Decomposing Opacity  
(Appendix)

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

Contents

1 Histories 2  
2 Opacity 5  
3 Markability 7  
4 Marking Theorem 10  
5 Synchronization Object Types 23  
6 Marking TL2 37  
7 Marking DSTM (visible reads) 57  
8 Marking NORec 59  
9 The Cost of Read Validation 61
1 Histories

Strings. We use $|s|$ to denote the size of the string $s$. If $s_1$ and $s_2$ are strings, we write $s_1 \subseteq s_2$ if $s_1$ is a subsequence of $s_2$. For example, $bd \subseteq abede$. Let $s$ be an isogram (i.e. contains no repeating occurrence of the alphabet.) For any $s_1, s_2 \subseteq s$, we write $s_1 \triangleleft s_2$ iff the last element of $s_1$ occurs before the first element of $s_2$ in $s$. For example, $ab \triangleleft_{abede} de$. 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 $Res = \{ret(l \triangleright v) \mid l \in Label, v \in V \cup \{\bot, \top\}\}$. ($\bot$ and $\top$ 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 $l \triangleright o.n_T(v)$ 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 $obj_X$, $name_X$, $trans_X$, $arg1_X$, $arg2_X$, $retu_X$ 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. Similarly, $iEv$ 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 $\prec_X$, $\preceq_X$, $\succeq_X$, $\nsucc_X$ on $Labels(X)$ as follows: First, $l_1 \prec_X l_2$ iff $rEv(l_1) \prec_X iEv(l_2)$. $l_1 \preceq_X l_2$ iff $l_1 \prec_X l_2$ or $l_1 = l_2$. Second, $l_1 \succeq_X l_2$ iff $l_1 \nsucc_X l_2$ and $l_2 \nsucc_X l_1$. Third, $l_1 \nsucc_X l_2$ iff $l_1 \sim_X l_2$ and $l_1 \sim_X l_2$ and $l_2 \sim_X l_1$. From the definition of $Sequential$ we have that $X \in Sequential$ iff $\forall l, l' \in X, l \preceq_X l' \lor l' \preceq_X l$. For an execution history $X$, we define the real-time relations $\ll_X$ and $\gg_X$ as follows. First, $T \ll_X T'$ iff $X|T \ll_X X|T'$. Second, $T \gg_X T'$ iff $T \gg_X T' \lor T = T'$.

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 $\bot$. The method $write_t(i, v)$ writes $v$ to location $i$ and returns $ok$ or aborts $t$ and returns $\bot$. The method $commit_t()$ tries to commit transaction $t$. If $t$ is successfully committed, it returns $C$; otherwise, it returns $\bot$. The method $abort_t()$ aborts $t$ and returns $\bot$. The object $mem$ can be implicit, that is $read_t(i)$ abbreviates $mem.read_t(i)$. The values $ok$, $\bot$, $C$ 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 $v_0$. $H_{Init} =$
For every execution history $X$ and method calls $l$ and $l'$, if $l \prec_X l'$ then $\neg(l' \prec_X l) \land \neg(l' \sim_X l) \land \neg(l' = l)$

For every execution history $X$ and method calls $l$, $l$, and $l''$, if $l \prec_X l'$ and $l' \prec l''$ then $l \prec_X l''$

For every execution history $X$ and method calls $l_1$, $l_2$, $l_3$, and $l_4$, if $l_1 \prec_X l_2$, $l_2 \preceq_X l_3$, and $l_3 \prec_X l_4$ then $l_1 \prec_X l_4$

For every execution history $X$ and method calls $l$ and $l'$, if $l \in X$, and $l' \in X$, then $(l \prec_X l') \lor (l' \sim_X l) \lor (l \sim_X l') \lor (l = l')$

For every execution history $X$ and method calls $l$ and $l'$, if $l \prec_X l'$ then $l \in X$, and $l' \in X$.

For every execution history $X$ and method calls $l$, $l'$, and $l''$ if $l \prec_X l'$ and $\text{inv}(l') \prec_X \text{inv}(l'')$ then $l \prec_X l''$.

For every execution history $X$ and method calls $l$, $l'$, and $l''$ if $\text{ret}(l) \prec_X \text{ret}(l')$ 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. $rEv(l) \prec_X iEv(l')$

As $X$ is a valid history, we have

4. $iEv(l) \prec_X rEv(l)$

5. $iEv(l') \prec_X rEv(l')$

From 4, 3, and 5, we have

6. $iEv(l) \prec_X rEv(l')$

From 5, we have

7. $\neg(rEv(l') \prec_X iEv(l))$

From 7, and definition of $\sim_X$, we have
From [3] and [7], we have

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

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

From $(2)$, we have

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

From $(3)$, we have

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

From $(6)$, we have

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

From $(7)$, we have

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

From $(1)$, $(8)$, and $(5)$, we have

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

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 $\sim_X$.

Lemma 7
Straightforward from the definition of $\prec_X$ and $\sim_X$. 

4
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 \( \pi \), let \( H(\pi) \) denote the set of complete transaction histories that \( \pi \) results.

