GlobalWrites}(S)\colon \\
& \qquad (T' \prec\!\!\prec_S T)\ \wedge\ T' \in \mathit{Committed}(S)\ \wedge\ \mathit{NoWriterBetween}_{S,i}(T', \underline{\prec\!\!\prec}_S, T) \\
\iff & S \in \mathit{LocalTSeqSpec}\ \wedge \\
& \forall T \in S\colon \forall i \in I\colon \forall v \in V\colon \forall R = \mathit{read}_T(i){:}v \in \mathit{GlobalReads}(S)\colon \\
& \qquad \exists T' \in \mathit{Committed}(S)\colon \exists W = \mathit{write}_{T'}(i,v) \in \mathit{GlobalWrites}(S)\colon \\
& \qquad (T' \prec\!\!\prec_S T)\ \wedge\ \mathit{NoWriterBetween}_{S,i}(T', \underline{\prec\!\!\prec}_S, T)
\end{array}
$$

In these thirteen steps, we apply:
1) the definition of $\mathit{TSeqSpec}$ and $S \in \mathit{TSequential} \cap \mathit{TComplete}$,
2) the definition of $\mathit{SeqSpec}(i)$,
3) R and W access location i,
4) that we can choose $T'' = T$,
5) $\mathit{Reads}(S) = \mathit{LocalReads}(S) \cup \mathit{GlobalReads}(S)$,
6) Lemma 12,
7) that R and W are both on location i
8) that R and W are by transactions T and T' respectively, $\mathit{Visible}(S,T) \in \mathit{TSequential}$, and $R \in \mathit{GlobalReads}(\mathit{Visible}(S,T))$ (because $R \in \mathit{GlobalReads}(R)$ and $\mathit{Visible}(S,T)|T = S|T$),
9) Lemma 8,
10) $T' \prec\!\!\prec_S T$ and $\mathit{NoWriteBetween}_{(\mathit{Visible}(S,T)\ |\ i)}(W,R)$,
11) Lemma 9,

12) $T' \in Visible(S,T)$ and $(T' \lll_S T)$, and
13) the definition of $Visible(S,T)$. □

Lemma 14 (Invariance) *If* $H \equiv H'$*, then* $Marking(H) = Marking(H')$ *and* $ReadPres(H) = ReadPres(H')$ *and* $WriteObs(H) = WriteObs(H')$*.*

Proof. Immediate from the definitions of $Marking$, $ReadPres$, and $WriteObs$. $\square$

Lemma 15 $\forall H \in THistory\colon \forall \sqsubseteq \in Marking(H)\colon \exists S \in TSequential\colon H \equiv S \ \wedge\ \underline{\prec\!\!\!\prec}_H \subseteq \underline{\prec\!\!\!\prec}_S \ \wedge\ \underline{\prec\!\!\!\prec}_S \subseteq \sqsubseteq$.

Proof. Let $H \in THistory$ and let $\sqsubseteq \in Marking(H)$. We have that $\sqsubseteq$ is a total order of $Trans$ so we can choose a permutation π on $1..n$ such that $\forall i, j \in 1..n\colon (i < j) \Leftrightarrow (T_{\pi(i)} \sqsubset T_{\pi(j)})$. Define: $S = H|T_{\pi(1)}, \ldots, H|T_{\pi(n)}$. It is straightforward to prove that $S \in TSequential \ \wedge\ H \equiv S \ \wedge\ \underline{\prec\!\!\!\prec}_H \subseteq \underline{\prec\!\!\!\prec}_S \ \wedge\ \underline{\prec\!\!\!\prec}_S \subseteq \sqsubseteq$. $\square$

Lemma 16 *Suppose* $\sqsubseteq \in Marking(H) \ \wedge\ p_2 \notin Writers_H(i)$*.*
If $NoWriterBetween_{H,i}(T_1, \sqsubseteq, p_2)$ *and* $NoWriterBetween_{H,i}(p_2, \sqsubseteq, T_3)$*,*
then $NoWriterBetween_{H,i}(T_1, \sqsubseteq, T_3)$*.*

Proof.

$$
\begin{aligned}
& NoWriterBetween_{H,i}(T_1, \sqsubseteq, p_2) \ \wedge\ NoWriterBetween_{H,i}(p_2, \sqsubseteq, T_3) \\
\iff\ & \forall T \in Writers_H(i)\colon (T \sqsubseteq T_1 \ \vee\ p_2 \sqsubseteq T) \ \wedge\ (T \sqsubseteq p_2 \ \vee\ T_3 \sqsubseteq T) \\
\iff\ & \forall T \in Writers_H(i)\colon (T \sqsubseteq T_1 \ \wedge\ (T \sqsubseteq p_2 \ \vee\ T_3 \sqsubseteq T)) \ \vee \\
& \qquad\qquad\qquad\qquad (p_2 \sqsubseteq T \ \wedge\ T \sqsubseteq p_2) \ \vee\ (p_2 \sqsubseteq T \ \wedge\ T_3 \sqsubseteq T) \\
\implies\ & \forall T \in Writers_H(i)\colon (T \sqsubseteq T_1) \ \vee\ (T_3 \sqsubseteq T) \\
\iff\ & NoWriterBetween_{H,i}(T_1, \sqsubseteq, T_3)
\end{aligned}
$$

The first step uses the definition of $NoWriterBetween$. The second step uses $\wedge$ distribution over $\vee$. The third step simplifies the first disjunct using conjunction elimination, eliminates the second disjunct using $p_2 \notin Writers_H(i)$ and simplifies the third disjunct using conjunction elimination. The fourth step uses the definition of $NoWriterBetween$. $\square$

Lemma 17 *Suppose $S \in TSequential \cap TComplete$. We have:*

$$S \in TSeqSpec \Longleftrightarrow S \in FinalStateMarkable \tag{4.1}$$

Proof. Let $S \in TSequential \cap TComplete$. From Lemma 13, the definition of $FinalStateMarkable$, and $S \in TComplete$, we have that we must prove:

$$\begin{aligned}
& S \in LocalTSeqSpec \ \wedge \\
& \forall T \in S\colon \forall i \in I\colon \forall v \in V\colon \forall R = read_T(i){:}v \in GlobalReads(S)\colon \\
& \exists T' \in Committed(S)\colon \exists W = write_{T'}(i, v) \in GlobalWrites(S)\colon \\
& \quad (T' \prec\!\!\prec_S T) \ \wedge \ NoWriterBetween_{S,i}(T', \underline{\prec\!\!\prec}_S, T) \\
\Longleftrightarrow \quad & \exists \sqsubseteq \in Marking(S)\colon \ \ \underline{\prec\!\!\prec}_S \subseteq \sqsubseteq \ \ \wedge \ \sqsubseteq \in ReadPres(S) \wedge \ \sqsubseteq \in WriteObs(S)
\end{aligned}$$

From the definition of $WriteObs$ and $LastPreAccessor$ we have that:

$$\begin{aligned}
& \sqsubseteq \in WriteObs(S) \\
\Longleftrightarrow \quad & S \in LocalTSeqSpec \ \wedge \\
& \forall T \in Trans\colon \forall i \in I\colon \forall v \in V\colon \forall R = read_T(i){:}v \in GlobalReads(S)\colon \\
& \exists T' \in Trans\colon \exists W = write_{T'}(i, v) \in GlobalWrites(S)\colon \\
& \quad T' \in Writers_S(i) \ \wedge \ T' \neq T \ \wedge \ T' \sqsubset R \ \wedge \ NoWriterBetween_{S,i}(T', \sqsubseteq, R) \\
\Longleftrightarrow \quad & S \in LocalTSeqSpec \ \wedge \\
& \forall T \in Trans\colon \forall i \in I\colon \forall v \in V\colon \forall R = read_T(i){:}v \in GlobalReads(S)\colon \\
& \exists T' \in Trans\colon \exists W = write_{T'}(i, v) \in GlobalWrites(S)\colon \\
& \quad T' \in Committed(S) \ \wedge \ T' \neq T \ \wedge \ T' \sqsubset R \ \wedge \ NoWriterBetween_{S,i}(T', \sqsubseteq, R)
\end{aligned}$$

We are now ready to prove the two directions of the equivalence.
⇒:
Assume that

$$\begin{aligned}
& S \in LocalTSeqSpec \ \wedge \\
& \forall T \in S\colon \forall i \in I\colon \forall v \in V\colon \forall R = read_T(i){:}v \in GlobalReads(S)\colon \\
& \exists T' \in Committed(S)\colon \exists W = write_{T'}(i, v) \in GlobalWrites(S)\colon \\
& \quad (T' \prec\!\!\prec_S T) \ \wedge \ NoWriterBetween_{S,i}(T', \underline{\prec\!\!\prec}_S, T)
\end{aligned}$$

Define:

$$\begin{aligned}
p_1 \sqsubset p_2 \quad \Longleftrightarrow \quad & (p_1 \prec\!\!\prec_S p_2) \ \vee \\
& (trans_S(p_1) \ \underline{\prec\!\!\prec}_S \ p_2) \ \vee \\
& (p_1 \ \underline{\prec\!\!\prec}_S \ trans_S(p_2)) \\
p_1 \sqsubseteq p_2 \quad \Longleftrightarrow \quad & p_1 \sqsubset \ \vee \ p_2 p_1 = p_2
\end{aligned}$$

We show that

$$\begin{aligned}
& \sqsubseteq \in Marking(S) \ \wedge \\
& \underline{\prec\!\!\prec}_S \subseteq \sqsubseteq \ \ \wedge \ \sqsubseteq \in ReadPres(S) \ \wedge \\
& S \in LocalTSeqSpec \ \wedge \\
& \forall T \in Trans\colon \forall i \in I\colon \forall v \in V\colon \forall R = read_T(i){:}v \in GlobalReads(S)\colon \\
& \exists T' \in Trans\colon \exists W = write_{T'}(i, v) \in GlobalWrites(S)\colon \\
& \quad T' \in Committed(S) \ \wedge \ T' \neq T \ \wedge \ T' \sqsubset R \ \wedge \ NoWriterBetween_{S,i}(T', \sqsubseteq, R)
\end{aligned}$$

It is straightforward to prove $\sqsubseteq \in \mathit{Marking}(S)$ and $\preceq_S \subseteq \sqsubseteq$, $\sqsubseteq \in \mathit{ReadPres}(S)$. Additionally, the first conjunct of $\mathit{WriteObs}(S)$ (that is, $S \in \mathit{LocalTSeqSpec}$) is immediate from the assumption. So, we still need to prove the second conjunct of $\mathit{WriteObs}(S)$.

Let $T \in \mathit{Trans}$, $i \in I$, $v \in V$, $R = \mathit{read}_T(i){:}v \in \mathit{GlobalReads}(S)$. From the assumption (the left-hand side), we have that we can find (1) $T' \in \mathit{Committed}(S)$ and (2) $W = \mathit{write}_{T'}(i, v) \in \mathit{GlobalWrites}(S)$ such that (3) $(T' \prec_S T)$ and (4) $\mathit{NoWriterBetween}_{S,i}(T', \preceq_S, T)$. Let us now prove each conjunct of $T' \neq T \ \wedge\ T' \sqsubseteq R \ \wedge\ \mathit{NoWriterBetween}_{S,i}(T', \sqsubseteq, R)$ in turn.

From [3] and that $\preceq_S$ is a total order of $\mathit{Trans}(S)$, we have (5) $T' \neq T$. From [3] and the definition of $\sqsubseteq$, we have $T' \sqsubseteq R$. From [4] and $\preceq_S \subseteq \sqsubseteq$, we have (6) $\mathit{NoWriterBetween}_{S,i}(T', \sqsubseteq, T)$. From $T \preceq_S T$ and the definition of $\sqsubseteq$, we have (7) $R \sqsubseteq T$. From [6], [7] and the definition of $\sqsubseteq$ and transitivity of $\preceq_S$, we have $\mathit{NoWriterBetween}_{S,i}(T', \sqsubseteq, R)$.

$\Leftarrow$:

Assume the right-hand side and choose $\sqsubseteq \in \mathit{Marking}(S)$ such that:

$$
\begin{aligned}
&\preceq_S \subseteq \sqsubseteq \ \ \wedge\ \sqsubseteq \in \mathit{ReadPres}(S)\ \wedge \\
&S \in \mathit{TLocalSeqSpec}\ \wedge \\
&\forall T \in \mathit{Trans}\colon \forall i \in I\colon \forall v \in V\colon \forall R = \mathit{read}_T(i){:}v \in \mathit{GlobalReads}(S)\colon \\
&\exists T' \in \mathit{Committed}(S)\colon \exists\ W = \mathit{write}_{T'}(i, v) \in \mathit{GlobalWrites}(S)\colon \\
&\qquad T' \neq T \ \wedge\ T' \sqsubseteq R \ \wedge\ \mathit{NoWriterBetween}_{S,i}(T', \sqsubseteq, R)
\end{aligned}
$$

We show that

$$
\begin{aligned}
&S \in \mathit{LocalTSeqSpec}\ \wedge \\
&\forall T \in S\colon \forall i \in I\colon \forall v \in V\colon \forall R = \mathit{read}_T(i){:}v \in \mathit{GlobalReads}(S)\colon \\
&\exists T' \in \mathit{Committed}(S)\colon \exists W = \mathit{write}_{T'}(i, v) \in \mathit{GlobalWrites}(S)\colon \\
&\qquad (T' \prec_S T)\ \wedge\ \mathit{NoWriterBetween}_{S,i}(T', \preceq_S, T)
\end{aligned}
$$

The first conjunct (of the left-hand side), $S \in \mathit{LocalTSeqSpec}$, is immediate from the assumption. From the assumption we have (1) $\preceq_S \subseteq \sqsubseteq$, (2) $\sqsubseteq \in \mathit{ReadPres}(S)$. Let $T \in \mathit{Trans}$, $i \in I$, $v \in V$, $R = \mathit{read}_T(i){:}v \in \mathit{GlobalReads}(S)$. From the above property of $\sqsubseteq$, we have that we can find (3) $T' \in \mathit{Committed}(S)$ and (4) $W = \mathit{write}_{T'}(i, v) \in \mathit{GlobalWrites}(S)$ such that (5) $T' \neq T$ and (6) $T' \sqsubseteq R$ and (7) $\mathit{NoWriterBetween}_{S,i}(T', \sqsubseteq, R)$. From [1], that $\sqsubseteq$ is a total order on $\mathit{Trans}(S)$ ($\sqsubseteq \in \mathit{Marking}(S)$), and that $\preceq_S$ is a total order on $\mathit{Trans}(S)$ ($S \in \mathit{TSequential}$), we have (8) $\forall T, T' \in \mathit{Trans}\colon T' \sqsubseteq T \Rightarrow T' \preceq_S T$.

First we prove $T' \prec_S T$. From [2] ,we have (9) $\mathit{NoWriterBetween}_{S,i}(T, \sqsubseteq, R)$. From [3] and [4], we have (10) $T' \in \mathit{Writers}_S(i)$. From [9] and [10], we have (11) $T' \sqsubseteq T \ \vee\ R \sqsubseteq T'$. From [6], $T' \neq R$ and $\sqsubseteq$ is a total order on $\{R\} \cup \mathit{Writers}_S(i)$ ($\sqsubseteq \in \mathit{Marking}(S)$), we have (12) $R \not\sqsubseteq T'$. From [11] and [12], we have (13) $T' \sqsubseteq T$. From [8] and [13], we have (14) $T' \preceq_S T$. From [14] and [5], we have $T' \prec_S T$.

Second, we prove $\mathit{NoWriterBetween}_{S,i}(T', \preceq_S, T)$. From [2], we have (15) $\mathit{NoWriterBetween}_{S,i}(R, \sqsubseteq, T)$. From $R \notin \mathit{Writers}_S(i)$, [7], [15], and Lemma 16, we have (16) $\mathit{NoWriterBetween}_{S,i}(T', \sqsubseteq, T)$. From [16] and [8] we have $\mathit{NoWriterBetween}_{S,i}(T', \preceq_S, T)$. □

Theorem 18 (Marking) $\mathit{FinalStateOpaque} = \mathit{FinalStateMarkable}$.

Proof.

$$
\begin{aligned}
& \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{\prec\!\!\prec}_{H'} \subseteq \underline{\prec\!\!\prec}_S \wedge S \in \mathit{TSeqSpec}\} \\
= \;& \{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{\prec\!\!\prec}_{H'} \subseteq \underline{\prec\!\!\prec}_S \wedge S \in \mathit{FinalStateMarkable}\} \\
= \;& \{H \in \mathit{THistory} \mid \exists H' \in \mathit{TExtension}(H)\colon \exists S \in \mathit{TSequential}\colon H' \equiv S \wedge \underline{\prec\!\!\prec}_{H'} \subseteq \underline{\prec\!\!\prec}_S \wedge \\
& \quad \exists \sqsubseteq \in \mathit{Marking}(S)\colon \underline{\prec\!\!\prec}_S \subseteq \sqsubseteq \wedge \sqsubseteq \in \mathit{ReadPres}(S) \cap \mathit{WriteObs}(S)\} \\
= \;& \{H \in \mathit{THistory} \mid \exists H' \in \mathit{TExtension}(H)\colon \exists S \in \mathit{TSequential}\colon H' \equiv S \wedge \underline{\prec\!\!\prec}_{H'} \subseteq \underline{\prec\!\!\prec}_S \wedge \\
& \quad \exists \sqsubseteq \in \mathit{Marking}(H')\colon \underline{\prec\!\!\prec}_S \subseteq \sqsubseteq \wedge \sqsubseteq \in \mathit{ReadPres}(H') \cap \mathit{WriteObs}(H')\} \\
= \;& \{H \in \mathit{THistory} \mid \exists H' \in \mathit{TExtension}(H)\colon \exists \sqsubseteq \in \mathit{Marking}(H')\colon \\
& \quad \sqsubseteq \in \mathit{ReadPres}(H') \cap \mathit{WriteObs}(H') \wedge \\
& \quad \exists S \in \mathit{TSequential}\colon H' \equiv S \wedge \underline{\prec\!\!\prec}_{H'} \subseteq \underline{\prec\!\!\prec}_S \wedge \underline{\prec\!\!\prec}_S \subseteq \sqsubseteq\} \\
= \;& \{H \in \mathit{THistory} \mid \exists H' \in \mathit{TExtension}(H)\colon \exists \sqsubseteq \in \mathit{Marking}(H')\colon \\
& \quad \underline{\prec\!\!\prec}_{H'} \subseteq \sqsubseteq \wedge \sqsubseteq \in \mathit{ReadPres}(H') \cap \mathit{WriteObs}(H') \wedge \\
& \quad \exists S \in \mathit{TSequential}\colon H' \equiv S \wedge \underline{\prec\!\!\prec}_{H'} \subseteq \underline{\prec\!\!\prec}_S \wedge \underline{\prec\!\!\prec}_S \subseteq \sqsubseteq\} \\
= \;& \{H \in \mathit{THistory} \mid \exists H' \in \mathit{TExtension}(H)\colon \exists \sqsubseteq \in \mathit{Marking}(H')\colon \\
& \quad \underline{\prec\!\!\prec}_{H'} \subseteq \sqsubseteq \wedge \sqsubseteq \in \mathit{ReadPres}(H') \cap \mathit{WriteObs}(H')\} \\
= \;& \mathit{Markable}
\end{aligned}
$$

In these eight steps we apply:
1) the definition of $\mathit{FinalStateOpaque}$,
2) Lemma 17 and $S \in \mathit{TComplete}$ (because $H' \in \mathit{TExtension}(H)$ and $H' \equiv S$),
3) the definition of $\mathit{FinalStateMarkable}$ and $S \in \mathit{TComplete}$,
4) Lemma 14,
5) logical rearrangement,
6) transitivity of $\subseteq$,
7) Lemma 15, and
8) the definition of $\mathit{FinalStateMarkable}$. □

5 Synchronization Object Types

In this subsection, we first define the semantics of basic and linearizable objects. Then, we define the interface and the sequential specifications of the following abstract object types: register, lock, try-lock, counter, set and map. For each abstract object type, we define concrete synchronization object types. We define the following synchronization object types: basic register, atomic register, atomic cas register, lock, try-lock, strong counter, basic set and basic map. For each synchronization object type, we present lemmas that characterize the properties of its execution histories. Please see the end of this section for notes on the proof of the lemmas that we present in this subsection.[1]

Basic and Linearizable Object Types

The abstract type of each object o specifies the sequential specification of o, denoted by $\mathit{SeqSpec}(o)$, that is the prefix-closed set of correct sequential histories of o. In the following subsections, we will consider several synchronization object types and define their sequential specifications.

We consider two concurrent types: basic and linearizable. Linearizable objects comply with their sequential specification in every concurrent execution. Basic objects, on the other hand, comply with their sequential specification if they are accessed sequentially.

Definition 1 (Basic Object Semantics) *Every sequential execution on a basic object is an execution in its sequential specification. The semantics of a basic object o, $\mathbb{H}_B(o)$, is a set of histories that is constrained as follows:*

$$\mathbb{H}_B(o) \cap \mathit{Sequential} \subseteq \mathit{SeqSpec}(o) \tag{5.1}$$

Definition 2 (Linearizable Object Semantics) *An execution history X is linearizable for an object o iff there is an indistinguishable sequential history L that is in the sequential specification of o and is real-time-preserving. L is a linearization and $\prec_L$ is a linearization order of X. The semantics of a linearizable object o, $\mathbb{H}_L(o)$, is defined as the following set of execution and linearization pairs.*

$$\mathbb{H}_L(o) = \{(X, L) \mid X \equiv L \ \wedge \ L \in \mathit{SeqSpec}(o) \wedge \ \prec_X \subseteq \prec_L\} \tag{5.2}$$

Lemma 19 (X2L) *For every linearization L of an execution history X on object o and method calls l and l', if $l \prec_X l'$ then $l \prec_L l'$.*

Lemma 20 (LASym) *For every linearization L of an execution history X on object o and method calls l and l', if $l \prec_L l'$ then $\neg(l' \prec_L l) \ \wedge \ \neg(l = l')$*

Lemma 21 (LTrans) *For every linearization L of an execution history X on object o and method calls l, l', and l'', if $l \prec_L l'$ and $l' \prec_L l''$ then $l \prec_L l''$.*

Lemma 22 (LTotal) *For every linearization L of an execution history X on object o and method calls l and l', if $l \in X$ and $l' \in X$ then $(l \prec_L l') \ \vee \ (l' \prec_L l) \ \vee \ (l = l')$*

Lemma 23 (L2X) *For every linearization L of an execution history X on object o and method calls l and l', if $(l \prec_L l')$ then $l \in X$, $l' \in X$, and l and l' are both on o.*

Lemma 24 (XLTrans) *For every linearization L of an execution history X on object o and method calls l_1, l_2, l_3, and l_4, if $l_1 \prec_X l_2$, $l_2 \prec_L l_3$, $l_3 \prec_X l_4$, then $l_1 \prec_X l_4$*

---

[1] In this subsection, we use $\forall$ and $\exists$ as a notational convenience. $\forall l \colon p$ can be rewritten as $\bigwedge_{(l \in Labels(X))} p(X)$ and $\exists l \colon p$ can be rewritten as $\bigvee_{(l \in Labels(X))} p(X)$.

Register
Register. A register *reg* is an object that encapsulates a value and supports *read* and *write* methods. The method call *reg.read*() returns the current encapsulated value of *reg*. The method call *reg.write*(v) overwrites the encapsulated value of *reg* with v.

Definition 3 *The sequential specification of register reg is the set of sequential histories of read and write method calls on reg where every read returns the argument of the latest preceding write (regardless of thread identifiers). (Note that it is assumed that a write method call initializes the register before other methods are invoked.) The sequential specification of a register r, SeqSpec*(r)*, is defined as follows:*

$$\begin{aligned}
isXRead_{X,r}(l_R) &= l_R \in X \ \wedge \ obj_X(l_R) = r \ \wedge \ name_X(l_R) = read && (5.3)\\
isXWrite_{X,r}(l_W) &= l_W \in X \ \wedge \ obj_X(l_W) = r \ \wedge \ name_X(l_W) = write && (5.4)\\
NoWriteBetween_{X,r}(l_W, l_R) &= \forall l'_W\colon isXWrite_{X,r}(l'_W) \Rightarrow (l'_W \preceq_X l_W \ \vee \ l_R \prec_X l'_W) && (5.5)\\
isXWriter_{X,r}(l_W, l_R) &= isXWrite_{X,r}(l_W) \ \wedge && (5.6)\\
&\quad\ l_W \prec_X l_R \ \wedge \\
&\quad\ NoWriteBetween_{X,r}(l_W, l_R)\\
Legal(r) &= \{S \mid \forall l_R\colon isXRead_{S,r}(l_R) \Rightarrow && (5.7)\\
&\qquad \exists l_W\colon isXWriter_{S,r}(l_W, l_R) \ \wedge \\
&\qquad\qquad retv_S(l_R) = arg1_S(l_W)\}\\
SeqSpec(r) &= \{S \mid S|r = S \ \wedge \ S \in Sequential \cap Legal(r)\} && (5.8)
\end{aligned}$$

Basic Register. A basic register is a basic instance of the register type.

Let *BasicRegister* denote the type of basic registers.

Lemma 25 *In every sequential execution on a basic register, every read reads the value that the latest preceding write writes. Formally,*

$$\begin{aligned}
&\forall reg \in BasicRegister\colon \forall X \in \mathbb{H}_B(reg)\colon X \in Sequential \Rightarrow && (5.9)\\
&\forall l_R\colon isXRead_{X,reg}(l_R) \Rightarrow\\
&\exists l_W\colon isXWriter_{X,reg}(l_W, l_R) \ \wedge\\
&\qquad retv_X(l_R) = arg1_X(l_W)
\end{aligned}$$

Two concurrent read method calls on a register do not conflict. Thus, basic registers can maintain consistency even when the execution involves concurrent read method calls. Let us define

$$\begin{aligned}
isXRaceFree_{X,r}(l) &= \forall l_w\colon isXWrite_{X,r}(l_w) \Rightarrow l_w \preceq_X l \ \vee \ l \prec_X l_w && (5.10)\\
isXSequentiallyWritten_r(X) &= \forall l \in X\colon isXWrite_{X,r}(l) \Rightarrow isRaceFree_{X,r}(l) && (5.11)
\end{aligned}$$

A method call is race-free if an only if there is no write method call that executes concurrent to it. An execution is sequentially-written if and only if every pair of write method calls on it are ordered in the execution order or in other words, every write method call on it is race-free.

Definition 4 (Basic Register Semantics) *An execution history on a basic register is in the semantics of the basic register if and only if it is not sequentially-written or it is sequentially-written and every race-free* *read reads the value that the latest preceding write writes. The semantics of a basic register r, $\mathbb{H}_B(r)$, is defined as follows.*

$$\begin{aligned}\mathbb{H}_B(r) \;=\; \{X \mid\; & X|o = X \;\wedge \\ & isXSequentiallyWritten_r(X) \Rightarrow \\ & \forall l_r\colon isXRead_{X,r}(l_r) \;\wedge\; isXRaceFree_{X,r}(l_r) \Rightarrow \\ & \exists l_w\colon isXWriter_{X,r}(l_w, l_r) \;\wedge \\ & \quad retv_X(l_r) = arg1_X(l_w)\,\}\end{aligned} \tag{5.12}$$

Note that if an execution is not sequentially-written, reads may return arbitrary values. Similarly, racy reads may return arbitrary values.

Note that this definition satisfies the constraint of Definition 1.

Lemma 26 (BReg) *In every sequentially-written execution on a basic register, every race-free read reads the value that the latest preceding write writes. Formally,*

$$\begin{aligned}& \forall reg \in BasicRegister\colon \forall X \in \mathbb{H}_B(reg)\colon isXSequentiallyWritten_r(X) \Rightarrow \\ & \forall l_R\colon isXRead_{X,reg}(l_R) \;\wedge\; isXRaceFree_{X,r}(l_R) \Rightarrow \\ & \exists l_W\colon isXWriter_{X,reg}(l_W, l_R) \;\wedge \\ & \quad retv_X(l_R) = arg1_X(l_W)\end{aligned} \tag{5.13}$$

Atomic Register. An atomic register is a linearizable instance of the register type.

Let *AtomicRegister* denote the type of atomic registers.

Let us define

$$LNoWriteBetween_{X,L,r}(l_W, l_R) \;=\; \forall l'_W\colon isXWrite_{X,r}(l'_W) \Rightarrow (l'_W \preceq_L l_W \;\vee\; l_R \prec_L l'_W) \tag{5.14}$$

$$\begin{aligned}isLWriter_{X,L,r}(l_W, l_R) \;=\; & isXWrite_{X,r}(l_W) \;\wedge \\ & l_W \prec_L l_R \;\wedge \\ & LNoWriteBetween_{X,L,r}(l_W, l_R)\end{aligned} \tag{5.15}$$

Lemma 27 (AReg) *In every execution on an atomic register, every read reads the value written by the last write linearized before it. Formally,*

$$\begin{aligned}& \forall r \in AtomicRegister\colon \forall (X, L) \in \mathbb{H}_L(r)\colon \\ & \forall l_R\colon isXRead_{X,r}(l_R) \Rightarrow \\ & \exists l_W\colon isLWriter_{X,L,r}(l_W, l_R) \;\wedge \\ & \quad retv_X(l_R) = arg1_X(l_W)\end{aligned} \tag{5.16}$$

CAS (Compare-And-Swap) Register

A CAS register is an object that encapsulates a value and supports the *cas* method in addition to *read* and *write* methods. The method call $r.cas(v_1, v_2)$ updates the value of the register to v_2 and returns *true* if the current value of the register is v_1. It returns $false$ otherwise.

A *successful write* is either a *write* method call or a successful *cas* method call. The *written value* of a successful write is its first argument, if it is a *write* method call or is its second argument, if it is a *cas* method call.

Definition 5 *The sequential specification of cas register reg is the set of sequential histories of read, write and cas method calls on reg with the following two conditions. Every read returns the written value of the latest preceding successful write (regardless of thread identifiers). (Note that it is assumed that a write method call initializes the register before other methods are invoked.) Every cas with the first argument v_1 returns true if the written value of the latest preceding successful write is v_1 and returns $false$ otherwise.*

Atomic CAS Register. An atomic CAS register is a linearizable instance of CAS register type.

Let $AtomicCASRegister$ denote the type of Atomic CAS registers.

Let us define

$$\begin{aligned}
isXCAS_{X,r}(l_W) &= l_W \in X \ \wedge \ obj_X(l_W) = r \ \wedge \ name_X(l_W) = cas && (5.17)\\
isXCWrite_{X,r}(l_W) &= isXWrite(l_W) \ \vee \ (isXCAS(l_W) \ \wedge \ retv_X(l_W) = true) && (5.18)\\
writtenValue_X(l_W) &= \begin{cases} arg1_X(l_W) & \text{if } name_X(l_W) = write \\ arg2_X(l_W) & \text{if } name_X(l_W) = cas \end{cases} && (5.19)\\
LNoWriteBetween_{X,L,r}(l_W, l_R) &= \forall l'_W : isXCWrite_{X,r}(l'_W) \Rightarrow (l'_W \preceq_L l_W \ \vee \ l_R \prec_L l'_W) && (5.20)\\
isLCWriter_{X,L,r}(l_W, l_R) &= isXCWrite_{X,r}(l_W) \ \wedge && (5.21)\\
&\quad\ l_W \prec_L l_R \ \wedge \\
&\quad\ LNoWriteBetween_{X,L,r}(l_W, l_R)
\end{aligned}$$

Lemma 28 (CASRegRead) *In every execution on an atomic cas register, every read returns the value the last successful write linearized before it writes. Formally,*

$$\begin{aligned}
&\forall r \in AtomicCASRegister \colon \forall (X, L) \in \mathbb{H}_L(r) \colon && (5.22)\\
&\forall l_R \colon isXRead_{X,r}(l_R) \Rightarrow \\
&\exists l_W \colon isLCWriter_{X,L,r}(l_W, l_R) \ \wedge \\
&\qquad retv_X(l_R) = arg1_X(l_W)
\end{aligned}$$

Lemma 29 (CASRegCAS) *In every execution on an atomic cas register, every cas returns true if its first argument is equal to the argument of the last successful write linearized before it and returns $false$ otherwise. Formally,*

$$\begin{aligned}
&\forall reg \in AtomicCASRegister \colon \forall (X, Reg) \in \mathbb{H}_L(reg) \colon && (5.23)\\
&\forall l_C, l_W \colon \\
&\quad isXCAS_{X,reg}(l_C) \ \wedge \\
&\quad isLCWriter_{X,Reg,reg}(l_W, l_R) \\
&\Rightarrow \\
&\quad (writtenValue_X(l_W) = arg1_X(l_C) \Rightarrow retv_X(l_C) = true) \ \wedge \\
&\quad (\neg(writtenValue_X(l_W) = arg1_X(l_C)) \Rightarrow retv_X(l_C) = false)
\end{aligned}$$

Lock

Abstract lock. An abstract lock l is an object that encapsulates a state, acquired $\mathbb{A}$ or released $\mathbb{R}$, and supports the following methods: *lock*: The method call $l.lock()$ changes the state from $\mathbb{R}$ to $\mathbb{A}$. *unlock*: The method call $l.unlock()$ changes the state from $\mathbb{A}$ to $\mathbb{R}$. *read*: The method call $l.read()$ returns $true$ if the state of *lock* is $\mathbb{A}$ and $false$ otherwise. The method calls *lock* and *unlock* are mutating method calls. The method call *read* is an accessor method call.