If H is a transaction history and p is a predicate on transaction identifiers, we define \( \text{filter}(H, p) \) 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, \( \text{Visible}(S, T) \), is the sequence of committed transactions before T in S and T itself. The sequential specification of a location i, \( \text{SeqSpec}(i) \), 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, \( \text{Visible}(S, T)|i \) 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.
\[
TReads(H) = \\
\{ R | R \in H \land obj_H(R) = \text{mem} \land \text{name}_H(R) = \text{read} \land \text{retv}_H(R) \neq A \}
\]
\[
TWrites(H) = \\
\{ W | W \in H \land obj_H(W) = \text{mem} \land \text{name}_H(W) = \text{write} \land \text{retv}_H(W) \neq A \}
\]
\[
\text{Commits}(H) = \\
\{ C | C \in H \land obj_H(C) = \text{mem} \land \text{name}_H(C) = \text{commit} \}
\]
\[
\text{Trans}(H) = \\
\{ T | \exists l \in H : \text{trans}_H(l) = T \}
\]
\[
\text{TSequential} = \\
\{ S \in T\text{History} | \preceq_S \text{ is a total order of } \text{Trans}(S) \}
\]
\[
\text{Committed}(H) = \\
\{ T | \exists l \in \text{Commits}(H) \land \text{retv}_H(l) = C \}
\]
\[
\text{Aborted}(H) = \\
\{ T | \exists l \in H : \text{obj}_H(l) = \text{mem} \land \text{trans}_H(l) = T \land \text{retv}_H(l) = A \}
\]
\[
\text{Completed}(H) = \\
\text{Committed}(H) \cup \text{Aborted}(H)
\]
\[
\text{Live}(H) = \\
\text{Trans}(H) \setminus \text{Completed}(H)
\]
\[
\text{TComplete} = \\
\{ H \in T\text{History} | \forall T \in \text{Trans}(H) : T \in \text{Completed}(H) \}
\]
\[
\text{CommitPending}(H) = \\
\{ T \in \text{Live}(H) | \exists l \in H : \text{obj}_H(l) = \text{mem} \land \text{trans}_H(l) = T \land \text{name}_H(l) = \text{commit} \}
\]
\[
\text{TExtension}(H) = \\
\{ H' \in T\text{History} | \exists H'' : H' = H \cdot H'' \\
\quad \text{Trans}(H'') \subseteq \text{Trans}(H) \land \forall T : ||H''[T]|| \leq 1 \land \\
\quad \text{Live}(H) \setminus \text{CommitPending}(H) \subseteq \text{Aborted}(H') \land \\
\quad \text{CommitPending}(H) \subseteq \text{Completed}(H') \}
\]
\[
\text{Visible}(S, T) = \\
\text{filter} (S, \lambda T'. (T' = T) \lor ((T' \not\preceq_S T) \land T' \in \text{Committed}(S)))
\]
\[
\text{NoWriteBetween}_S(W, R) = \\
\forall W' \in \text{TWrites}(S) : W' \preceq_S W \lor R \not\preceq_S W'
\]
\[
\text{SeqSpec}(i) = \\
\{ S \in \text{Sequential} | \forall R \in \text{TReads}(S) : \exists W \in \text{TWrites}(S) : \\
\quad W \preceq_S R \land \text{NoWriteBetween}_S(W, R) \land \\
\quad \text{retv}_S(R) = \text{arg}_2 S(W) \}
\]
\[
\text{TSpec} = \\
\{ S \in \text{TSequential} \cap \text{TComplete} | \forall T \in S : \forall i \in I : \\
\quad (\text{Visible}(S, T) \mid i) \in \text{SeqSpec}(i) \}
\]
\[
\text{FinalStateOpaque} = \\
\{ H \in T\text{History} | \exists H' \in \text{TExtension}(H) : \exists S \in \text{TSequential} : \\
\quad H' \equiv S \land \preceq_{H'} \subseteq \preceq_S \land S \in \text{TSpec} \}
\]

Figure 1: 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.
Committed$(H) = \{ T \mid \exists l \in H: \text{obj}_H(l) = \text{mem} \land \text{trans}_H(l) = T \land \text{retv}_H(l) = C \}$

Aborted$(H) = \{ T \mid \exists l \in H: \text{obj}_H(l) = \text{mem} \land \text{trans}_H(l) = T \land \text{retv}_H(l) = A \}$

Completed$(H) = \text{Committed}(H) \cup \text{Aborted}(H)$

Live$(H) = \text{Trans}(H) \setminus \text{Completed}(H)$

CommitPending$(H) = \{ T \in \text{Live}(H) \mid \exists l \in H: \text{obj}_H(l) = \text{mem} \land \text{trans}_H(l) = T \land \text{name}_H(l) = \text{commit} \}$

$T\text{Extension}(H) = \{ H' \in T\text{History} \mid \exists H'': H' = H \cdot H'' \land \text{Trans}(H'') \subseteq \text{Trans}(H) \land \forall T: |H''|T| \leq 1 \land \text{Live}(H) \setminus \text{CommitPending}(H) \subseteq \text{Aborted}(H') \land \text{CommitPending}(H) \subseteq \text{Completed}(H') \}$

$T\text{Reads}(H) = \{ R \mid R \in H \land \text{obj}_H(R) = \text{mem} \land \text{name}_H(R) = \text{read} \land \text{retv}_H(R) \neq \text{A} \}$

$T\text{Writes}(H) = \{ W \mid W \in H \land \text{obj}_H(W) = \text{mem} \land \text{name}_H(W) = \text{write} \land \text{retv}_H(W) \neq \text{A} \}$

LocalTReads$(H) = \{ R \mid R \in T\text{Reads}(H) \land \exists W \in T\text{Writes}(H): \text{trans}_H(R) = \text{trans}_H(W) \land \text{arg}_1(H)(R) = \text{arg}_1(H)(W) \land W \prec_H R \}$

GlobalTReads$(H) = T\text{Reads}(H) \setminus \text{LocalTReads}(H)$

LocalTWrites$(H) = \{ W \mid W \in T\text{Writes}(H) \land \exists W' \in T\text{Writes}(H): \text{trans}_H(W) = \text{trans}_H(W') \land \text{arg}_1(H)(W) = \text{arg}_1(H)(W') \land W \prec_H W' \}$

GlobalTWrites$(H) = T\text{Writes}(H) \setminus \text{LocalTWrites}(H)$

Writing$_H(i) = \{ T \in \text{Trans}(H) \mid \exists l \in T\text{Write}(H): \text{arg}_1(H)(l) = i \land \text{trans}_H(l) = T \land T \in \text{Committed}(H) \}$