Definition 6 *The sequential specification of a lock l is the set of sequential histories L of lock, unlock, and read method calls on l where the sub-history of L for mutating methods is an alternating sequence of lock and unlock methods and every read method call in L returns true if the last mutating method call before it in L is a lock and returns false otherwise.*

Lock. A lock is a linearizable instance of the abstract lock type.

Let *Lock* denote the type of locks.

Now, we present some preliminary definitions and then lemmas about locks.

$$\begin{aligned}&isXLock_{X,lo}(l) = \\ &\qquad l \in X \ \wedge \ obj_X(l) = lo \ \wedge \ name_X(l) = lock\end{aligned} \tag{5.24}$$

$$\begin{aligned}&isXUnlock_{X,lo}(l) = \\ &\qquad l \in X \ \wedge \ obj_X(l) = lo \ \wedge \ name_X(l) = unlock\end{aligned} \tag{5.25}$$

$$\begin{aligned}&isXRead_{X,lo}(l) = \\ &\qquad l \in X \ \wedge \ obj_X(l) = lo \ \wedge \ name_X(l) = read\end{aligned} \tag{5.26}$$

The common usage protocol for locks is that a thread unlocks a lock only if it has already acquired it. Many languages including Java enforce this property of programs by runtime checks. We capture this property as follows.

Definition 7 *A history is owner-respecting for a lock if every thread in the history releases the lock only after it has already acquired it.*

$$\begin{aligned}&isXOwnerRespecting_{lo}(X) = \\ &\quad \forall l\colon isXUnlock_{X,lo}(l) \Rightarrow \\ &\quad \exists l'\colon isXLock_{X,lo}(l') \wedge \\ &\qquad thread_X(l') = thread_X(l) \wedge \\ &\qquad l' \prec_X l \wedge \\ &\qquad \forall l''\colon (isXUnLock_{X,lo}(l'') \ \wedge \ thread_X(l'') = thread_X(l)) \Rightarrow (l'' \prec_X l' \ \vee \ l \preceq_X l'')\end{aligned} \tag{5.27}$$

Lemma 30 *If l is a lock, X is an owner-respecting history of l and L is the linearization of X, then the sub-history of L for mutating method calls is a sequence of pairs of lock and unlock method calls by the same thread (possibly followed by a lock method call).*

Lemma 31 (Lock) *In an owner-respecting execution for a lock l, if a lock method call by a thread T_1 is linearized before an unlock method call by a thread T_2, then an unlock method call by T_1 is linearized before a lock method call by T_2. Formally,*

$$\begin{aligned}&\forall o \in Lock\colon \forall (X, L) \in \mathbb{H}_L(o)\colon \forall l_{l1}, l_{u2}\colon \\ &\qquad (isXOwnerRespecting_o(X) \ \wedge \\ &\qquad isXLock_{X,o}(l_{l1}) \ \wedge \\ &\qquad isXUnlock_{X,o}(l_{u2}) \ \wedge \\ &\qquad l_{l1} \prec_L l_{u2}) \Rightarrow \\ &\exists l_{u1}, l_{l2}\colon \\ &\qquad isXUnlock_{X,o}(l_{u1}) \ \wedge \ thread_X(l_{l1}) = thread_X(l_{u1}) \ \wedge \\ &\qquad isXLock_{X,o}(l_{l2}) \ \wedge \ thread_X(l_{l2}) = thread_X(l_{u2}) \ \wedge \\ &\qquad l_{u1} \prec_L l_{l2}\end{aligned} \tag{5.28}$$

Lemma 32 (LockReadL) *In an owner-respecting execution for a lock l, if a read method call that returns $false$ is linearized before an unlock method call by a thread T, then the read method call is linearized before a lock method call by T. Formally,*

$$\begin{aligned}
&\forall o \in \mathit{Lock}\colon \forall (X, L) \in \mathbb{H}_L(o)\colon \forall l_{u1}, l_{r2}\colon \\
&\qquad (\mathit{isXOwnerRespecting}_o(X) \ \wedge \\
&\qquad \mathit{isXRead}_{X,o}(l_{r2}) \ \wedge \ \mathit{retv}_X(l_{r2}) = \mathit{false} \\
&\qquad \mathit{isXUnlock}_{X,o}(l_{u1}) \ \wedge \\
&\qquad l_{r2} \prec_L l_{u1}) \Rightarrow \\
&\exists l_{l1}\colon \\
&\qquad \mathit{isXLock}_{X,o}(l_{l1}) \ \wedge \ \mathit{thread}_X(l_{l1}) = \mathit{thread}_X(l_{u1}) \ \wedge \\
&\qquad l_{r2} \prec_L l_{l1}
\end{aligned} \tag{5.29}$$

Lemma 33 (LockReadR) *In an owner-respecting execution for a lock l, if a lock method call by a thread T is linearized before a read method call that returns $false$, then an unlock method call by T is linearized before the read method call. Formally,*

$$\begin{aligned}
&\forall o \in \mathit{Lock}\colon \forall (X, L) \in \mathbb{H}_L(o)\colon \forall l_{l1}, l_{r2}\colon \\
&\qquad (\mathit{isXOwnerRespecting}_o(X) \ \wedge \\
&\qquad \mathit{isXLock}_{X,o}(l_{l1}) \ \wedge \\
&\qquad \mathit{isXRead}_{X,o}(l_{r2}) \ \wedge \ \mathit{retv}_X(l_{r2}) = \mathit{false} \\
&\qquad l_{l1} \prec_L l_{r2}) \Rightarrow \\
&\exists l_{u1}\colon \\
&\qquad \mathit{isXUnlock}_{X,o}(l_{u1}) \ \wedge \ \mathit{thread}_X(l_{l1}) = \mathit{thread}_X(l_{u1}) \ \wedge \\
&\qquad l_{u1} \prec_L l_{r2}
\end{aligned} \tag{5.30}$$

Lemma 34 (LockReadM) *In an owner-respecting execution for a lock l, every read method call that is linearized between a pair of matching lock and unlock method calls returns true. Formally,*

$$\begin{aligned}
&\forall o \in \mathit{Lock}\colon \forall (X, L) \in \mathbb{H}_L(o)\colon \forall l_{l1}, l_{u1}, l_{r2}\colon \\
&\qquad (\mathit{isXOwnerRespecting}_o(X) \ \wedge \\
&\qquad \mathit{isXLock}_{X,o}(l_{l1}) \ \wedge \\
&\qquad \mathit{isXUnlock}_{X,o}(l_{u1}) \ \wedge \\
&\qquad \mathit{thread}_X(l_{l1}) = \mathit{thread}_X(l_{u1}) \ \wedge \\
&\qquad \forall l'_{u1}\colon (\mathit{isXUnlock}_{X,o}(l'_{u1}) \ \wedge \ \mathit{thread}_X(l_{l1}) = \mathit{thread}_X(l'_{u1})) \Rightarrow (l'_{u1} \prec_X l_{l1} \ \vee \ l_{u1} \preceq_X l'_{u1}) \\
&\qquad \mathit{isXRead}_{X,o}(l_{r2}) \ \wedge \\
&\qquad l_{l1} \prec_L l_{r2} \ \wedge \ l_{r2} \prec_L l_{u1}) \\
&\Rightarrow \\
&\qquad \mathit{retv}_X(l_{r2}) = \mathit{true}
\end{aligned} \tag{5.31}$$

Try-Lock

Abstract Try-lock. A try-lock l is an object that encapsulates an abstract state, acquired $\mathbb{A}$ or released $\mathbb{R}$, and in addition to *lock*, *unlock* and *read* methods, it supports the *trylock* method. If the state of the *lock* is $\mathbb{R}$, $l.\mathit{trylock}()$ changes it to $\mathbb{A}$ and returns *true*. Otherwise, it returns $false$.

We call a *lock* method call or a successful *tryLock* method call, a *successful lock* method call. We call a *lock* method call, successful *tryLock* method call or *unlock* method call, a *mutating* method call.

Definition 8 *The sequential specification of a try-lock l is the set of sequential histories L of lock, unlock, read and tryLock method calls on l with the following conditions: The last mutating method call before a successful lock method call is an unlock method call. Similarly, the last mutating method call before an unlock method call is a successful lock method call. A tryLock method call returns true if the latest preceding mutating method call is an unlock and returns $false$ otherwise. Similarly, A read method call returns true if the latest preceding mutating method call is a successful lock and returns $false$ otherwise.*

Try-Lock. A try-lock is a linearizable instance of the abstract try-lock type.

Let $TryLock$ denote the type of try-locks.

Similar to the *Lock* type, after some preliminary definitions, we define the owner-respecting histories and state the $TryLock$ type lemmas.

$$isXTryLock_{X,o}(l) = \qquad l \in X \ \wedge \ obj_X(l) = o \ \wedge \ name_X(l) = tryLock \tag{5.32}$$

$$isXTLock_{X,o}(l) = \qquad isXLock_{X,o}(l) \ \vee \ (isXTryLock_{X,o}(l) \ \wedge \ retv_X(l) = true) \tag{5.33}$$

The intuition for owner-respecting histories remains the same. A history is owner-respecting for a try-lock if every thread in the history releases the lock only after it has already acquired it. The minor difference from the prior definition for locks is that the acquisition of a try-lock is either by a *lock* method call or a successful *tryLock* method call.

$$\begin{aligned}
&isXTOwnerRespecting_o(X) = \\
&\quad \forall l\colon isXUnlock_{X,o}(l) \Rightarrow \\
&\quad \exists l'\colon isXTLock_{X,o}(l') \wedge \\
&\qquad thread_X(l') = thread_X(l) \wedge \\
&\qquad l' \prec_X l \wedge \\
&\qquad \forall l''\colon (isXUnLock_{X,o}(l'') \ \wedge \ thread_X(l'') = thread_X(l)) \Rightarrow l'' \prec_X l' \ \vee \ l \preceq_X l''
\end{aligned} \tag{5.34}$$

Lemma 35 *If l is a try-lock, X is an owner-respecting history of l and L is the linearization of X, then the sub-history of L for mutating method calls is a sequence of pairs of successful lock and unlock method calls by the same thread (possibly followed by a successful lock method call).*

Lemma 36 (TryLock) *In an owner-respecting execution for a try-lock l, if a successful lock method call by a thread T_1 is linearized before an unlock method call by a thread T_2, then an unlock method call by T_1*

is linearized before a successful lock method call by T_2. Formally,

$$\begin{aligned}
&\forall o \in \mathit{TryLock}\colon \forall (X, L) \in \mathbb{H}_L(o)\colon \forall l_{l1}, l_{u2}\colon \\
&\qquad (\mathit{isXTOwnerRespecting}_o(X) \ \wedge \\
&\qquad \mathit{isXTLock}_{X,o}(l_{l1}) \ \wedge \\
&\qquad \mathit{isXUnlock}_{X,o}(l_{u2}) \ \wedge \\
&\qquad l_{l1} \prec_L l_{u2}) \Rightarrow \\
&\exists l_{u1}, l_{l2}\colon \\
&\qquad \mathit{isXUnlock}_{X,o}(l_{u1}) \ \wedge \ \mathit{thread}_X(l_{l1}) = \mathit{thread}_X(l_{u1}) \ \wedge \\
&\qquad \mathit{isXTLock}_{X,o}(l_{l2}) \ \wedge \ \mathit{thread}_X(l_{l2}) = \mathit{thread}_X(l_{u2}) \ \wedge \\
&\qquad l_{u1} \prec_L l_{l2}
\end{aligned} \tag{5.35}$$

Lemma 37 (TryLockReadL) *In an owner-respecting execution for a try-lock l, a read method call that returns $false$ is linearized before if an unlock method call by a thread T then the read method call is linearized before a successful lock method call by T. Formally,*

$$\begin{aligned}
&\forall o \in \mathit{TryLock}\colon \forall (X, L) \in \mathbb{H}_L(o)\colon \forall l_{u1}, l_{r2}\colon \\
&\qquad (\mathit{isXTOwnerRespecting}_o(X) \ \wedge \\
&\qquad \mathit{isXRead}_{X,o}(l_{r2}) \ \wedge \ \mathit{retv}_X(l_{r2}) = \mathit{false} \\
&\qquad \mathit{isXUnlock}_{X,o}(l_{u1}) \ \wedge \\
&\qquad l_{r2} \prec_L l_{u1}) \Rightarrow \\
&\exists l_{l1}\colon \\
&\qquad \mathit{isXTLock}_{X,o}(l_{l1}) \ \wedge \ \mathit{thread}_X(l_{l1}) = \mathit{thread}_X(l_{u1}) \ \wedge \\
&\qquad l_{r2} \prec_L l_{l1}
\end{aligned} \tag{5.36}$$

Lemma 38 (TryLockReadR) *In an owner-respecting execution for a try-lock l, if a successful lock method call by a thread T is linearized before a read method call that returns $false$, then an unlock method call by T is linearized before the read method call. Formally,*

$$\begin{aligned}
&\forall o \in \mathit{TryLock}\colon \forall (X, L) \in \mathbb{H}_L(o)\colon \forall l_{l1}, l_{r2}\colon \\
&\qquad (\mathit{isXTOwnerRespecting}_o(X) \ \wedge \\
&\qquad \mathit{isXTLock}_{X,o}(l_{l1}) \ \wedge \\
&\qquad \mathit{isXRead}_{X,o}(l_{r2}) \ \wedge \ \mathit{retv}_X(l_{r2}) = \mathit{false} \\
&\qquad l_{l1} \prec_L l_{r2}) \Rightarrow \\
&\exists l_{u1}\colon \\
&\qquad \mathit{isXUnlock}_{X,o}(l_{u1}) \ \wedge \ \mathit{thread}_X(l_{l1}) = \mathit{thread}_X(l_{u1}) \ \wedge \\
&\qquad l_{u1} \prec_L l_{r2}
\end{aligned} \tag{5.37}$$

Lemma 39 (TryLockReadM) *In an owner-respecting execution for a try-lock l, every read method call* *that is linearized between a pair of matching successful and unlock method calls returns true. Formally,*

$$
\begin{aligned}
&\forall o \in \mathit{TryLock}\colon \forall (X, L) \in \mathbb{H}_L(o)\colon \forall l_{l1}, l_{u1}, l_{r2}\colon \\
&\quad (\mathit{isXOwnerRespecting}_o(X) \ \wedge \\
&\quad \mathit{isXTLock}_{X,o}(l_{l1}) \ \wedge \\
&\quad \mathit{isXUnlock}_{X,o}(l_{u1}) \ \wedge \\
&\quad \mathit{thread}_X(l_{l1}) = \mathit{thread}_X(l_{u1}) \ \wedge \\
&\quad \forall l'_{u1}\colon (\mathit{isXUnlock}_{X,o}(l'_{u1}) \ \wedge \ \mathit{thread}_X(l_{l1}) = \mathit{thread}_X(l'_{u1})) \Rightarrow (l'_{u1} \prec_X l_{l1} \ \vee \ l_{u1} \preceq_X l'_{u1}) \\
&\quad \mathit{isXRead}_{X,o}(l_{r2}) \ \wedge \\
&\quad l_{l1} \prec_L l_{r2} \ \wedge \ l_{r2} \prec_L l_{u1}) \\
&\Rightarrow \\
&\quad \mathit{retv}_X(l_{r2}) = \mathit{true}
\end{aligned}
\tag{5.38}
$$

Seq-Lock

Abstract seq-lock. A seq-lock l is an object that encapsulates a number and an abstract state, acquired $\mathbb{A}$ or released $\mathbb{R}$. It supports the *read*, *compareAndLock* and *incAndUnlock* methods. The method call *l.read*() returns the pair of the encapsulated number and *true* if the state of *lock* is $\mathbb{A}$ and *false* otherwise. The method call *l.compareAndLock*(n) compares the the encapsulated number with n and if they are equal, changes the state from $\mathbb{R}$ to $\mathbb{A}$ and returns *true*. Otherwise, it does not change the state of the seq-lock and returns *false*. The method call *l.incAndUnlock*() increments the encapsulated number and changes the state from $\mathbb{A}$ to $\mathbb{R}$.

A successful *compareAndLock* and *incAndUnlock* are mutating method calls. The method call *read* is an accessor method call.

Definition 9 *The sequential specification of a seq-lock l is the set of sequential histories L of read, compareAndLock, and incAndUnlock method calls on l with the following conditions:*

Every read method call returns the pair of the number of incAndUnlock method calls before it and true if the last mutating method call before it is a successful compareAndLock and false otherwise.

A compareAndLock method call returns true if the last mutating method call before it is an incAndUnlock method call and the number of incAndUnlock method calls before it is equal to its argument. It returns false otherwise.

The last mutating method call before an incAndUnlock method call is a successful compareAndLock method call.

Seq-Lock. A seq-lock is a linearizable instance of the abstract seq-lock type.

Let *SeqLock* denote the type of seq-locks.

Counter

Abstract Counter: A counter c is an object that encapsulates a number and supports the following two methods: The method call *c.read*() returns the current value of c. The method call *c.iaf*() increments the value of c and returns the incremented value.

Definition 10 *The sequential specification of a counter c is the set of sequential histories of read and iaf method calls on c where every method call returns the number of iaf method calls before it (including the method call itself). Note that it is assumed that the initial value of the counter is zero.*

Strong Counter. A strong counter is a linearizable instance of abstract counter type.

Let $SCounter$ denote the type of strong counters.

Lemma 40 (SCounter) *The return value of every method call that is linearized before an iaf method call is smaller than the return value of the iaf method call. Formally,*

$$\begin{aligned}&\forall c \in SCounter\colon \forall (X, C) \in \mathbb{H}_L(c)\colon \forall l, l'\colon \\ &\qquad l \in X \ \wedge\ l' \in X \ \wedge\ name_X(l') = iaf \ \wedge\ l \prec_C l' \\ &\Rightarrow \\ &\qquad retv_X(l) < retv_X(l')\end{aligned} \tag{5.39}$$

Set

A set s is an object that represents a set of values and supports the following methods: *add*: The method call $s.add(v)$ adds value v to set s. *contains*: The method call $s.containts(v)$ returns $true$ if v is a member of s and $false$ otherwise.

Definition 11 *The sequential specification of a set s is the set of sequential histories of add and contains method calls on s where every contains method call returns true if there is a preceding add method call with the same argument, and returns $false$ otherwise. Note that it is assumed that the set is initially empty.*

Basic Set. A basic set is a basic instance of set type.

Let $BasicSet$ denote the type of basic sets.

Let us define

$$\begin{aligned}&isXContains_{X,s}(l) = \\ &\qquad l \in X \ \wedge\ obj_X(l) = s \ \wedge\ name_X(l) = contains\end{aligned} \tag{5.40}$$

$$\begin{aligned}&isXAdd_{X,s}(l) = \\ &\qquad l \in X \ \wedge\ obj_X(l) = s \ \wedge\ name_X(l) = add\end{aligned} \tag{5.41}$$

Lemma 41 (BasicSetContains) *In every sequential execution on a basic set, for every contains method call that returns true, there is a preceding add method call with the same argument. Formally,*

$$\begin{aligned}&\forall s \in BasicSet\colon \forall X \in \mathbb{H}_B(s)\colon X \in Sequential \Rightarrow \\ &\forall l_c\colon isXContains_{X,s}(l_c) \ \wedge\ retv_X(l_c) = true \Rightarrow \\ &\exists l_a\colon isXAdd_{X,s}(l_a) \ \wedge \\ &\qquad arg1(l_a) = arg1(l_c) \ \wedge\ l_a \prec_X l_c\end{aligned} \tag{5.42}$$

Lemma 42 (BasicSetAdd) *In every sequential execution on a basic set, every contains method call that succeeds an add method call with the same argument returns true. Formally,*

$$\begin{aligned}&\forall s \in BasicSet\colon \forall X \in \mathbb{H}_B(s)\colon X \in Sequential \Rightarrow \\ &\forall l_c, l_a\colon \\ &\qquad isXContains_{X,s}(l_c) \ \wedge \\ &\qquad isXAdd_{X,s}(l_a) \ \wedge \\ &\qquad arg1(l_a) = arg1(l_c) \ \wedge\ l_a \prec_X l_c \\ &\Rightarrow \\ &\qquad retv_X(l_c) = true\end{aligned} \tag{5.43}$$

Map
A map m is an object that represents a mapping from a set of keys to a set of values and supports the following methods: *put*: The method call $m.put(k, v)$ adds or updates the mapping of the key k to the value v ($v \neq \bot$) in the map m. *get*: The method call $m.get(k)$ returns the value that the map m associates with the key k. It returns $\bot$ if m does not map k.

Definition 12 *The sequential specification of a map m is the set of sequential histories of put and get method calls on m where every get method call returns $\bot$ if there is no preceding put method call with the same key argument; otherwise it returns the second argument of the latest preceding put method call with the same key argument. Note that it is assumed that the map is initially empty.*

Basic Map. A basic set is a basic instance of map type.

Let $BasicMap$ denote the type of basic maps.

Let us define

$$isXGet_{X,m}(l) = \tag{5.44}$$
$$l \in X \ \wedge \ obj_X(l) = m \ \wedge \ name_X(l) = get$$
$$isXPut_{X,m}(l) = \tag{5.45}$$
$$l \in X \ \wedge \ obj_X(l) = m \ \wedge \ name_X(l) = put$$
$$isXPutter_{X,m}(l_p, l_g) \Leftrightarrow \tag{5.46}$$
$$isXPut_{X,m}(l_p) \ \wedge \ arg1_X(l_p) = arg1_X(l_g) \ \wedge \ l_p \prec_X l_g \wedge \tag{5.47}$$
$$\forall l'_p \colon isXPut_{X,m}(l'_p) \ \wedge \ arg1_X(l'_p) = arg1_X(l_g) \Rightarrow (l'_p \preceq_X l_p \ \vee \ l_g \prec_X l'_p) \tag{5.48}$$

Lemma 43 (BasicMapGet) *In every sequential execution on a basic map, the return value of every get method call that does not return $\bot$ is equal to the value argument of the latest preceding put method call with the same key argument. Formally,*

$$\begin{aligned}
&\forall m \in BasicMap \colon \forall X \in \mathbb{H}_B(m) \colon X \in Sequential \Rightarrow \\
&\forall l_g \colon isXGet_{X,m}(l_g) \ \wedge \ \neg(retv_X(l_g) = \bot) \Rightarrow \\
&\exists l_p \colon isPutter_{X,m}(l_p, l_g) \ \wedge \\
&\qquad arg2_X(l_p) = retv_X(l_g)
\end{aligned} \tag{5.49}$$

Lemma 44 (BasicMapPut) *In every sequential execution on a basic map, for every get method call g, if p is the latest preceding put method call with the same key argument then the return value of g is equal to the value argument of p. Formally,*

$$\begin{aligned}
&\forall m \in BasicMap \colon \forall X \in \mathbb{H}_B(m) \colon X \in Sequential \Rightarrow \\
&\forall l_g, l_p \colon \\
&\qquad isXGet_{X,m}(l_g) \ \wedge \\
&\qquad isPutter_{X,m}(l_p, l_g) \ \wedge \\
&\Rightarrow \\
&\qquad retv_X(l_g) = arg2_X(l_p)
\end{aligned} \tag{5.50}$$

Proof Sketches.
Lemma 19:

Straightforward from $\prec_X \subseteq \prec_L$.

Lemma 20:

We have

(1) $l \prec_L l'$

From [1], we have

(2) $rEv(l) \lhd_L iEv(l')$

From the well-formedness of the history O,
we have

(3) $iEv(l) \lhd_L rEv(l)$

(4) $iEv(l') \lhd_L rEv(l')$

From [3], [2] and [4], we have

(5) $iEv(l) \lhd_L rEv(l')$

From [5], we have

(6) $\neg(rEv(l') \lhd_L iEv(l))$

From [2] and [6], we have

(7) $\neg(l' = l)$

From the definition of $\prec_X$ on [6], we have

(8) $\neg(l' \prec_L l)$

The conclusion is

[8] and [7]

Lemma 21:

Straightforward from the fact that L is a member of sequential specification and
a sequential specification is a set of sequential histories and
the execution order is total in sequential histories.

Lemma 22:

Straightforward from the fact that L is a member of sequential specification and
a sequential specification is a set of sequential histories and
the execution order is total in sequential histories.

We have

(1) $l \in X$

(2) $l' \in X$

(3) $X \equiv L$

(4) $L \in SeqSpec(o)$

From [4], we have

(5) $L \in Sequential$

From [3], [1] and [2], we have

(6) $l \in L$

(7) $l' \in L$

From [4], [6] and [7], we have

$l \prec_L l' \ \vee \ l' \prec_L l \ \vee \ l = l'$

Lemma 23:

Straightforward from the fact that L is equivalent to X.

We have

(1) $X \equiv L$

(2) $L \in SeqSpec(o)$
(3) $l \prec_L l'$
From [3], we have
(4) $l \in L$
(5) $l' \in L$
From [2] on [4] and [5], we have
(6) $obj_L(l) = o$
(7) $obj_L(l') = o$
From [1] on [4] and [5], we have
$l \in X$
$l' \in X$
From [1] on [6] and [7], we have
$obj_X(l) = o$
$obj_X(l') = o$

Lemma 24:
Using L2X and XTotal, we have four cases:
Case: $l \prec l'$
Straightforward from XTrans.
Case: $l \sim l'$
Straightforward from XXTrans.
Case: $l' \prec l$
Straightforward from X2L and LASym.
Case: $l' = l$
Straightforward from LASym.

Lemma 25:
Derived from the semantics of basic objects (Definition 1) and the sequential specification of register (Definition 3).

Lemma 26:
Derived from the semantics of basic register (Definition 4).

Lemma 27:
This is a restatement of Theorem 3 from the original definition of linearizability []. Derivable from the semantics of linearizable objects (Definition 2) and the sequential specification of register (Definition 3).

Lemma 28:
Derivable from the semantics of linearizable objects (Definition 2) and the sequential specification of cas register (Definition 5).

Lemma 29:
Derivable from the semantics of linearizable objects (Definition 2) and the sequential specification of cas register (Definition 5).

Lemma 30:
Derivable from the semantics of linearizable objects (Definition 2), the sequential specification of the lock

(Definition 6), the owner-respecting property (Definition 7), and that the sub-history for each thread is sequential (from the definition of execution histories).

Lemma 31:
Derived from Lemma 30.

Lemma 32:
Derived from Lemma 30 and the sequential specification of lock (Definition 6).

Lemma 33:
Derived from Lemma 30 and the sequential specification of lock (Definition 6).

Lemma 34:
Derived from Lemma 30 and the sequential specification of lock (Definition 6).

Lemma 35:
Derivable from the semantics of linearizable objects (Definition 2), the sequential specification of the lock (Definition 8), the owner-respecting property (Definition 35), and that the sub-history for each thread is sequential (from the definition of execution histories).

Lemma 36:
Derived from Lemma 35.

Lemma 37:
Derived from Lemma 35 and the sequential specification of try-lock (Definition 8).

Lemma 38:
Derived from Lemma 35 and the sequential specification of try-lock (Definition 8).

Lemma 39:
Derived from Lemma 35 and the sequential specification of try-lock (Definition 8).

Lemma 40:
Derivable from the semantics of linearizable objects (Definition 2), the sequential specification of counter (Definition 10).

Lemma 41:
Derivable from the semantics of basic objects (Definition 1), the sequential specification of set (Definition 11).

Lemma 42:
Derivable from the semantics of basic objects (Definition 1), the sequential specification of set (Definition 11).

Lemma 43:
Derivable from the semantics of basic objects (Definition 1), the sequential specification of set (Definition 12).

Lemma 44:
Derivable from the semantics of basic objects (Definition 1), the sequential specification of set (Definition 12).

6 Marking TL2

```
$reg$: BasicRegister$[|I|]$,
$ver$: AtomicRegister$[|I|]$,
$lock$: TryLock$[|I|]$,
$clock$: SCounter,
$rver$: ThreadLocal BasicRegister,
$rset$: ThreadLocal BasicSet,
$wset$: ThreadLocal BasicMap,
$lset$: ThreadLocal BasicSet
```

```
def $init_t()$
$\boxed{I01 \triangleright}$    $snap = clock.read()$,
$I02 \triangleright$    $rver[t].write(snap)$,
$I03 \triangleright$    return $ok$,

def $read_t(i)$
$R01 \triangleright$    $pv = wset[t].get(i)$,
            if $(pv \neq \bot)$
$R02 \triangleright$        return $pv$,

$R03 \triangleright$    $s_1 = ver[i].read()$,
$\boxed{R04 \triangleright}$    $v = reg[i].read()$,
$R05 \triangleright$    $l = lock[i].read()$,
$R06 \triangleright$    $s_2 = ver[i].read()$,
$R07 \triangleright$    $sver = rver[t].read()$,
            if $(\neg(\neg l \ \wedge \ s_1 = s_2 \ \wedge \ s_2 \leq sver))$
$R08 \triangleright$        return $\mathbb{A}$,

$R09 \triangleright$    $rver[t].add(i)$,
$R10 \triangleright$    return $v$,
$\{R03 \rightarrow R04,\ R04 \rightarrow R05,\ R05 \rightarrow R06\}$,

def $write_t(i, v)$
$W01 \triangleright$    $wset[t].put(i, v)$,
$W02 \triangleright$    return $ok$,

def $abort_t()$
$A01 \triangleright$        return $\mathbb{A}$,
```

```
def $commit_t()$
$C01 \triangleright$    foreach $(i \in wset[t])$
$C02_i \triangleright$        $locked = lock[i].trylock()$,
            if $(\neg locked)$
$C03_i \triangleright$            $lset.add(i)$
            else
$C04_i \triangleright$            foreach $(j \in lset)$
$C05_{ij} \triangleright$                $lock[j].unlock()$,
$C06_i \triangleright$            return $\mathbb{A}$,

$\boxed{C07 \triangleright}$    $wver = clock.iaf()$,

$C08 \triangleright$    $sver = rver[t].read()$,
        if $(wver \neq sver + 1)$
$C09 \triangleright$        foreach $(i \in rset[t])$
$C10_i \triangleright$            $l = lock[i].read()$,
$C11_i \triangleright$            $s = ver[i].read()$,
                if $(\neg(\neg l \ \wedge \ s \leq sver))$
$C12_i \triangleright$                foreach $(j \in lset)$
$C13_{ij} \triangleright$                    $lock[j].unlock()$,
$C14_i \triangleright$                return $\mathbb{A}$,

$C15 \triangleright$    foreach $((i, v) \in wset[t])$
$\boxed{C16_i \triangleright}$        $reg[i].write(v)$,
$C17_i \triangleright$        $ver[i].write(wver)$,
$C18_i \triangleright$        $lock[i].unlock()$,

$C19 \triangleright$    return $\mathbb{C}$,
$\{C01 \rightarrow C07,\ C10 \rightarrow C11,\ C09 \rightarrow C15,$
 $C16 \rightarrow C17, C17 \rightarrow C18\}$,
```

Figure 4: TL2 Algorithm Specification

Atomic register, try-lock and strong counter are linearizable object types and basic register, basic set and basic map are basic object types. (At a high level, for every execution on a linearizable object, there is an equivalent sequential execution that complies with the sequential specification of the object. On the other hand, a basic object complies with its sequential specification only if it is accessed sequentially.) TL2 uses the following base objects: Value registers reg: an array of basic registers. Version registers ver: an array of atomic registers with the initial value 0. Locks $lock$: an array of try-locks that are initially released. The arrays are of size memory location count $|I|$.[2] Global version clock $clock$: a strong counter with the initial value 0. A strong counter provides two methods in its interface: iaf (inc-and-fetch) that increments the counter and returns the counter value and $read$ that returns the counter value. Read version $rver$: a thread-local basic register. Read set $rset$: a thread-local basic set that is initially $\emptyset$. Write set $wset$: a thread-local basic map that is initially $\emptyset$. Lock set $lset$: a thread-local basic set that is initially $\emptyset$. As relaxed execution may reorder program statements, any order that is not implied by the data or control dependencies but is required for the correctness of the algorithm is explicitly declared at the end of each method definition. The values ok, $\mathbb{A}$, $\mathbb{C}$ are reserved to denote successful completion of writes and abortion and commitment of transactions respectively.