Figure 2: Basic Definitions
Marking($H$) = \{\sqsubseteq | \\
\forall T_1, T_2, T_3 \in Trans(H): \\
(T_1 \sqsubseteq T_2 \lor T_2 \sqsubseteq T_1) \land \\
(T_1 \sqsubseteq T_2 \land T_2 \sqsubseteq T_1) \Rightarrow (T_1 = T_2) \land \\
(T_1 \sqsubseteq T_2) \land (T_2 \sqsubseteq T_3) \Rightarrow (T_1 \sqsubseteq T_3) \land \\
\forall R, T: \text{Let } i = \text{arg}1_H(R): (R \in GlobalTRead(H) \land T \in Writers_H(i)) \Rightarrow \\
(R \sqsubseteq T \lor T \sqsubseteq R) \land \\
(R \sqsubseteq T \Rightarrow \neg T \sqsubseteq R) \land (T \sqsubseteq R \Rightarrow \neg R \sqsubseteq T)\} \\
NoWriteBetween_H(W, R) \Leftrightarrow \\
\forall W' \in TWrites(H): W' \preceq_H W \lor R \prec_H W' \\
LocalWriteObs(H) \Leftrightarrow \\
\forall R \in LocalTReads(H): \text{Let } T = \text{trans}_H(R), i = \text{arg}1_H(R), H' = H|T|i: \\
\exists W \in TWrites(H'): W \prec_{H'} R \land NoWriteBetween_H(W, R) \land \text{retv}_H(R) = \text{arg}2_{H'}(W) \\
NoWriterBetween_{H,i}(x, \sqsubseteq, x') \Leftrightarrow \\
\forall T \in Writers_H(i): T \sqsubseteq x \lor x' \sqsubseteq T \\
LastPreAccessor_{H,\sqsubseteq}(T', R) \Leftrightarrow \text{Let } i = \text{arg}1_H(R), T = \text{trans}_H(R): \\
T' \in Writers_H(i) \land T' \neq T \land T' \sqsubseteq R \land NoWriterBetween_{H,i}(T', \sqsubseteq, R) \\
GlobalWriteObs(H, \sqsubseteq) \Leftrightarrow \\
\forall R \in GlobalTReads(H): \exists W \in GlobalTWrites(H): \text{Let } T' = \text{trans}_H(W): \\
LastPreAccessor_{H,\sqsubseteq}(T', R) \land \text{arg}1_H(R) = \text{arg}1_H(W) \land \text{retv}_H(R) = \text{arg}2_H(W) \\
WriteObs(H, \sqsubseteq) \Leftrightarrow \\
LocalWriteObs(H) \land GlobalWriteObs(H, \sqsubseteq) \\
ReadPres(H, \sqsubseteq) \Leftrightarrow \\
\forall R \in GlobalTReads(H): \text{Let } i = \text{arg}1_H(R), T = \text{trans}_H(R): \\
NoWriterBetween_{H,i}(R, \sqsubseteq, T) \land NoWriterBetween_{H,i}(T, \sqsubseteq, R) \\
RealTimePres(H, \sqsubseteq) \Leftrightarrow \\
\preceq_H \sqsubseteq \sqsubseteq \\
FinalStateMarkable = \{H \in T\text{History} \mid \exists H' \in T\text{Extension}(H): \exists \sqsubseteq \in \text{Marking}(H'): \\
\text{ReadPres}(H', \sqsubseteq) \land \text{WriteObs}(H', \sqsubseteq) \land \text{RealTimePres}(H', \sqsubseteq)\} \\
\text{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
\[ \exists l \in X: \text{obj}_X(l) = o \land \text{name}_X(l) = n \land \text{trans}_X(l) = T \land \text{arg}_1_X(l) = v_1 \land \text{ret}_X(l) = v_2 \]
and similarly for universal quantification.

We also use \( W, R \) to denote labels.

Lemma 8 For all \( S \in T_{\text{Sequential}}, T \in S, S' = \text{Visible}(S,T), \text{ and } T', T'' \in S', \) we have \( T' \preceq_S T'' \iff T' \preceq_S T'' \).

Proof.

\[ T' \preceq_S T'' \]
\[ \iff S|T' \triangleleft_S S'|T'' \lor T' = T'' \]
\[ \iff S|T' \triangleleft_S S|T'' \lor T' = T'' \]
\[ \iff T' \preceq_S T'' \]

In these four steps we apply:
1) the definition of \( \preceq_S \),
2) that the definition of \( \text{Visible}(S,T) \) implies both \( S'|T' = S|T' \) and \( S'|T'' = S|T'' \),
3) \( S' \subseteq S \), and
4) the definition of \( \preceq_S \).
Lemma 9 For all $S \in T\text{Sequential} $, $ T \in S, i \in I, v, v' \in V $, $ R = \text{ready}_T(i); v \in \text{GlobalReads}(S), S' = \text{Visible}(S,T), T' \in S'$, and $ W' = \text{write}_T(i,v') \in \text{GlobalWrites}(S) $, we have

$$ \text{NoWriteBetween}_{S'}(W', R) \iff \text{NoWriterBetween}_{S,i}(T', \preceq_S, T) $$

Proof.

$$ \text{NoWriteBetween}_{S'}(W', R) \iff \forall W'' \in \text{Writes}(S') \cap W'' \preceq_{(S')} W' \lor R \preceq_{(S')} W'' $$

$$ \iff \forall T'' \in S': \forall i \in S': \forall v'' \in V: \forall W'' = \text{write}_{T''}(i,v'') \in S': W'' \preceq_{S'} W' \lor R \preceq_{S'} W'' $$

$$ \iff \forall T'' \in S': \forall v'' \in V: \forall W'' = \text{write}_{T''}(i,v'') \in S': T'' \preceq_{S'} T' \lor T'' \preceq S T'' $$

$$ \iff \forall T'' \in S': \forall v'' \in V: \forall W'' = \text{write}_{T''}(i,v'') \in S': T'' \preceq_{S} T' \lor T'' \preceq_{S} T'' $$

$$ \iff \forall T'' \in S': \forall v'' \in V: \forall W'' = \text{write}_{T''}(i,v'') \in S':$$

$$ \left[ (T'' = T) \lor (T'' \preceq_{S} T \land T'' \in \text{Committed}(S)) \land [T'' \preceq_{S} T] \right] \Rightarrow T'' \preceq_{S} T' $$

$$ \iff \forall T'' \in S': \forall v'' \in V: \forall W'' = \text{write}_{T''}(i,v'') \in S':$$

$$ (T'' \in \text{Committed}(S) \land T'' \preceq_{S} T) \Rightarrow T'' \preceq_{S} T' $$

$$ \iff \forall T'' \in \text{Writers}(i): T'' \preceq_{S} T \Rightarrow T'' \preceq_{S} T' $$

$$ \iff \forall T'' \in \text{Writers}(i): T'' \preceq_{S} T' \lor T'' \preceq_{S} T'' $$

$$ \iff \text{NoWriterBetween}_{S,i}(T', \preceq_{S}, T) $$

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

$\square$
**Lemma 10** $T_{\text{Sequential}} \subseteq S_{\text{Sequential}}$

*Proof.* Straightforward from definitions of $T_{\text{Sequential}}$, $T_{\text{History}}$ and $S_{\text{Sequential}}$. $\square$

**Lemma 11** $\forall i \in I : \forall v, v' \in V : \forall T, T' \in \text{Trans} : \text{if } R = \text{read}_T(i) : v, W = \text{write}_T(i, v), W' = \text{write}_T(i, v'), S \in T_{\text{Sequential}}, W \prec_S R, \text{NoWriteBetween}_S(W, R) \text{ and } W' \prec_S R, \text{ then } T = T'$. 