TL2 is a deferred-update TM algorithm. A value that a transaction t writes to a location is buffered in the write set $wset[t]$ at $W01$ and is written back to register $reg[i]$ at $C16_i$ while t is committing. TL2 records a version in the register $ver[i]$ for the value stored in the register $reg[i]$. The version register $ver[i]$ is updated to ascending numbers at $C17_i$ after new values are written back to $reg[i]$ at $C16_i$. The try-lock $lock[i]$ is used for exclusive access to the registers for location i. At commit, the lock $lock[i]$ of each location i in the write set $wset[t]$ is acquired at $C01$ to $C06$. (If a lock cannot be acquired, the previously acquired locks are released at $C05$ and the transaction is aborted at $C06$.) Then, a new snapshot number is read from $clock$ at $C07$. Then, for each location in the read set $rset[t]$, first $lock[i]$ and then $ver[i]$ are read at $C10_i$ and $C11_i$ and the read is validated. (If a read is not validated, the acquired locks are released at $C13$ and the transaction is aborted at $C14$.) Finally, the value buffered for each location i in $wset[t]$ is written back at $C15_i$ to $C18_i$. For each pair in the write set $wset[t]$, the following three operations are executed in order. First, the buffered value is written back to $reg[i]$, then $ver[i]$ is updated, and then $lock[i]$ is released. In the $init$ method, each transaction t reads the current snapshot version from $clock$ at $I01$ and writes it to the read version register $rver[t]$ at $I02$. The read version is read at $R07$ and $C08$ to validate the read values. To read a location i, a transaction reads $ver[i]$, $reg[i]$, $lock[i]$ and again $ver[i]$ in order at $R03$ to $R06$ and then validates the read. (If the validation fails, the transaction is aborted.) Finally, i is added to the read set $rset[t]$ and the read value is returned.

---

[2] As observed by previous work [3], in the original TL2 paper, the authors maintain the version number and the lock bit of every location in the same memory word, thus, the order of reading the lock and the version register in the commit method is ambiguous. In our specification, we treat the lock and the version as separate registers and make the orders explicit.

Notation. Let us remind the notation. Consider an execution history H. We use $l_1 \prec_H l_2$ to denote that l_1 is executed before l_2. We use $l_1 \sim_H l_2$ to denote that l_1 is executed concurrently to l_2. We use $l_1 \precsim_H l_2$ to denote that l_1 is executed before or concurrently to l_2. We use $\prec_{clock}$, $\prec_{ver[i]}$ and $\prec_{lock[i]}$ to denote the linearization order of *clock*, *ver*[*i*] and *lock*[*i*] respectively.

A label c_1'c_2 is a call string that denotes a method call labeled c_2 that is executed in the body of the method call labeled c_1.

We use $\mathit{initOf}_H(T)$ and $\mathit{commitOf}_H(T)$ to denote the *init* and *commit* method calls of transaction T in history H.

Marking Relation. Now, we define the marking relation for TL2. The effect order of transactions is the linearization order of their calls to the *clock*. Every transaction reads an initial snapshot number at $I01$. A committing transaction makes a new snapshot at $C07$. A TL2 transaction takes effect at $C07$ if it is committed and at $I01$ otherwise. The access order of read operations and writer transactions to location i is the execution order of their accesses to the *reg*[*i*] register. The read method reads *reg*[*i*] at $R04$ and a writer transaction writes to *reg*[*i*] at $C16_i$.

Definition 13 (Marking TL2) *Consider an execution history $H \in \mathbb{H}(TL2)$. Let*

$$\begin{aligned}
\mathit{readAcc}(R) &= R\text{'}R04 \\
\mathit{writeAcc}(T,i) &= \mathit{commitOf}_H(T)\text{'}C16_i \\
\mathit{Eff}(T) &= \begin{cases} \mathit{initOf}_H(T)\text{'}I01 & \mathit{if}\ T \in \mathit{Aborted}(H) \\ \mathit{commitOf}_H(T)\text{'}C07 & \mathit{if}\ T \in \mathit{Committed}(H) \end{cases}
\end{aligned}$$

The marking $\sqsubseteq$ for H is the reflexive closure of $\sqsubset$ that is define as follows:

$$\begin{aligned}
&\{(T,T') \mid T,T' \in \mathit{Trans}(H) \ \wedge\ \mathit{Eff}(T) \prec_{clock} \mathit{Eff}(T')\} \cup \\
&\{(T,R) \mid \exists i\colon R \in \mathit{GlobalTReads}(H), i = \mathit{arg1}(R), T \in \mathit{Writers}_H(i) \ \wedge\ \mathit{writeAcc}(T,i) \prec_H \mathit{readAcc}(R)\} \cup \\
&\{(R,T) \mid \exists i\colon R \in \mathit{GlobalTReads}(H), i = \mathit{arg1}(R), T \in \mathit{Writers}_H(i) \ \wedge\ \mathit{readAcc}(R) \precsim_H \mathit{writeAcc}(T,i)\}
\end{aligned}$$

We have formally proved the markability of TL2 using a novel program logic that facilitates reasoning about execution and linearization orders. To keep the focus of this paper on markability, we avoid the formal presentation of the logic and present a simplified reasoning.

In addition to the lemmas presented in the previous section, we use the rule P2X that states the program-order-preservation property. If a method call l_1 is ordered before a method call l_2 in the program, and methods l_1 and l_2 are executed, then l_1 is executed before l_2.

Lemma 45 *TL2 preserves reads of aborted transactions (part 1).*

$$\begin{aligned}&\forall H \in \mathbb{H}(TL2)\colon\\ &\forall R \in \mathit{GlobalTReads}(H)\colon \mathit{Let}\ i = arg1_H(R), T = trans_H(R)\colon\\ &\quad T \in \mathit{Aborted}(H) \Rightarrow NoWriterBetween_{H,i}(R, \sqsubseteq, T)\end{aligned}$$

Proof Sketch.

T	T'
	$C02_i \triangleright$ $lock[i].trylock()$
	...
	$\boxed{C07 \triangleright}$ $wver = clock.iaf()$
$\boxed{I01 \triangleright}$ $snap = clock.read()$	...
...	
$R03 \triangleright$ $s_1 = ver[i].read()$	
$\boxed{R04 \triangleright}$ $v = reg[i].read()$	
	$\boxed{C16_i \triangleright}$ $reg[i].write(v)$
	$C17_i \triangleright$ $ver[i].write(wver)$
$R05 \triangleright$ $l = lock[i].read()$	$C18_i \triangleright$ $lock[i].unlock()$
$R06 \triangleright$ $s_2 = ver[i].read()$	
$R07 \triangleright$ $sver = rver[t].read()$	
if $(\neg(\neg l \ \wedge\ s_1 = s_2 \ \wedge\ s_2 \leq sver))$ **return** $\mathbb{A}$	

Figure 5: Case $T \in \mathit{Aborted}(H) \ \wedge\ R \sqsubset T' \sqsubset T$

We consider an aborted transaction T with an unaborted global read operation R from a location i and a writer T' of i.
We assume that
T' accesses i after R
that is
(1) $T' \sqsubset R$
and
T' takes effect before T
that is
(2) $T' \sqsubset T$
We show that
TL2 aborts R.
Figure 5 depicts the two transactions.
By Definition 13 on [1], we have
(3) $R04 \prec_H C16_i$
By Definition 13 on [2], we have
(4) $C07 \prec_{clock} I01$
The method calls $R05$ and $C18_i$ are on the object $lock[i]$. We consider two cases for the linearization order of them and prove that R returns $\mathbb{A}$ in both cases.
Case 1:
(5) $R05 \prec_{lock[i]} C18_i$

By P2X on the algorithm, we have

(6) $C02_i \prec_H C07$

(7) $I01 \prec_H R05$

By the Lemma XLTrans on [6], [4] and [7], we have

$C02_i \prec_H R05$

thus, by the Lemma X2L, we have

(8) $C02_i \prec_{lock[i]} R05$

By the Lemma TryLockReadM on [8] and [5], we have that

$R05$ returns *true* i.e. $l = true$

Thus,

The validation check fails and R returns $\mathbb{A}$.

Case 2:

(9) $C18_i \prec_{lock[i]} R05$

By P2X on the algorithm, we have

(10) $C17_i \prec_H C18_i$

(11) $R05 \prec_H R06$

By the Lemma XLTrans on [10], [9] and [11], we have

$C17_i \prec_H R06$

Thus, by the Lemma X2L, we have

(12) $C17_i \prec_{ver[i]} R06$

By Lemma 54 on [12], we have

(13) $wver \leq s_2$

By P2X on the algorithm, we have

(14) $R03 \prec_H R04$

(15) $C16_i \prec_H C17_i$

By the Lemma XXTrans on [14], [3] and [15], we have

$R03 \prec_H C17_i$

Thus, by the Lemma X2L, we have

(16) $R03 \prec_{ver[i]} C17_i$

By Lemma 54 on [16], we have

(17) $s_1 < wver$

From [13] and [17], we have

$\neg(s_1 = s_2)$

Thus,

The validation check fails and R returns $\mathbb{A}$ in this case too.

□

Lemma 46 *TL2 preserves reads of aborted transactions (part 2).*

$$\begin{array}{l} \forall H \in \mathbb{H}(TL2)\colon \\ \forall R \in GlobalTReads(H)\colon Let\ i = arg1_H(R), T = trans_H(R)\colon \\ \quad T \in Aborted(H) \Rightarrow NoWriterBetween_{H,i}(T, \sqsubseteq, R) \end{array}$$

Proof Sketch.

We consider an aborted transaction T with an unaborted global read operation R from a location i and a writer T' of i.

We assume that

T		T'	
$\boxed{I01 \triangleright}$	$snap = clock.read()$	$C02_i \triangleright$	$lock[i].trylock()$
$I02 \triangleright$	$rver[t].write(snap)$		
		$\boxed{C07 \triangleright}$	$wver = clock.iaf()$
		$\boxed{C16_i \triangleright}$	$reg[i].write(v)$
$\boxed{R04 \triangleright}$	$v = reg[i].read()$	$C17_i \triangleright$	$ver[i].write(wver)$
$R05 \triangleright$	$l = lock[i].read()$	$C18_i \triangleright$	$lock[i].unlock()$
$R06 \triangleright$	$s_2 = ver[i].read()$		
$R07 \triangleright$	$sver = rver[t].read()$		
	if $(\neg(\neg l \ \wedge\ s_1 = s_2 \ \wedge\ s_2 \leq sver))$ **return** $\mathbb{A}$		

Figure 6: Case $T \in Aborted(H) \ \wedge\ T \sqsubset T' \sqsubset R$

T' takes effect after T

that is

(1) $T \sqsubset T'$

and

T' accesses i before R

that is

(2) $T' \sqsubset R$

We show that

TL2 aborts R.

Figure 6 depicts the two transactions.

By Definition 13 on [1], we have

(3) $I01 \prec_{clock} C07$

By Definition 13 on [2], we have

(4) $C16_i \preceq R04$

The method calls $R05$ and $C18_i$ are on the object $lock[i]$. We consider two cases for the linearization order of them and prove that R returns $\mathbb{A}$ in both cases.

Case 1:

(5) $R05 \prec_{lock[i]} C18_i$

By P2X on the algorithm, we have

(6) $C02_i \prec_H C16_i$

(7) $R04 \prec_H R05$

By the Lemma XXTrans on [6], [4] and [7], we have

$C02_i \prec_H R05$

thus, by the Lemma X2L, we have

(8) $C02_i \prec_{lock[i]} R05$

By the Lemma TryLockReadM on [8] and [5], we have that

$R05$ returns $true$ i.e. $l = true$.

Thus,

The validation check fails and R returns $\mathbb{A}$.

Case 2:

(9) $C18_i \prec_{lock[i]} R05$

By P2X on the algorithm, we have

(10) $C17_i \prec_H C18_i$

(11) $R05 \prec_H R06$

By the Lemma XLTrans on [10], [9] and [11], we have

$C17_i \prec_H R06$

Thus, by the Lemma X2L, we have

(12) $C17_i \prec_{ver[i]} R06$

By Lemma 53 on [12], we have

(13) $wver \leq s_2$

By the Lemma SCounter on [3], we have

(14) $snap < wver$

The value of $sver$ is read at $R07$ from $rver$.

The thread-local register $rver$ is only assigned at $I02$ to $snap$.

Thus, we have

(15) $snap = sver$

From [13], [14] and [15], we have

$sver > s_2$

Thus,

The validation check fails and R returns $\mathbb{A}$ in this case too.

□

Lemma 47 *TL2 preserves reads of aborted transactions.*

$$\begin{aligned}
&\forall H \in \mathbb{H}(TL2):\\
&\forall R \in GlobalTReads(H): Let\ i = arg1_H(R), T = trans_H(R):\\
&\quad T \in Aborted(H) \Rightarrow\\
&\quad NoWriterBetween_{H,i}(R, \sqsubseteq, T) \ \wedge \ NoWriterBetween_{H,i}(T, \sqsubseteq, R)
\end{aligned}$$

Proof. Immediate from Lemma 45 and Lemma 46. □

Lemma 48 *TL2 preserves reads of committed transactions (part 1).*

$$\begin{aligned}
&\forall H \in \mathbb{H}(TL2):\\
&\forall R \in GlobalTReads(H): Let\ i = arg1_H(R), T = trans_H(R):\\
&\quad T \in Committed(H) \Rightarrow\\
&\quad NoWriterBetween_{H,i}(R, \sqsubseteq, T)
\end{aligned}$$

Proof Sketch.

We consider a committed transaction T with an unaborted global read operation R from a location i and a writer T' of i.

We assume that

T' accesses i after R

that is

(1) $R \sqsubset T'$

and

T' takes effect before T

that is

T		T'	
		$C02'_i \triangleright$	$lock[i].trylock()$
			...
$I01 \triangleright$	$snap = clock.read()$	$\boxed{C07' \triangleright}$	$wver' = clock.iaf()$
$I02 \triangleright$	$rver[t].write(snap)$		...
	...		
$\boxed{R04 \triangleright}$	$v = reg[i].read()$		
	...	$\boxed{C16'_i \triangleright}$	$reg[i].write(v')$
$\boxed{C07 \triangleright}$	$wver = clock.iaf()$		
	...		
$C08 \triangleright$	$sver = rver[t].read()$		
	if $(wver \neq sver + 1)$	$C17'_i \triangleright$	$ver[i].write(wver')$
$C10_i \triangleright$	$l = lock[i].read()$	$C18'_i \triangleright$	$lock[i].unlock()$
$C11_i \triangleright$	$s = ver[i].read()$		
	if $(\neg(\neg l \ \wedge \ s \leq sver))$ **foreach** $(j \in lset)$ $lock[j].unlock()$ **return** $\mathbb{A}$		

Figure 7: Case $T \in Committed(H) \ \wedge \ R \sqsubset T' \sqsubset T$

(2) $T' \sqsubset T$

We show that

TL2 aborts R.

Figure 7 depicts the two transactions. We annotate the labels and variables of T' by a prime so that they do not conflict with the labels and variables of T.

By Definition 13 on [1], we have

(3) $R04 \prec_H C16_i$

By Definition 13 on [2], we have

(4) $C07' \prec_{clock} C07$

The method calls $I01$ and $C07'$ are on the object $clock$. We consider two cases for the linearization order of them.

Case 1:

(5) $C07' \prec_{clock} I01$

From [5] and [3],

The proof of this case reduces to the proof of Lemma 45.

Case 2:

(6) $I01 \prec_{clock} C07'$

By the Lemma SCounter on [4], we have

(7) $wver' < wver$

By the Lemma SCounter on [6], we have

(8) $snap < wver'$

The value of $sver$ is read at $R07$ from $rver$.

The thread-local register $rver$ is only assigned at $I02$ to $snap$.

Thus, we have

(9) $snap = sver$

From [8] and [9], we have

(10) $sver < wver'$

From [10] and [7], we have

(11) $wver \neq sver + 1$

Thus,

The if branch is taken.

The method calls $C10_i$ and $C18'_i$ are on the object $lock[i]$.

We consider two cases for the linearization order of them.

Case 2.1:

(12) $C10_i \prec_{lock[i]} C18'_i$

By P2X on the algorithm, we have

(13) $C02'_i \prec_H C07'$

(14) $C07 \prec_H C10_i$

By the Lemma XLTrans on [13], [4] and [14], we have

$C02'_i \prec_H C10_i$

thus, by the Lemma X2L, we have

(15) $C02'_i \prec_{lock[i]} C10_i$

By the Lemma TryLockReadM on [15] and [12], we have that

$R05$ returns $true$ i.e. $l = true$

Thus,

The validation check fails and R returns $\mathbb{A}$.

Case 2.2:

(16) $C18'_i \prec_{lock[i]} C10_i$

By P2X on the algorithm, we have

(17) $C17'_i \prec_H C18'_i$

(18) $C10_i \prec_H C11_i$

By the Lemma XLTrans on [17], [16] and [18], we have

$C17'_i \prec_H C11_i$

Thus, by the Lemma X2L, we have

(19) $C17'_i \prec_{ver[i]} C11_i$

By Lemma 54 on [19], we have

(20) $wver' \leq s$

From [10], [20], we have

$sver < s$

Thus,

The validation check fails and R returns $\mathbb{A}$ in this case too.

□

Lemma 49 *TL2 preserves reads of committed transactions (part 2).*

$$\begin{aligned}
&\forall H \in \mathbb{H}(TL2):\\
&\forall R \in GlobalTReads(H): \textit{Let } i = arg1_H(R), T = trans_H(R):\\
&\quad T \in Committed(H) \Rightarrow\\
&\quad NoWriterBetween_{H,i}(T, \sqsubseteq, R)
\end{aligned}$$

T	T'
$\boxed{R04 \triangleright}$ $v = reg[i].read()$	
...	
$\boxed{C07 \triangleright}$ $wver = clock.iaf()$	
...	$\boxed{C07' \triangleright}$ $wver' = clock.iaf()$
	...
	$\boxed{C16'_i \triangleright}$ $reg[i].write(v')$

Figure 8: Case $T \in Committed(H) \ \wedge \ T \sqsubset T' \sqsubset R$

Proof Sketch.

We consider a committed transaction T with an unaborted global read operation R from a location i and a writer T' of i. We should show that it is impossible that T' takes effect after T and T' accesses i before R.

We assume that

T' takes effect after T

that is

(1) $T \sqsubset T'$

We show that

T' accesses i after R.

that is

(2) $R \sqsubset T'$

Figure 8 depicts the two transactions. We annotate the labels and variables of T' by a prime so that they do not conflict with the labels and variables of T.

By Definition 13 on [1], we have

(3) $C07 \prec_{clock} C07'$

By Definition 13 on [2], we have to show

$R04 \prec_H C16_i$

By P2X and the algorithm, we have

(4) $C04 \prec_H C07$

(5) $C07' \prec_H C16'_i$

By the Lemma XLTrans on [4], [3], and [5], we have

$R04 \prec_H C16_i$

□

Lemma 50 *TL2 preserves reads of committed transactions.*

$$\begin{array}{l} \forall H \in \mathbb{H}(TL2)\colon \\ \forall R \in GlobalTReads(H)\colon Let\ i = arg1_H(R), T = trans_H(R)\colon \\ \quad T \in Committed(H) \Rightarrow \\ \quad NoWriterBetween_{H,i}(R, \sqsubseteq, T) \ \wedge \ NoWriterBetween_{H,i}(T, \sqsubseteq, R) \end{array}$$

Proof. Immediate from Lemma 48 and Lemma 49. □

Lemma 51 *TL2 is read-preserving.*

$$\forall H \in \mathbb{H}(TL2)\colon ReadPres(H, \sqsubseteq)$$

Proof. Immediate from Lemma 47 and Lemma 50. □

Lemma 52 *Version registers are updated to ascending numbers.*

Let $C17_i^1$ *denote the method call at line* $C17_i$ *executed by a transaction* T_1 *and let* $wver^1$ *denote its argument. Similarly, let* $C17_i^2$ *denote the method call at line* $C17_i$ *executed by a transaction* T_2 *and let* $wver^2$ *denote its argument. If* $C17_i^1 \prec_{ver[i]} C17_i^2$*, then* $wver^1 < wver^2$*.*

Proof Sketch.

T_1		T_2	
	...		
$C02_i^1$ ▷	$\mathit{locked}^1 = \mathit{lock}[i].\mathit{trylock}()$		
	...		
$C07^1$ ▷	$\mathit{wver}^1 = \mathit{clock}.\mathit{iaf}()$		
	...		
$C17_i^1$ ▷	$\mathit{ver}[i].\mathit{write}(\mathit{wver}^1)$		
$C18_i^1$ ▷	$\mathit{lock}[i].\mathit{unlock}()$		...
	...	$C02_i^2$ ▷	$\mathit{locked}^2 = \mathit{lock}[i].\mathit{trylock}()$
			...
		$C07^2$ ▷	$\mathit{wver}^2 = \mathit{clock}.\mathit{iaf}()$
			...
		$C17_i^2$	$\mathit{ver}[i].\mathit{write}(\mathit{wver}^2)$
		$C18_i^2$	$\mathit{lock}[i].\mathit{unlock}()$
			...

Figure 9: Updating Version Registers

We have that
(1) $C17_i^1 \prec_{ver[i]} C17_i^2$
We show that
$wver^1 < wver^2$

By P2X on the algorithm, we have
(2) $C02_i^1 \prec_H C17_i^1$
(3) $C17_i^2 \prec_H C18_i^2$
By the Lemma XLTrans on [2], [1] and [3], we have
(4) $C02_i^1 \prec_H C18_i^2$
Thus, by the Lemma X2L, we have
(5) $C02_i^1 \prec_{lock[i]} C18_i^2$
From the algorithm,
(6) The ownership of *lock*[*i*] is respected.
By the Lemma TryLock on [6] and [5], we have
(7) $C18_i^1 \prec_{lock[i]} C02_i^2$
By P2X on the algorithm, we have
(8) $C07_1 \prec_H C18_i^1$
(9) $C02_i^2 \prec_H C07^2$
By the Lemma XLTrans on [8], [7], and [9], we have
(10) $C07^1 \prec_H C07^2$
By the Lemma X2L on [10], we have

(11) $C07^1 \prec_{clock} C07^2$

By the Lemma SCounter on [11], we have

$wver^1 < wver^2$

$\square$

Lemma 53 *For every write method call* W *on* $ver[i]$ *with argument* v *and every read method call* R *on* $ver[i]$ *with the return value* v'*, if* $W \prec_{ver[i]} R$ *then* $v \leq v'$*.*

Proof Sketch.

We have

(1) W is a write method call on $ver[i]$.
(2) R is a read method call on $ver[i]$.
(3) $W \prec_{ver[i]} R$.
(4) The argument of W is v.
(5) The return value of R is v'.

We show that

$v \leq v'$

Let

(6) W' is last write on $ver[i]$ linearized before R.
(7) The argument of W' is v''.

By the Lemma AReg' on [6], [7], and [5], we have

(8) $v' = v''$

From [6], and [1], we have

(9) $W \preceq_{ver[i]} W'$

By the algorithm and [1], and [6], we have

(10) W and W' are both at $C17$.

By Lemma 52 on [10], [9], [4] and [7], we have

(11) $v \leq v''$

From [8] and [11], we have

$v \leq v'$

$\square$

Lemma 54 *For every write method call* W *on* $ver[i]$ *with argument* v *and every read method call* R *on* $ver[i]$ *with the return value* v'*, if* $R \prec_{ver[i]} W$ *then* $v' < v$*.*

Proof Sketch.

We have

(1) W is a write method call on $ver[i]$.
(2) R is a read method call on $ver[i]$.
(3) $R \prec_{ver[i]} W$.
(4) The argument of W is v.
(5) The return value of R is v'.

We show that

$v' < v$

Let

(6) W' is last write on $ver[i]$ linearized before R.

(7) The argument of W' is v''.

By the Lemma AREG' on [6], [7], and [5], we have

(8) $v' = v''$

From [3], and [6], we have

(9) $W' \prec_{ver[i]} W$

By the algorithm and [1], and [6], we have

(10) W and W' are both at $C17$.

By Lemma 52 on [10], [9], [4] and [7], we have

(11) $v'' < v$

From [8] and [11], we have

$v' < v$

$\square$

Lemma 55 *TL2 is global-write-observant.*

$$\begin{array}{l}
\forall H \in \mathbb{H}(TL2) \colon \\
\forall R \in \mathit{GlobalTReads}(H) \colon \exists W \in \mathit{GlobalTWrites}(H) \colon \mathit{Let}\ T' = \mathit{trans}_H(W) \colon \\
\mathit{LastPreAccessor}_{H,\sqsubseteq}(T', R) \wedge \\
\mathit{arg1}_H(R) = \mathit{arg1}_H(W) \ \wedge \ \mathit{retv}_H(R) = \mathit{arg2}_H(W)
\end{array}$$

Proof Sketch.

We consider a transaction T with an unaborted global read operation R from a location i.
The read operation R is from the location i, thus,
(1) The argument of R is i.
As R is global, thus,
(2) The return value of R is the return value of $R04$.
We first show that
(3) The read method call from $reg[i]$ at $R04$ is race-free.
We assume that there is a write method call on $reg[i]$ concurrent to it and show that TL2 aborts R.
Figure 10 depicts this situation.

T		T'	
		$C02_i$ ▷	$locked = lock[i].trylock()$
			...
$R03$ ▷	$s_1 = ver[i].read()$		
$R04$ ▷	$v = reg[i].read()$	$C16_i$ ▷	$v = reg[i].write(v)$
	...	$C17_i$ ▷	$ver[i].write(wver)$
$R05$ ▷	$lock[i].read()$	$C18_i$ ▷	$lock[i].unlock()$
$R06$ ▷	$s_2 = ver[i].read()$		...
$R07$ ▷	$sver = rver[t].read()$		
	if $(\neg(\neg l \ \wedge \ s_1 = s_2 \ \wedge \ s_2 \leq sver))$ **return** $\mathbb{A}$		

Figure 10: $R04$ is race-free

We assume that there a race between $R04$ and $C16_i$. Thus,
(4) $R04 \sim C16_i$
The method calls $R05$ and $C18_i$ are on the object $lock[i]$.
We consider two cases for the linearization order of them and prove that R returns $\mathbb{A}$ in both cases.
We consider two cases
Case 1:
(5) $R04 \prec_{lock[i]} C18_i$
By P2X and the algorithm, we have
(6) $C02_i \prec_H C16_i$
(7) $R04 \prec_H R05$
By the Lemma XXTrans on [6], [4], and [7], we have
(8) $C02_i \prec_H R05$
By the Lemma X2L on [8], we have
(9) $C02_i \prec_{lock[i]} R05$
By the Lemma TryLockReadM on [9] and [5], we have that
$R05$ returns *true* i.e. $l = true$

Thus,

The validation check fails and R returns $\mathbb{A}$.

Case 2:

(10) $C18_i \prec_{lock[i]} R04$

By P2X and the algorithm, we have

(11) $R03 \prec_H R04$

(12) $R05 \prec_H R06$

(13) $C16_i \prec_H C17_i$

(14) $C17_i \prec_H C18_i$

By the Lemma XXTrans on [11], [4], and [13], we have

(15) $R03 \prec_H C17_i$

By Lemma 54 on [15], we have

(16) $s_1 < wver$

By the Lemma XLTrans on [14], [10], and [12], we have

(17) $C17_i \prec_H R06$

By Lemma 53 on [17], we have

(18) $s_2 > wver$

From [15] and [17], we have

(19) $s_1 \neq s_2$

Thus,

The validation check fails and R returns $\mathbb{A}$.

Second, we show that

(20) The register $reg[i]$ is sequentially-written i.e. no two write methods on $reg[i]$ are concurrent.

We assume two concurrent write method calls on $reg[i]$ and show a contradiction.

Figure 11 depicts this situation.

T		T'	
$C02_i \triangleright$	$locked = lock[i].trylock()$		
	...	$C02'_i \triangleright$	$locked' = lock[i].trylock()$
			...
$C16_i \triangleright$	$v = reg[i].write(v)$	$C16'_i \triangleright$	$v' = reg[i].write(v')$
	...		...
$C18_i \triangleright$	$lock[i].unlock()$		
		$C18'_i \triangleright$	$lock[i].unlock()$

Figure 11: $reg[i]$ is sequentially-written

We assume that $C16_i$ and $C16'_i$ are concurrent. Thus,

(21) $C16_i \sim C16'_i$

By P2X and the algorithm, we have

(22) $C02_i \prec_H C16_i$

(23) $C16'_i \prec_H C18'_i$

By the Lemma XXTrans on [22], [21], and [23], we have

(24) $C02_i \prec_H C18'_i$

By the Lemma X2L on [8], we have

(25) $C02_i \prec_{lock[i]} C18'_i$

By the Lemma TryLock on [25], we have that

(26) $C18_i \prec_{lock[i]} C02'_i$

By P2X and the algorithm, we have

(27) $C16_i \prec_H C18_i$

(28) $C02'_i \prec_H C16'_i$

By the Lemma XLTrans on [27], [26], and [28], we have

(29) $C16_i \prec_H C16'_i$

That is a contradiction to [21].

By the Lemma BReg on [3], and [20], we have

(30) There is a write method call w on $reg[i]$ such that
The argument of w is equal to the return value of $R04$.
The last write method call on $reg[i]$ that is executed before $R04$ is w.

By the algorithm, we have

(31) The register $reg[i]$ is written only at $C16_i$.

From [28] and [29], we have

There is a transaction T' such that
(We annotate the labels and variables of T' by a prime
so that they do not conflict with the labels and variables of T.)

(32) The argument of $C16'_i$ is equal to the return value of $R04$.

(33) The last write method call on $reg[i]$ that is executed before $R04$ is $C16'_i$.

By the algorithm, we have

(34) The argument of $C16'_i$ is the value of the key i in the map $wset[T']$ in the commit.

(35) The map $wset[T']$ is updated only at $W01$ in a *write* of T' such that
The key is equal to the first argument of the *write*.
The value is equal to the second argument of the *write*.

From [34], and [35], we have

(36) There exists a write W of T'

(37) The first argument of W is equal to i.

(38) W is the last *write* of T' with the first argument equal to i.

(39) The second argument of W is equal to the argument of $C16'_i$.

From [1], and [37], we have

(40) The first argument of R is the first argument of W.

From [2], [32], and [39], we have

(41) The return value of R is the second argument of W.

From [38], we have

(42) W is a global write.

We show that

(43) The transaction T' is the last pre-accessor of R.