*Proof.* Suppose (1) $S \in T_{\text{Sequential}}$, (2) $W \prec_S R$, (3) $\text{NoWriteBetween}_S(W, R)$ and (4) $W' \prec_S R$. From [1] and Lemma 10, we have (5) $S \in S_{\text{Sequential}}$. From [4] and [5], we have (6) $\neg(R \prec_S W')$. From [3] we have (7) $W' \preceq_S W \lor R \prec_S W'$. 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 $S \in T_{Sequential}$. We have:

\[
\forall T \in S: \forall i \in I: \forall v \in V: \forall R = read_T(i)v \in LocalReads(S):
\exists T' \in Visible(S, T): \exists W = write_T(i, v) \in Visible(S, T):
W \prec_{(Visible(S, T) | i)} R \land NoWriteBetween_{(Visible(S, T) | i)}(W, R)
\iff S \in LocalTSeqSpec
\]

Proof. Suppose $S \in T_{Sequential}$. Thus, from Lemma 10 we have $S \in Sequential$. Let $S' = Visible(S, T)$. From $S \in T_{Sequential}$ and Lemma 3 we have $S' \in T_{Sequential}$. Thus, from Lemma 10 we have $S' \in Sequential$. From the definition of $Visible$, we have $S'|T = S|T$.

\[
\forall T \in S: \forall i \in I: \forall v \in V: \forall R = read_T(i)v \in LocalReads(S):
\exists T' \in S': \exists W = write_T(i, v) \in S':
W \prec_{(S' | i)} R \land NoWriteBetween_{(S' | i)}(W, R)
\iff \forall T \in S: \forall i \in I: \forall v \in V: \forall R = read_T(i)v \in LocalReads(S):
\exists v' \in V: \exists W' = write_T(i, v') \in S': W' \prec_{S} R \land
\exists T' \in S': \exists W = write_T(i, v) \in S':
\iff \forall T \in S: \forall i \in I: \forall v \in V: \forall R = read_T(i)v \in LocalReads(S):
\exists v' \in V: \exists W' = write_T(i, v') \in S': W' \prec_{S} R \land
\exists T' \in S': \exists W = write_T(i, v) \in S':
\iff \forall T \in S: \forall i \in I: \forall v \in V: \forall R = read_T(i)v \in LocalReads(S):
\exists v' \in V: \exists W' = write_T(i, v') \in S': W' \prec_{S'} R \land
\exists T' \in S': \exists W = write_T(i, v) \in S':
\iff \forall T \in S: \forall i \in I: \forall v \in V: \forall R = read_T(i)v \in LocalReads(S):
\exists v' \in V: \exists W' = write_T(i, v') \in S': W' \prec_{S'} R \land
\exists W = write_T(i, v) \in S':
W \prec_{(S' | i)} R \land NoWriteBetween_{(S' | i)}(W, R)
\iff \forall T \in S: \forall i \in I: \forall v \in V: \forall R = read_T(i)v \in LocalReads(S):
\exists W = write_T(i, v) \in S':
W \prec_{(S' | i)} R \land NoWriteBetween_{(S' | i)}(W, R)
\iff \forall T \in S: \forall i \in I: \forall v \in V: \forall R = read_T(i)v \in LocalReads(S):
\exists W = write_T(i, v) \in S':
W \prec_{(S' | i)} R \land NoWriteBetween_{(S' | i)}(W, R)
\iff \forall T \in S: \forall i \in I: \forall v \in V: \forall R = read_T(i)v \in LocalReads(S):
\exists W = write_T(i, v) \in S':
W \prec_{(S' | i)} R \land NoWriteBetween_{(S' | i)}(W, R)
\end{align*}
\[ \forall T \in S: \forall i \in I: \forall v \in V: \forall R = read_T(i): v \in LocalReads(S): \]
\[ \exists W = write_T(i, v) \in S: \]
\[ W \prec_S R \land NoWriteBetween(S \mid i)(W, R) \]

\[ \forall T \in S: \forall i \in I: \forall v \in V: \forall R = read_T(i): v \in LocalReads(S): \]
\[ \exists W = write_T(i, v) \in S: \]
\[ W \prec_S R \land \forall W' \in Writes(S' \mid i): W' \preceq_{(S' \mid i)} W \lor R \prec_{(S' \mid i)} W' \]

\[ \forall T \in S: \forall i \in I: \forall v \in V: \forall R = read_T(i): v \in LocalReads(S): \]
\[ \exists W = write_T(i, v) \in S: \]
\[ W \prec_S R \land \forall W' \in Writes(S' \mid i): W' \prec_{(S' \mid i)} W \prec_{(S' \mid i)} R \]

\[ \forall T \in S: \forall i \in I: \forall v \in V: \forall R = read_T(i): v \in LocalReads(S): \]
\[ \exists W = write_T(i, v) \in S: \]
\[ W \prec_S R \land \forall W' \in Writes(S \mid i): W \prec_{(S \mid i)} W' \prec_{(S \mid i)} R \]

\[ \forall T \in S: \forall i \in I: \forall v \in V: \forall R = read_T(i): v \in LocalReads(S): \]
\[ \exists W = write_T(i, v) \in S: \]
\[ W \prec_S R \land \forall W' \in Writes(S \mid i): W' \prec_{(S \mid i)} W \lor R \prec_{(S \mid i)} W' \]

\[ \forall T \in S: \forall i \in I: \forall v \in V: \forall R = read_T(i): v \in LocalReads(S): \]
\[ \exists W = write_T(i, v) \in S: \]
\[ W \prec_{S[T \mid i]} R \land \forall W' \in Writes(S[T \mid i]): W' \preceq_{(S[T \mid i]} W \lor R \prec_{(S[T \mid i]} W' \]

\[ \forall T \in S: \forall i \in I: \forall v \in V: \forall R = read_T(i): v \in LocalReads(S): \]
\[ \exists W = write_T(i, v) \in S[T \mid i]: \]
\[ W \prec_{S[T \mid i]} R \land NoWriteBetween_{(S[T \mid i]}(W, R) \]

\[ S \in LocalTSeqSpec \]

In these twenty steps, we apply: 1) the definition of LocalReads,
2) the definition of \textit{Visible},
3) $S'|T = S|T$ and that both $W'$ and $R$ are by $T$,
4) that both $W'$ and $R$ are on $i$,
6) duplicate conjunction,
7) the definition of \textit{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 \textit{NoWriteBetween},
11) first-order logic,
12) $(S' | i) \in \textit{Sequential}$,
13) from $(S' | i) \in \textit{TSequential}$, $R$ and $W$ are by transaction $T$ and $W'$ is between them, we have $W'$ is by $T$,
14) $S'|T = S|T$,
15) from $(S | i) \in \textit{TSequential}$, $R$ and $W$ are by transaction $T$ and $W'$ is between them, we have $W'$ is by $T$.
16) first-order logic,
17) $(S | i) \in \textit{Sequential}$,
18) $(S | i) \in \textit{Sequential}$, $\text{trans}_H(R) = \text{trans}_H(W) = T$ and $\text{arg}_1_H(R) = \text{arg}_1_H(W) = i$,
19) the definition of \textit{NoWriteBetween},
20) the definition of \textit{LocalTSeqSpec}.

□
Lemma 13 Suppose $S \in T_{\text{Sequential}} \cap T_{\text{Complete}}$. We have:

$$S \in T_{\text{SeqSpec}}$$

$$\iff S \in \text{LocalTSeqSpec} \land$$

$$\forall T \in S: \forall i \in I: \forall v \in V: \forall R = \text{read}_T(i): v \in \text{GlobalReads}(S):$$

$$\exists T' \in \text{Committed}(S): \exists W = \text{write}_{T'}(i, v) \in \text{GlobalWrites}(S):$$

$$(T' \prec_s T) \land \text{NoWriterBetween}_{S,i}(T', \prec_s S, T)$$

Proof. Suppose $S \in T_{\text{Sequential}} \cap T_{\text{Complete}}$. From $S \in T_{\text{Sequential}}$ and Lemma 8 we have $\text{Visible}(S, T) \in T_{\text{Sequential}}$.

$$S \in T_{\text{SeqSpec}}$$

$$\iff \forall T \in S: \forall i \in I: (\text{Visible}(S, T) \mid i) \in \text{SeqSpec}(i)$$

$$\iff \forall T \in S: \forall i \in I:$$

$$\forall T'' \in (\text{Visible}(S, T) \mid i): \forall v \in V: \forall R = \text{read}_{T''}(i): v \in (\text{Visible}(S, T) \mid i):$$

$$\exists T' \in \text{Visible}(S, T) \mid i): \exists W = \text{write}_{T'}(i, v) \in \text{Visible}(S, T) \mid i):$$

$$W \prec (\text{Visible}(S, T) \mid i) R \land \text{NoWriteBetween}_{\text{Visible}(S, T) \mid i}(W, R)$$

$$\iff \forall T \in S: \forall i \in I:$$

$$\forall T'' \in \text{Visible}(S, T): \forall v \in V: \forall R = \text{read}_{T''}(i): v \in \text{Visible}(S, T):$$

$$\exists T' \in \text{Visible}(S, T): \exists W = \text{write}_{T'}(i, v) \in \text{Visible}(S, T):$$

$$W \prec (\text{Visible}(S, T) \mid i) R \land \text{NoWriteBetween}_{\text{Visible}(S, T) \mid i}(W, R)$$

$$\iff \forall T \in S: \forall i \in I:$$

$$\forall v \in V: \forall R = \text{read}_{T}(i): v \in \text{LocalReads}(S):$$

$$\exists T' \in \text{Visible}(S, T): \exists W = \text{write}_{T'}(i, v) \in \text{Visible}(S, T):$$

$$W \prec (\text{Visible}(S, T) \mid i) R \land \text{NoWriteBetween}_{\text{Visible}(S, T) \mid i}(W, R)$$

$$\land$$

$$\forall T \in S: \forall i \in I:$$

$$\forall v \in V: \forall R = \text{read}_{T}(i): v \in \text{GlobalReads}(S):$$

$$\exists T' \in \text{Visible}(S, T): \exists W = \text{write}_{T'}(i, v) \in \text{Visible}(S, T):$$

$$W \prec (\text{Visible}(S, T) \mid i) R \land \text{NoWriteBetween}_{\text{Visible}(S, T) \mid i}(W, R)$$

$$\iff S \in \text{LocalTSeqSpec} \land$$

$$\forall T \in S: \forall i \in I:$$

$$\forall v \in V: \forall R = \text{read}_{T}(i): v \in \text{GlobalReads}(S):$$

$$\exists T' \in \text{Visible}(S, T): \exists W = \text{write}_{T'}(i, v) \in \text{Visible}(S, T):$$

$$W \prec (\text{Visible}(S, T) \mid i) R \land \text{NoWriteBetween}_{\text{Visible}(S, T) \mid i}(W, R)$$
In these thirteen steps, we apply:

1) the definition of \( TSeqSpec \) and \( S \in TSequential \cap TComplete \),
2) the definition of \( SeqSpec(i) \),
3) \( R \) and \( W \) access location \( i \),
4) that we can choose \( T'' = T \),
5) \( Reads(S) = LocalReads(S) \cup GlobalReads(S) \),
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, \( Visible(S,T) \in TSequential \), and \( R \in GlobalReads(Visible(S,T)) \) (because \( R \in GlobalReads(R) \) and \( Visible(S,T) \cap T = S \cap T \)),
9) Lemma 8,
10) \( T' \preceq \) and \( NoWriteBetween(Visible(S,T) \cap i)(W,R) \),
12) $T' \in \text{Visible}(S, T)$ and $(T' \prec_S T)$, and
13) the definition of $\text{Visible}(S, T)$. 

□
Lemma 14 (Invariance) If $H \equiv H'$, then $\text{Marking}(H) = \text{Marking}(H')$ and $\text{ReadPres}(H) = \text{ReadPres}(H')$ and $\text{WriteObs}(H) = \text{WriteObs}(H')$.

Proof. Immediate from the definitions of $\text{Marking}$, $\text{ReadPres}$, and $\text{WriteObs}$. \qed

Lemma 15 $\forall H \in T\text{History}: \forall \sqsubseteq \in \text{Marking}(H): \exists S \in T\text{Sequential}: H \equiv S \land \sqsubseteq H \subseteq \sqsubseteq S \land \sqsubseteq S \subseteq \sqsubseteq$.

Proof. Let $H \in T\text{History}$ and let $\sqsubseteq \in \text{Marking}(H)$. We have that $\sqsubseteq$ is a total order of $\text{Trans}$ so we can choose a permutation $\pi$ on $1..n$ such that $\forall i,j \in 1..n: (i < j) \Leftrightarrow (T_{\pi(i)} \sqsubseteq T_{\pi(j)})$. Define: $S = H|T_{\pi(1)}, \ldots, H|T_{\pi(n)}$. It is straightforward to prove that $S \in T\text{Sequential} \land H \equiv S \land \sqsubseteq H \subseteq \sqsubseteq S \land \sqsubseteq S \subseteq \sqsubseteq$. \qed

Lemma 16 Suppose $\sqsubseteq \in \text{Marking}(H) \land p_2 \notin \text{Writers}_H(i)$.
If $\text{NoWriterBetween}_{H,i}(T_1, \sqsubseteq, p_2)$ and $\text{NoWriterBetween}_{H,i}(p_2, \sqsubseteq, T_3)$, then $\text{NoWriterBetween}_{H,i}(T_1, \sqsubseteq, T_3)$.