From [33], we have

(44) $C16'_i \prec_H R04$

By Definition 13 on [44], we have

(45) $T' \sqsubset R$

Now, we show that

(46) Every transaction T'' other than T' that accesses i before R, takes effect before T'.

We assume that

(47) $T'' \neq T'$

(48) $T'' \sqsubset R$

We should show that

$T'' \sqsubset T'$

By Definition 13 on [48], we have

(We annotate the labels and variables of T' by a double prime.)

(49) $C16''_i \prec_H R04$

From [33], [33], and [49], we have

(50) $C16''_i \prec_H C16'_i$

Consider Figure 12.

T''		T'	
$C02''_i$ ▷	$locked'' = lock[i].tryLock()$		
	...	$C02'_i$ ▷	$locked' = lock[i].tryLock()$
$C07''$ ▷	$wver'' = clock.iaf()$		...
	...	$C07'$ ▷	$wver' = clock.iaf()$
$C16''_i$ ▷	$reg[i].write(v'')$		...
	...	$C16'_i$ ▷	$reg[i].write(v')$
$C18''_i$ ▷	$lock[i].unlock()$		...
		$C18'_i$ ▷	$lock[i].unlock()$

Figure 12: Effect-order of pre-accessors

By P2X and the algorithm, we have

(51) $C02''_i \prec_H C16''_i$

(52) $C16'_i \prec_H C18'_i$

By the Lemma XXTrans on [51], [50], and [52], we have

(53) $C02''_i \prec_H C18'_i$

By the Lemma X2L on [53], we have

(54) $C02''_i \prec_{lock[i]} C18'_i$

By the Lemma TryLock on [45], we have that

(55) $C18''_i \prec_{lock[i]} C02'_i$

By P2X and the algorithm, we have

(56) $C07''_i \prec_H C18''_i$

(57) $C02'_i \prec_H C07'_i$

By the Lemma XLTrans on [56], [55], and [57], we have

(58) $C07'' \prec_H C07'$

By Definition 13 on [58], we have

$T'' \sqsubset T'$.

The conclusion is

[36], [42], [40], [41], and [43]

□

Lemma 56 *TL2 is local-write-observant.*

$$\begin{array}{l}
\forall H \in \mathbb{H}(TL2)\colon \\
\forall R \in LocalTReads(H)\colon \mathit{Let}\ T = trans_H(R), i = arg1_H(R), H' = H|T|i\colon \\
\exists W \in TWrites(H')\colon \\
\qquad W \prec_{H'} R \,\wedge\, NoWriteBetween_{H'}(W, R) \,\wedge \\
\qquad retv_{H'}(R) = arg2_{H'}(W)
\end{array}$$

Proof Sketch.
Let

(1) The operation R is a local read with the first argument i by the transaction T.

From [1], as R is local, we have

(2) There is a write operation before R with the first argument i by T.

From [2], let

(3) The operation W is the last write operation before R with the first argument i by the transaction T.

By the algorithm

(4) The value of a key i in $wset$ is updated only at $W01$ in a write operation with the first argument i and the value of the key i is updated to the second argument of the write operation.

From [3] and [4], we have

(5) The value of a key i in $wset$ during the execution of R is equal to the second argument of W.

Thus, by the algorithm

(6) $R01$-$R02$ find a value for the key i in $wset$.

Thus,

(7) The return value of R is equal to the value of key i in $wset$.

From [7] and [5], we have

(8) The return value of R is equal to the second argument of W.

The conclusion is

[3] and [8]

□

Lemma 57 *TL2 is write-observant.*

$$\forall H \in \mathbb{H}(TL2)\colon\ WriteObs(H, \sqsubseteq)$$

Proof. Immediate from Lemma 56 and Lemma 55. □

Lemma 58 *TL2 is real-time-preserving.*

$$\forall H \in \mathbb{H}(TL2)\colon RealTimePres(H, \sqsubseteq)$$

Proof Sketch.
We assume that
  (1) $T \preccurlyeq_H T'$
We show that
  $T \sqsubseteq T'$
By the definition of $\preccurlyeq_H$, from [1], we have
  (2) All the operations of T are executed before all the operations of T'.
By the Lemma X2L, from [2], we have
  (3) All the operations of T on *clock* are linearized before all the operations of T' on *clock*.
By Definition 13,
  (4) The effect point of each transaction is one of its own operations on the *clock* object.
From [3] and [4], we have
  (5) The transaction T takes effect before the transaction T'.
that is
  $T \sqsubseteq T'$

□

Lemma 59 *The relation* $\sqsubseteq$ *is a marking relation.*

$$\forall H \in \mathbb{H}(TL2)\colon \sqsubseteq \in Marking(H)$$

Proof Sketch.

Consider Definition 13.

By the totality of the linearization order $\prec_{clock}$, the relation $\sqsubseteq$ is a total on the set of transactions.

As every pair of method calls either execute in order or concurrently, every read operation of a location i is ordered either before or after every writer to i. In addition, as no method call can execute before another method call and also after after or concurrent to it, no read operation of a location i is ordered both before and after a writer to i.

□

Lemma 60 *TL2 is markable.*

$$\forall H \in \mathbb{H}(TL2)\colon H \in FinalStateMarkable$$

Proof.
Immediate from Lemma 59, Lemma 51, Lemma 57, and Lemma 58.

□

Theorem 61 *TL2 is opaque.*

$$\forall H \in \mathbb{H}(TL2)\colon H \in FinalStateOpaque$$

Proof.
Immediate from Lemma 60, and Theorem 18.

□

7 Marking DSTM (visible reads)

```
$\mathcal{T}$:
    $Loc$ {
        $writer$: BasicRegister,
        $rset$: BasicSet,
        $oldVal$: BasicRegister,
        $newVal$: BasicRegister},
    $state$: AtomicCASRegister[ ],
    $start$: AtomicCASRegister[ ]
```

```
$\mathcal{D}$:
```

```
def $init_t()$
$I01 \triangleright$  $state[t].write(\mathbb{R})$,
$I02 \triangleright$  return $ok$,
```

```
def $read_t(i)$
$R01 \triangleright$  $r_1 = start[i].read()$,
$R02 \triangleright$  $v = currentValue_t(r_1)$,
$R03 \triangleright$  $r_2 = clone(r_1)$,
$R04 \triangleright$  $r_2.rset.add(t)$,
$R05 \triangleright$  $rd = start[i].cas(r_1, r_2)$,
$R06 \triangleright$  $s = state[t].read()$,
           if $(\neg rd \ \vee \ (s = \mathbb{A}))$
$R07 \triangleright$      return $\mathbb{A}$
           else
$R08 \triangleright$      return $v$,
$\{R05 \to R06\}$,
```

```
def $commit_t()$
$C01 \triangleright$  $c = state[t].cas(\mathbb{R}, \mathbb{C})$,
           if $(c)$
$C02 \triangleright$      return $\mathbb{C}$
           else
$C03 \triangleright$      return $\mathbb{A}$,
```

```
def $currentValue_t(r)$
$V01 \triangleright$  $t_2 = r.writer.read()$,
           if $(\neg(t_2 = t))$
$V02 \triangleright$      $state[t_2].cas(\mathbb{R}, \mathbb{A})$,
$V03 \triangleright$  $s = state[t_2].read()$,
           if $(s = \mathbb{A})$
$V04 \triangleright$      return $r.oldVal$
           else
$V05 \triangleright$      return $r.newVal$,
```

```
def $write_t(i, v)$
$W01 \triangleright$  $r_1 = start[i].read()$,
$W02 \triangleright$  $w = r_1.writer.read()$,
           if $(w = t)$
$W03 \triangleright$      $r_1.newVal.write(v)$,
$W04 \triangleright$      return $ok$,

$W05 \triangleright$  $v_2 = currentValue_t(r_1)$,
$W06 \triangleright$  foreach $(t_2 \in r_1.rset)$
$W07 \triangleright$      $state[t_2].cas(\mathbb{R}, \mathbb{A})$,

$W08 \triangleright$  $r_2 = new\ Loc()$,
$W09 \triangleright$  $r_2.writer.write(t)$,
$W10 \triangleright$  $r_2.oldVal.write(v_2)$,
$W11 \triangleright$  $r_2.newVal.write(v)$,
$W12 \triangleright$  $wd = start[i].cas(r_1, r_2)$,
           if $(wd)$
$W13 \triangleright$      return $ok$
           else
$W14 \triangleright$      return $\mathbb{A}$
$\{W06 \to W12\}$
```

Figure 13: *DSTMVis* DSTM (visible reads) Algorithm Specification

Notation. Let us remind the notation. Consider an execution history H.

We write $e_1 \lhd_H e_2$ to denote that the event e_1 comes before the event e_2 in the history H.

We use $l_1 \prec_H l_2$ to denote that l_1 is executed before l_2. We use $l_1 \sim_H l_2$ to denote that l_1 is executed concurrently to l_2. We use $l_1 \precsim_H l_2$ to denote that l_1 is executed before or concurrently to l_2.

We use $\prec_{start[i]}$ to denote the linearization order of $start[i]$.

A label c_1'c_2 is a call string that denotes a method call labeled c_2 that is executed in the body of the method call labeled c_1.

We use $initOf_H(T)$ and $commitOf_H(T)$ to denote the *init* and *commit* method calls of the transaction T in the history H. We use $LastTRead_H(T)$ to denote the last read method call by the transaction T in the history H. We use $FirstTWrite_H(T, i)$ to denote the first write method call to location i by the transaction T in the history H.

Marking Relation. Now, we define the marking relation for DSTM.

Definition 14 (Marking DSTM) *Consider an execution history $H \in \mathbb{H}(DSTMVis)$. Let*

$$\begin{aligned}
Eff(T) &= \begin{cases} commitOf_H(T)\text{'}C01 & \text{if } T \in Committed(H) \\ LastTRead_H(T)\text{'}R05 & \text{if } T \in Aborted(H) \;\wedge\; TReads(H) \neq \emptyset \\ initOf_H(T)\text{'}I01 & \text{if } T \in Aborted(H) \;\wedge\; TReads(H) = \emptyset \end{cases} \\
readAcc(R) &= R\text{'}R05 \\
writeAcc(T, i) &= FirstTWrite_H(T, i)\text{'}W12
\end{aligned}$$

The marking $\sqsubseteq$ for H is the reflexive closure of $\sqsubset$ that is define as follows:

$$\begin{aligned}
&\{(T, T') \mid T, T' \in Trans(H) \;\wedge\; inv(Eff(T)) \lhd_H inv(Eff(T'))\} \;\cup \\
&\{(T, R) \mid \exists i\colon R \in GlobalTReads(H), i = arg1(R), T \in Writers_H(i) \;\wedge\; writeAcc(T, i) \prec_{start[i]} readAcc(R)\} \;\cup \\
&\{(R, T) \mid \exists i\colon R \in GlobalTReads(H), i = arg1(R), T \in Writers_H(i) \;\wedge\; readAcc(R) \prec_{start[i]} writeAcc(T, i)\}
\end{aligned}$$

A committed transactions takes effect at the invocation event of $C01$, the *cas* method call in its commit method call. An aborted transaction that has a successful read method call takes effect at the invocation event of $R05$ of its last successful read method call. An aborted transaction that has no successful read method call takes effect at the invocation event of $I01$ in its initialization method call.

The access point of a read method call is at $R05$. The access point of a writer transaction to location i is at $W12$ of its first write method call to i.

8 Marking NORec

```
$\mathcal{T}$:
    $seqLock$: SeqLock,
    $reg$: BasicRegister[ ]
    $snap$: ThreadLocal BasicRegister,
    $rset$: ThreadLocal BasicMap,
    $wset$: ThreadLocal BasicMap,
```

```
$\mathcal{D}$:
```

```
def $init_t()$
            do
$I01$ ▷       $(s, l) = seqLock.read()$
            while $(l)$,
$I02$ ▷     $snap[t] = s$,
```

```
def $read_t(i)$
$R01$ ▷     $pv = wset[t].get(i)$,
            if $(pv \neq \bot)$
$R02$ ▷       return $pv$,
            do
$R03$ ▷       $v = reg[i].read()$,
$R04$ ▷       $s1 = snap[t].read()$,
$R05$ ▷       $(s2, l2) = seqLock.read()$,
              if $(s2 = s1 \ \wedge \ \neg l2)$
$R06$ ▷         break,
$R07$ ▷       $b = validate_t()$,
              if $(\neg b)$
$R08$ ▷         return $\mathbb{A}$,
            while $(true)$,
$R09$ ▷     $rset[t].put(i, v)$,
$R10$ ▷     return $v$,
$\{R03 \rightarrow R05\}$,
```

```
def $write_t(i, v)$
$W01$ ▷     $wset[t].put(i, v)$,
$W02$ ▷     return $ok$,
```

```
def $abort_t()$
$A01$ ▷     return $\mathbb{A}$
```

```
def $validate_t()$
$V01$ ▷     while $(true)$
              do
$V02$ ▷         $(s1, l1) = seqLock.read()$,
              while $(l1)$
              foreach $((i, v) \in rset[t])$
$V03_i$ ▷       $v' = reg[i].read()$,
                if $(v \neq v')$,
$V04_i$ ▷         return $false$,
$V05$ ▷       $(s2, l2) = seqLock.read()$,
              if $(s2 = s1 \ \wedge \ \neg l2)$
$V06$ ▷         $snap[t].write(s1)$,
$V07$ ▷         return $true$,
$\{V02 \rightarrow V03_i, V03_i \rightarrow V05\}$,
```

```
def $commit_t()$
$C01$ ▷     $e = wset[t].isEmpty()$,
            if $(e)$
$C02$ ▷       return $\mathbb{C}$,
            do
$C03$ ▷       $s = snap[t].read()$,
$C04$ ▷       $d = seqLock.compareAndLock(s)$,
              if $(d)$
$C05$ ▷         break,
$C06$ ▷       $b = validate_t()$,
              if $(\neg b)$
                return $\mathbb{A}$,
            while $(true)$,
            foreach $((i, v) \in wset[t])$
$C07_i$ ▷     $reg[i].write(v)$,
$C08$ ▷     $seqLock.incAndUnlock()$,
$C09$ ▷     return $\mathbb{C}$
$\{C04 \rightarrow C07_i, C07_i \rightarrow C08\}$,
```

Figure 14: *NORec* NORec Algorithm Specification

Notation. Let us remind the notation. Consider an execution history H.

We use $l_1 \prec_H l_2$ to denote that l_1 is executed before l_2. We use $l_1 \sim_H l_2$ to denote that l_1 is executed concurrently to l_2. We use $l_1 \precsim_H l_2$ to denote that l_1 is executed before or concurrently to l_2.

We use $\prec_{seqLock}$ to denote the linearization order of *seqLock*.

A label c_1'c_2 is a call string that denotes a method call labeled c_2 that is executed in the body of the method call labeled c_1.

We use $\mathit{initOf}_H(T)$ and $\mathit{commitOf}_H(T)$ to denote the *init* and *commit* method calls of the transaction T in the history H.

Marking Relation. Now, we define the marking relation for NoRec.

Definition 15 (Marking NoRec) *Consider an execution history $H \in \mathbb{H}(NORec)$. Let*

$$\begin{array}{rcl}
\mathit{REff}(T) & = & \textit{The last execution of } I01 \textit{ or } V05 \\
\mathit{Eff}(T) & = & \begin{cases} \mathit{REff}(T) & \textit{if } T \in \mathit{Aborted}(H) \ \vee \ TWrites(H) = \emptyset \\ \mathit{commitOf}(T)\text{'}C04 & \textit{if } T \in \mathit{Committed}(H) \ \wedge \ TWrites(H) \neq \emptyset \end{cases} \\
\mathit{readAcc}(T,i) & = & \begin{cases} R\text{'}R03 & \textit{if } \mathit{REff}(T) \prec_H R\text{'}R03 \\ \textit{Let } \mathit{REff}(T) = V\text{'}V05 \textit{ in } V\text{'}V03_i & \textit{if } R\text{'}R03 \prec_H \mathit{REff}(T) \end{cases} \\
\mathit{writeAcc}(T,i) & = & \mathit{commitOf}(T)\text{'}C07_i
\end{array}$$

The marking $\sqsubseteq$ for H is the reflexive closure of $\sqsubset$ that is define as follows:

$$\begin{array}{l}
\{(T,T') \mid T,T' \in Trans(H) \ \wedge \ \mathit{Eff}(T) \prec_{seqLock} \mathit{Eff}(T')\} \ \cup \\
\{(T,R) \mid \exists i\colon R \in GlobalTReads(H), i = arg1(R), T \in Writers_H(i) \ \wedge \ \mathit{writeAcc}(T,i) \prec_H \mathit{readAcc}(T,i)\} \ \cup \\
\{(R,T) \mid \exists i\colon R \in GlobalTReads(H), i = arg1(R), T \in Writers_H(i) \ \wedge \ \mathit{readAcc}(T,i) \prec_H \mathit{writeAcc}(T,i)\}
\end{array}$$

An aborted transaction or a read-only transaction takes effect at the last execution of $I01$ or $V05$. This method call reads that most recent snapshot value that the transaction is still consistent for. A committed transactions that has write method calls takes effect at $C04$.