Proof. $\text{NoWriterBetween}_{H,i}(T_1, \sqsubseteq, p_2) \land \text{NoWriterBetween}_{H,i}(p_2, \sqsubseteq, T_3)$

$\iff \forall T \in \text{Writers}_H(i): (T \sqsubseteq T_1 \lor p_2 \sqsubseteq T) \land (T \sqsubseteq p_2 \lor T_3 \sqsubseteq T)$

$\iff \forall T \in \text{Writers}_H(i): (T \sqsubseteq T_1 \land (T \sqsubseteq p_2 \lor T_3 \sqsubseteq T)) \lor$

$(p_2 \sqsubseteq T \land T \sqsubseteq p_2) \lor (p_2 \sqsubseteq T \land T_3 \sqsubseteq T)$

$\iff \forall T \in \text{Writers}_H(i): (T \sqsubseteq T_1) \lor (T_3 \sqsubseteq T)$

$\iff \text{NoWriterBetween}_{H,i}(T_1, \sqsubseteq, T_3)$

The first step uses the definition of $\text{NoWriterBetween}$. The second step uses $\land$ distribution over $\lor$. The third step simplifies the first disjunct using conjunction elimination, eliminates the second disjunct using $p_2 \notin \text{Writers}_H(i)$ and simplifies the third disjunct using conjunction elimination. The fourth step uses the definition of $\text{NoWriterBetween}$. \qed
Lemma 17 Suppose $S \in T_{Sequential} \cap T_{Complete}$. We have:

$$S \in T_{SeqSpec} \iff S \in FinalStateMarkable$$

(4.1)

Proof. Let $S \in T_{Sequential} \cap T_{Complete}$. From Lemma 13 the definition of $FinalStateMarkable$, and $S \in T_{Complete}$, we have that we must prove:

$$S \in LocalTSeqSpec \land \forall T \in S: \forall i \in I: \forall v \in V: \forall R = read_T(i): v \in GlobalReads(S):$$

$$\exists T' \in Committed(S): \exists W = write_T(i, v) \in GlobalWrites(S):$$

$$T' \in Writers_{S,i}(i) \land T' \neq T \land T' \sqsubset R \land NoWriterBetween_{S,i}(T', \sqsubseteq, R)$$

$$\iff \exists \subseteq \in Marking(S): \subseteq \sqsubseteq \sqsubseteq \land \subseteq \in ReadPres(S) \land \subseteq \in WriteObs(S)$$

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

$$\subseteq \in WriteObs(S)$$

$$\iff S \in LocalTSeqSpec \land \forall T \in Trans: \forall i \in I: \forall v \in V: \forall R = read_T(i): v \in GlobalReads(S):$$

$$\exists T' \in Trans: \exists W = write_T(i, v) \in GlobalWrites(S):$$

$$T' \in Committed(S) \land T' \neq T \land T' \sqsubset R \land NoWriterBetween_{S,i}(T', \sqsubseteq, R)$$

$$\iff S \in LocalTSeqSpec \land \forall T \in Trans: \forall i \in I: \forall v \in V: \forall R = read_T(i): v \in GlobalReads(S):$$

$$\exists T' \in Trans: \exists W = write_T(i, v) \in GlobalWrites(S):$$

$$T' \in Committed(S) \land T' \neq T \land T' \sqsubset R \land NoWriterBetween_{S,i}(T', \sqsubseteq, R)$$

We are now ready to prove the two directions of the equivalence.

$\Rightarrow$:

Assume that

$$S \in LocalTSeqSpec \land \forall T \in S: \forall i \in I: \forall v \in V: \forall R = read_T(i): v \in GlobalReads(S):$$

$$\exists T' \in Committed(S): \exists W = write_T(i, v) \in GlobalWrites(S):$$

$$(T' \preceq_S T) \land NoWriterBetween_{S,i}(T', \preceq_S, T)$$

Define:

$$p_1 \sqsubseteq p_2 \iff (p_1 \preceq_S p_2) \lor (trans_{S}(p_1) \preceq_S p_2) \lor (p_1 \preceq_S trans_{S}(p_2))$$

$$p_1 \sqsubseteq p_2 \iff p_1 \sqsubseteq \lor p_2 p_1 = p_2$$

We show that

$$\subseteq \in Marking(S) \land \preceq_S \subseteq \subseteq \land \subseteq \in ReadPres(S) \land S \in LocalTSeqSpec \land \forall T \in Trans: \forall i \in I: \forall v \in V: \forall R = read_T(i): v \in GlobalReads(S):$$

$$\exists T' \in Trans: \exists W = write_T(i, v) \in GlobalWrites(S):$$

$$T' \in Committed(S) \land T' \neq T \land T' \sqsubset R \land NoWriterBetween_{S,i}(T', \sqsubseteq, R)$$
It is straightforward to prove $\sqsubseteq \in \text{Marking}(S)$ and $\preceq_S \sqsubseteq \sqsubseteq, \sqsubseteq \in \text{ReadPres}(S)$. Additionally, the first conjunct of $\text{WriteObs}(S)$ (that is, $S \in \text{LocalTSeqSpec}$) is immediate from the assumption. So, we still need to prove the second conjunct of $\text{WriteObs}(S)$.

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

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

$\leftarrow$:

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

\[
\begin{align*}
\preceq_S &\subseteq \sqsubseteq \land \sqsubseteq \in \text{ReadPres}(S) \land \\
S &\in \text{LocalTSeqSpec} \land \\
\forall T \in \text{Trans}: \forall i \in I: \forall v \in V: \forall R = \text{read}_T(i) : v \in \text{GlobalReads}(S) : \\
\exists T' \in \text{Committed}(S) : \exists W = \text{write}_T(i,v) \in \text{GlobalWrites}(S) : \\
T' &\neq T \land T' \sqsubseteq R \land \text{NoWriterBetween}_{S,i}(T', \sqsubseteq, R)
\end{align*}
\]

We show that

\[
\begin{align*}
S &\in \text{LocalTSeqSpec} \land \\
\forall T \in S : \forall i \in I : \forall v \in V : \forall R = \text{read}_T(i) : v \in \text{GlobalReads}(S) : \\
\exists T' \in \text{Committed}(S) : \exists W = \text{write}_T(i,v) \in \text{GlobalWrites}(S) : \\
(T' &\not\preceq_S T) \land \text{NoWriterBetween}_{S,i}(T', \preceq_S T)
\end{align*}
\]

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

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

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

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

Proof.

\[
\text{FinalStateOpaque} = \{ H \in \text{THistory} | \exists H' \in \text{TExtension}(H): \exists S \in \text{TSequential}:
    \begin{align*}
        H' &\equiv S \land \preceq_{H'} \subseteq \preceq_S \land S \in \text{TSeqSpec} \\
    \end{align*}
\]

\[
= \{ H \in \text{THistory} | \exists H' \in \text{TExtension}(H): \exists S \in \text{TSequential}:
    \begin{align*}
        H' &\equiv S \land \preceq_{H'} \subseteq \preceq_S \land S \in \text{TSeqSpec} \\
        \exists \subseteq \in \text{Marking}(S): \quad \preceq_S \subseteq \subseteq \land \subseteq \in \text{ReadPres}(S) \cap \text{WriteObs}(S) \\
    \end{align*}
\]

\[
= \{ H \in \text{THistory} | \exists H' \in \text{TExtension}(H): \exists S \in \text{TSequential}:
    \begin{align*}
        H' &\equiv S \land \preceq_{H'} \subseteq \preceq_S \land \exists \subseteq \in \text{Marking}(H'): \quad \preceq_S \subseteq \subseteq \land \subseteq \in \text{ReadPres}(H') \cap \text{WriteObs}(H') \\
    \end{align*}
\]

\[
= \{ H \in \text{THistory} | \exists H' \in \text{TExtension}(H): \exists \subseteq \in \text{Marking}(H'):
    \begin{align*}
        \subseteq \in \text{ReadPres}(H') \cap \text{WriteObs}(H') \land \\
        \exists S \in \text{TSequential}: H' \equiv S \land \preceq_{H'} \subseteq \preceq_S \land \preceq_S \subseteq \subseteq \\
    \end{align*}
\]

\[
= \{ H \in \text{THistory} | \exists H' \in \text{TExtension}(H): \exists \subseteq \in \text{Marking}(H'):
    \begin{align*}
        \preceq_{H'} \subseteq \subseteq \land \subseteq \in \text{ReadPres}(H') \cap \text{WriteObs}(H') \\
    \end{align*}
\]

\[
= \text{Markable}
\]

In these eight steps we apply:
1) the definition of \( \text{FinalStateOpaque} \),
2) Lemma \ref{lemma17} and \( S \in \text{TComplete} \) (because \( H' \in \text{TExtension}(H) \) and \( H' \equiv S \)),
3) the definition of \( \text{FinalStateMarkable} \) and \( S \in \text{TComplete} \),
4) Lemma \ref{lemma14},
5) logical rearrangement,
6) transitivity of \( \subseteq \),
7) Lemma \ref{lemma15} and
8) the definition of \( \text{FinalStateMarkable} \).  \( \square \)
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, counter, set and 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.\footnote{In this subsection, we use $\forall$ and $\exists$ as a notational convenience. $\forall l : p$ can be rewritten as $\bigwedge_{(l \in \text{Labels}(X))} p(X)$ and $\exists l : p$ can be rewritten as $\bigvee_{(l \in \text{Labels}(X))} p(X)$.}

Basic and Linearizable Object Types

The abstract type of each object $o$ specifies the sequential specification of $o$, denoted by $\text{SeqSpec}(o)$, 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:

$$\mathbb{H}_B(o) \cap \text{Sequential} \subseteq \text{SeqSpec}(o)$$

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.

$$\mathbb{H}_L(o) = \{(X, L) \mid X \equiv L \land L \in \text{SeqSpec}(o) \land \prec_X \subseteq \prec_L\}$$

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 $\neg(l' \prec_L l) \land \neg(l = l')$.

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 $(l \prec_L l') \lor (l' \prec_L l) \lor (l = l')$.

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 $l_1 \prec_X l_4$.\footnote{In this subsection, we use $\forall$ and $\exists$ as a notational convenience. $\forall l : p$ can be rewritten as $\bigwedge_{(l \in \text{Labels}(X))} p(X)$ and $\exists l : p$ can be rewritten as $\bigvee_{(l \in \text{Labels}(X))} p(X)$.}
Register

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

**Definition 3** The sequential specification of register \( \text{reg} \) is the set of sequential histories of \textit{read} and \textit{write} method calls on \( \text{reg} \) 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 \( r \), \( \text{SeqSpec}(r) \), is defined as follows:

\[
\begin{align*}
\text{isXRead}_{X,r}(l_R) &= l_R \in X \land \text{obj}_X(l_R) = r \land \text{name}_X(l_R) = \text{read} \\
\text{isXWrite}_{X,r}(l_W) &= l_W \in X \land \text{obj}_X(l_W) = r \land \text{name}_X(l_W) = \text{write} \\
\text{NoWriteBetween}_{X,r}(l_W,l_R) &= \forall l'_W: \text{isXWrite}_{X,r}(l'_W) \Rightarrow (l'_W \preceq_X l_W \lor l_R <_X l'_W) \\
\text{isXWriter}_{X,r}(l_W,l_R) &= \text{isXWrite}_{X,r}(l_W) \land \text{l}_W <_X l_R \land \text{NoWriteBetween}_{X,r}(l_W,l_R) \\
\text{Legal}(r) &= \{ S \mid \forall l_R: \text{isXRead}_{S,r}(l_R) \Rightarrow \exists l_W: \text{isXWriter}_{S,r}(l_W,l_R) \land \text{retv}_X(l_R) = \text{arg}_1 X(l_W) \} \\
\text{SeqSpec}(r) &= \{ S \mid S|_r = S \land S \in \text{Sequential} \cap \text{Legal}(r) \}
\end{align*}
\]

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

Let \text{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,

\[
\forall \text{reg} \in \text{BasicRegister} : \forall X \in \mathbb{H}_B(\text{reg}) : X \in \text{Sequential} \Rightarrow
\forall l_R: \text{isXRead}_{X,\text{reg}}(l_R) \Rightarrow
\exists l_W: \text{isXWriter}_{X,\text{reg}}(l_W,l_R) \land
\text{retv}_X(l_R) = \text{arg}_1 X(l_W)
\]

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

\[
\begin{align*}
\text{isXRaceFree}_{X,r}(l) &= \forall l_w: \text{isXWrite}_{X,r}(l_w) \Rightarrow l_w \preceq_X l \lor l <_X l_w \\
\text{isXSequentiallyWritten}_{r}(X) &= \forall l \in X: \text{isXWrite}_{X,r}(l) \Rightarrow \text{isRaceFree}_{X,r}(l)
\end{align*}
\]

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.