The access point of a read method call is at $R03$ if the last recent snapshot is read before $R03$; otherwise, it is at $V03_i$ of the latest successful validate method call. The access point of a writer transaction to location i is at $C07_i$.

9 The Cost of Read Validation

The read-preservation invariant requires the TM algorithm to check that a read location is not overwritten between the point where the location is read and the point where the transaction takes effect. This requirement motivated us to study how read-preservation can influence the time complexity of TM operations and helped us construct client scenarios that exhibit lower bounds. We present a generalization of the seminal lower bound result presented in [2]. Let us first remind some definitions from previous works on the inherent complexity of TM [1, 2, 4, 5].

An aborted transaction that did not invoke an abort operation is said to be *forcefully* aborted. We say that two transactions *conflict* if they access the same location and one of them writes to the location. A TM algorithm is (weakly) *progressive* if and only if it forcefully aborts a transaction only when it conflicts with a live transaction. More precisely, it aborts a transaction only when there is a time t at which it conflicts with another concurrent transaction that is live at time t (not committed or aborted by time t). In addition to providing progress, progressive TM algorithms are expected to retry transactions less frequently and therefore, improve performance.

A TM algorithm is *invisible-reads* if and only if no read operation mutates any base object. Mutating base objects can potentially invalidate the caches and adversely affect performance. Thus, most high-performance TM algorithms are invisible-reads. A transaction is *read-only* if and only if it does invoke any write operations. We assume that the abort operation for a read-only transaction does not mutate any base shared object.

Two transactions *contend* on a base object o if and only if they access o and at least one of them mutates o. A TM algorithm is (strictly) *disjoint-access-parallel* if and only if two transactions contend on a base object only if they access a common memory location. Disjoint-access-parallelism can improve scalability as transactions that access disjoint memory locations access disjoint base objects.

A TM algorithm is *single-version* if and only if it stores a single value for each memory location in the base objects.

Theorem 62 *The time complexity of the commit operation of every opaque, progressive, disjoint-access-parallel and invisible-reads TM algorithm is* $\Omega(|\mathcal{R}|)$ *where* $\mathcal{R}$ *is the read set.*

We explain the key idea here and then present the proof. Consider a TM algorithm TM that is opaque, progressive, disjoint-access-parallel and invisible-reads. Consider the following client scenario. Invoke the following methods in sequence. Wait for the response of the method call of each step before going to the next step. (1) $init_{T_1}()$, (2) $read_{T_1}(i)$ (3) $init_{T_2}()$, (4) $write_{T_2}(i, v_1)$, (5) $commit_{T_2}()$, (6) $init_{T_3}()$, (7) $read_{T_3}(j)$, (8) $abort_{T_3}()$, (9) $write_{T_1}(j, v_1)$, (10) $commit_{T_1}()$. As the TM is opaque, progressive and invisible-reads, it can be shown that it results in the history H_1 depicted in Figure 15(a). The initializing transaction T_0 (that initializes every location to v_0) and also the initializing operations of transactions are elided for brevity.

To make sure that the read location i is not overwritten, the commit operation of T_1 should access a shared object that T_2 (that is a writer of i) mutates. Assume otherwise i.e. the commit operation of T_1 does not access any shared object that T_2 mutates. Thus, T_2 is invisible to T_1. As TM is invisible-reads, it can be shown that T_3 is invisible to other transactions. As T_2 and T_3 are invisible to T_1, removing them from the client scenario does not affect the responses that T_1 receives. Therefore, the execution of T_1 alone results in the execution history H_2 depicted in Figure 15(b). As there is no conflicting transaction and TM is progressive, TM cannot forcefully abort the commit operation of T_1. The commit operation should have returned $\mathbb{C}$ but has returned $\mathbb{A}$ that is a contradiction. Therefore, we conclude that the commit operation of T_1 accesses a shared object that T_2 mutates. The scenario can be trivially extended to an arbitrary location k in the read set $\mathcal{R}$ by generalizing the transaction T_2 with the transaction $T_2^k = write_{T_2^k}(k, v_1) \cdot commit_{T_2^k}()$. It can be shown that for every $k \in \mathcal{R}$, the commit operation of T_1 accesses a shared object that the transaction

T_2^k mutates. The transactions $\{T_2^k \mid k \in \mathcal{R}\}$ access disjoint locations. As TM is strictly disjoint-access-parallel, these transactions access disjoint shared objects. Thus, the commit operation of T_1 accesses a separate shared object for every $k \in \mathcal{R}$. Therefore, the commit operation of T_1 accesses at least $|\mathcal{R}|$ shared objects. Therefore, the time complexity of the commit operation of T_1 is $\Omega(|\mathcal{R}|)$.

This theorem shows that designers should pick at least one of the following sources of inefficiency in the design of every opaque TM algorithm: aborting non-conflicting transactions, sharing base objects between transactions that access disjoint locations, visible reads or linear-time complexity of the commit method. As an example, TL2 shares the *clock* object between all transactions and is, therefore, not disjoint-access-parallel. In addition, it has linear-time read-validation in the commit method.

Proof.
Consider a TM algorithm TM that is opaque, progressive, disjoint-access-parallel and invisible-reads. We describe the following client scenario with three transactions T_1, T_2 and T_3 and consider its execution with TM.

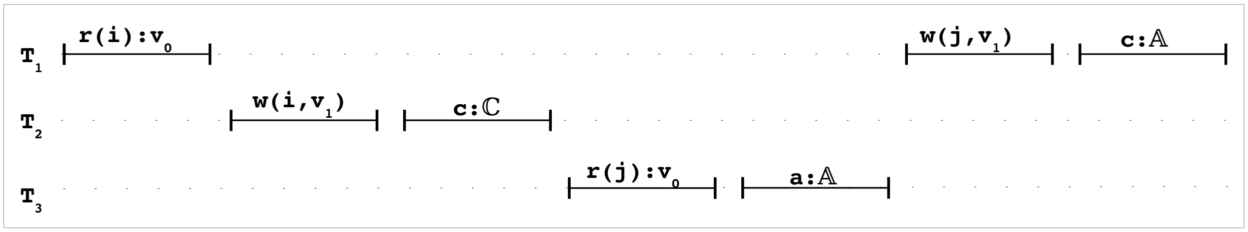
1. Invoke $init_{T_1}()$ and wait for the response.
2. Invoke $read_{T_1}(i)$ and wait for the response.
3. Invoke $init_{T_2}()$ and wait for the response.
4. Invoke $write_{T_2}(i, v_1)$ and wait for the response.
5. Invoke $commit_{T_2}()$ and wait for the response.
6. Invoke $init_{T_3}()$ and wait for the response.
7. Invoke $read_{T_3}(j)$ and wait for the response.
8. Invoke $abort_{T_3}()$ and wait for the response.
9. Invoke $write_{T_1}(j, v_1)$ and wait for the response.
10. Invoke $commit_{T_1}()$ and wait for the response.

The resulting history H_1 for this scenario is depicted in Figure 15(a). The initializing transaction T_0 (that initializes every location to v_0) and also the initializing operations of each transaction are elided for brevity.

The transaction T_1 first invokes the init and then a read operation on the location i. As TM is progressive and T_1 is not in conflict with any other transaction, TM does not forcefully abort the read operation. Therefore, it returns a value. As TM is opaque, there should be a justifying history S for the current execution history (after the read operation returns). As the initializing transaction T_0 is executed before T_1, the real-time-order property requires T_0 to be ordered before T_1 in S. The transaction T_0 writes the initial value v_0 to every location and commits. Thus, the read operation returns v_0.

Then, the transaction T_2 invokes the init and then a write operation to i with the value v_1 and then invokes the commit operation. As TM is invisible-reads, the read operation of T_1 is invisible to T_2. Thus, T_2 does not observe any inconsistency and as TM is progressive, both the write and commit operations are successful.

Next, the transaction T_3 invokes the init and then a read operation on the location j and then invokes the abort operation. As there are no conflicting operations on j and TM is progressive, the read operation is not forcefully aborted. Therefore, it returns a value. As TM is opaque, there should be a justifying history S' for the current execution history (after the read operation returns). As the initializing transaction T_0 is executed before T_3, the real-time-order property requires T_0 to be ordered before T_3 in S'. The transaction



(a) H_1

(b) H_2

Figure 15: The execution histories constructed by the scenarios in the proof of Theorem 62. The letters r, w, c, and a abbreviate read, write, commit and abort operations. The initializing transaction T_0 (that initializes every location to v_0) and also the initializing operations of each transaction are elided.

T_0 is the only committed transaction that has written to j. Thus, the read operation returns the initial value v_0.

Next, the transaction T_1 invokes a write operation on location j with the value v_1. When this write operation is invoked, neither T_2 nor T_3 are alive and TM is progressive. Therefore, TM does not forcefully abort the write operation. Finally, T_1 invokes the commit operation. We show that the commit operation aborts i.e. returns $\mathbb{A}$. Let us assume otherwise, i.e. T_1 commits. As TM is opaque, there is a justifying history S'' for H_1 i.e. S'' is a sequential history that is equivalent to H_1, is real-time-preserving and is a member of transactional sequential specification. As T_2 is executed before T_3 in H_1, the real-time-preservation property requires T_2 to be before T_3 in S''. Thus, there are three possible transaction orderings for S''. We show that none of them is a justifying history.

- $S'' = H_1|T_0 \cdot H_1|T_1 \cdot H_1|T_2 \cdot H_1|T_3$

  We have that $Visible(S'', T_3)|j = write_{T_0}(j, v_0) \cdot write_{T_1}(j, v_1) \cdot read_{T_3}(j){:}v_0 \notin SeqSpec(j)$. The read operation is expected to return the value v_1 but has returned v_0. Thus, S'' is not a justifying history.

- $S'' = H_1|T_0 \cdot H_1|T_2 \cdot H_1|T_1 \cdot H_1|T_3$

  Similar to the previous case, $Visible(S'', T_3)|j = write_{T_0}(j, v_0) \cdot write_{T_1}(i, v_1) \cdot read_{T_3}(j){:}v_0 \notin SeqSpec(j)$. The read operation is expected to return the value v_1 but has returned v_0. Thus, S'' is not a justifying history.

- $S'' = H_1|T_0 \cdot H_1|T_2 \cdot H_1|T_3 \cdot H_1|T_1$

  We have that $Visible(S'', T_1)|i = write_{T_0}(i, v_0) \cdot write_{T_2}(i, v_1) \cdot read_{T_1}(i){:}v_0 \notin SeqSpec(i)$. The read operation is expected to return the value v_1 but has returned v_0. Thus, S'' is not a justifying history.

Thus, we arrive at a contradiction. Therefore, we conclude that the commit operation of T_1 returns $\mathbb{A}$.

Now, we argue that the commit operation of T_1 should access a shared object that T_2 mutates. Assume otherwise i.e. the commit operation of T_1 does not access any shared object that T_2 mutates. Thus, T_2 is invisible to T_1. As TM is invisible-reads, the read operation of T_3 does not mutate any shared objects. Furthermore, T_3 is a read-only transaction. Thus, its abort operation does not mutate any shared objects. Therefore, T_3 is invisible to other transactions. As T_2 and T_3 are invisible to T_1, removing them from the client scenario does not affect the responses that T_1 receives. Therefore, the execution of T_1 alone results in the execution history H_2 depicted in Figure 15(b). As there is no conflicting transaction and TM is progressive, TM cannot forcefully abort the commit operation of T_1. The commit operation should have returned $\mathbb{C}$ but has returned $\mathbb{A}$ that is a contradiction. Therefore, we conclude that the commit operation of T_1 accesses a shared object that T_2 mutates.

In the above client scenario, the read set of T_1 was the singleton set i. The scenario can be trivially extended to an arbitrary location k in a read set $\mathcal{R}$.

1. Invoke $init_{T_1}()$ and wait for the response.
2. For each $i \in \mathcal{R}$:
    2.1. Invoke $read_{T_1}(i)$ and wait for the response.
3. Invoke $init_{T_2^k}()$ and wait for the response.
4. Invoke $write_{T_2^k}(k, v_1)$ and wait for the response.
5. Invoke $commit_{T_2^k}()$ and wait for the response.
6. Invoke $init_{T_3}()$ and wait for the response.
7. Invoke $read_{T_3}(j)$ and wait for the response.
8. Invoke $abort_{T_3}()$ and wait for the response.
9. Invoke $write_{T_1}(j, v_1)$ and wait for the response.
10. Invoke $commit_{T_1}()$ and wait for the response.

A similar reasoning concludes that for every $k \in \mathcal{R}$, the commit operation of T_1 accesses a shared object that T_2^k mutates.

The transactions T_2^k (for $k \in \mathcal{R}$) access disjoint locations. As TM is strictly disjoint-access-parallel, the transactions T_2^k (for $k \in \mathcal{R}$) access disjoint shared objects. Thus, the commit operation of T_1 accesses a separate shared object for every $k \in \mathcal{R}$. Therefore, the commit operation of T_1 accesses at least $|\mathcal{R}|$ shared objects. Therefore, the time complexity of the commit operation of T_1 is $\Omega(|\mathcal{R}|)$.

□

Theorem 63 *The time complexity of the commit operation of every opaque, progressive, and invisible-reads TM algorithm that stores information about a constant number of locations in each shared object is* $\Omega(|\mathcal{R}|)$ *where* $\mathcal{R}$ *is the read set.*

The proof of this theorem uses the same client scenario as the proof of Theorem 62. The main difference is the final step of reasoning. As information about a constant number c of locations can be obtained from each shared object, the commit operation of T_1 has to read at least $|\mathcal{R}|/c$ shared objects.

Proof.
The proof of this theorem flows similar to the proof of Theorem 62 to the point that we have that
(1) For every $k \in \mathcal{R}$, the commit operation of T_1 reads a shared object that T_2^k mutates.

We have that
(2) Each transaction T_2^k accesses a separate location k.
From the premises we have that

(3) Information about a constant number c of locations is stored in each shared object.
From 2 and 3, we have that
(4) At most c of the set of writer transactions $T_2^k, k \in \mathcal{R}$ can write to the same shared object.

From 1 and 4, we have
The commit operation of T_1 has to read at least $|\mathcal{R}|/c$ shared objects.
Therefore,
The time complexity of the commit operation of T_1 is $\Omega(|\mathcal{R}|)$. □

We restate Theorem 3 of [2] below. Our Theorem 63 generalizes this theorem by dropping the single-version requirement. Note that the assumption about limited capacity of shared objects is stated before the theorem in [2] and explicitly in the theorem here.

Theorem 64 ***(Theorem 3 of [2])*** *The time complexity of every opaque, progressive, single-version and invisible-reads TM algorithm that stores information about a constant number of locations in each shared object is* $\Omega(|I|)$ *(where* I *is the set of locations).*

References

[1] Hagit Attiya, Eshcar Hillel, and Alessia Milani. Inherent limitations on disjoint-access parallel implementations of transactional memory. *Theory of Computing Systems*, 49(4), 2011.

[2] R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *PPOPP*, 2008.

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

[4] Rachid Guerraoui and Michal Kapalka. On obstruction-free transactions. In *SPAA*, 2008.

[5] Dmitri Perelman, Rui Fan, and Idit Keidar. On maintaining multiple versions in stm. In *PODC*, 2010.