\[
\mathbb{H}_B(r) = \{ X \mid X|_0 = X \land \\
isXSequentiallyWritten_r(X) \Rightarrow \\
\forall l_r: isXRead_{X,r}(l_r) \land isXRaceFree_{X,r}(l_r) \Rightarrow \\
\exists l_w: isXWriter_{X,r}(l_w, l_r) \land \\
retv_X(l_r) = arg1_X(l_w) \}
\] (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,

\[
\forall reg \in BasicRegister: \forall X \in \mathbb{H}_B(reg): isXSequentiallyWritten_r(X) \Rightarrow \\
\forall l_R: isXRead_{X,reg}(l_R) \land isXRaceFree_{X,r}(l_R) \Rightarrow \\
\exists l_W: isXWriter_{X,reg}(l_W, l_R) \land \\
retv_X(l_R) = arg1_X(l_W)
\] (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

\[
LNoWriteBetween_{X,L,r}(l_W, l_R) = \forall l_W: isXWrite_{X,r}(l_W) \Rightarrow (l_W \preceq_L l_W \lor l_R \prec_L l_W)
\] (5.14)

\[
isLWriter_{X,L,r}(l_W, l_R) = \forall l_W: isXWrite_{X,r}(l_W) \land \\
l_W \prec_L l_R \land \\
LNoWriteBetween_{X,L,r}(l_W, l_R)
\] (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,

\[
\forall r \in AtomicRegister: \forall (X, L) \in \mathbb{H}_L(r): \\
\forall l_R: isXRead_{X,r}(l_R) \Rightarrow \\
\exists l_W: isLWriter_{X,L,r}(l_W, l_R) \land \\
retv_X(l_R) = arg1_X(l_W)
\] (5.16)

**CAS (Compare-And-Swap) Register**

A CAS register is an object that encapuslates a value and supports the \( \text{cas} \) method in addition to \( \text{read} \) and \( \text{write} \) 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 \( \text{write} \) method call or a successful \( \text{cas} \) method call. The written value of a successful write is its first argument, if it is a \( \text{write} \) method call or is its second argument, if it is a \( \text{cas} \) method call.
Definition 5  The sequential specification of cas register \( \text{reg} \) is the set of sequential histories of read, write and cas method calls on \( \text{reg} \) 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 \( \text{AtomicCASRegister} \) denote the type of Atomic CAS registers.

Let us define

\[
\begin{align*}
\text{isXCAS}_{X,r}(l_w) &= l_w \in X \land \text{obj}_X(l_w) = r \land \text{name}_X(l_w) = \text{cas} \quad (5.17) \\
\text{isXCWrite}_{X,r}(l_w) &= \text{isXWrite}(l_w) \lor (\text{isXCAS}(l_w) \land \text{retv}_X(l_w) = \text{true}) \quad (5.18) \\
\text{writtenValue}_X(l_w) &= \begin{cases} 
\text{arg}_1X(l_w) & \text{if name}_X(l_w) = \text{write} \\
\text{arg}_2X(l_w) & \text{if name}_X(l_w) = \text{cas}
\end{cases} \quad (5.19) \\
\text{LNoWriteBetween}_{X,L,r}(l_w,l_R) &= \forall l'_w: \text{isXCWrite}_{X,r}(l'_w) \Rightarrow (l'_w \preceq_L l_w \lor l_R \prec_L l'_w) \quad (5.20) \\
\text{isLCWriter}_{X,L,r}(l_w,l_R) &= \text{isXCWrite}_{X,r}(l_w) \land \\
& l_w \prec_L l_R \land \\
& \text{LNoWriteBetween}_{X,L,r}(l_w,l_R)
\end{align*}
\]

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,

\[
\begin{align*}
\forall r \in \text{AtomicCASRegister}: \forall (X,L) \in \mathbb{H}_L(r): \\
\forall l_R: \text{isXRead}_{X,r}(l_R) \Rightarrow \\
\exists l_w: \text{isLCWriter}_{X,L,r}(l_w,l_R) \land \\
\text{retv}_X(l_R) = \text{arg}_1X(l_w) 
\end{align*}
\]

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,

\[
\begin{align*}
\forall \text{reg} \in \text{AtomicCASRegister}: \forall (X,\text{Reg}) \in \mathbb{H}_L(\text{reg}): \\
\forall l_C,l_w: \\
\text{isXCAS}_{X,\text{reg}}(l_C) \land \\
\text{isLCWriter}_{X,\text{Reg},\text{reg}}(l_w,l_R) \Rightarrow \\
(\text{writtenValue}_X(l_w) = \text{arg}_1X(l_C) \Rightarrow \text{retv}_X(l_C) = \text{true}) \land \\
(\neg(\text{writtenValue}_X(l_w) = \text{arg}_1X(l_C)) \Rightarrow \text{retv}_X(l_C) = \text{false})
\end{align*}
\]

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: \( \text{lock} \): The method call \( l.\text{lock}() \) changes the state from \( \mathbb{R} \) to \( \mathbb{A} \). \( \text{unlock} \): The method call \( l.\text{unlock}() \) changes the state from \( \mathbb{A} \) to \( \mathbb{R} \). \text{read} \: The method call \( l.\text{read}() \) returns \text{true} if the state of \( \text{lock} \) is \( \mathbb{A} \) and \( \text{false} \) otherwise. The method calls \( \text{lock} \) and \( \text{unlock} \) are mutating method calls. The method call \( \text{read} \) 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 \( \text{Lock} \) denote the type of locks.

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

\[
isXLock_{X,lo}(l) = \begin{cases} 
    l \in X & \text{and } \text{obj}_X(l) = lo \land \text{name}_X(l) = \text{lock} \\
\end{cases} 
\]

\[
isXUnlock_{X,lo}(l) = \begin{cases} 
    l \in X & \text{and } \text{obj}_X(l) = lo \land \text{name}_X(l) = \text{unlock} \\
\end{cases} 
\]

\[
isXRead_{X,lo}(l) = \begin{cases} 
    l \in X & \text{and } \text{obj}_X(l) = lo \land \text{name}_X(l) = \text{read} \\
\end{cases} 
\]

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.

\[
isXOwnerRespecting_{lo}(X) = \\
\forall l: isXUnlock_{X,lo}(l) \Rightarrow \\
\exists l': isXLock_{X,lo}(l') \land \\
thread_X(l') = thread_X(l) \land \\
l' <_X l \land \\
\forall l'': (isXUnlock_{X,lo}(l'') \land thread_X(l'') = thread_X(l)) \Rightarrow (l'' <_X l' \lor l \preceq_X l'') 
\]

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,

\[
\forall o \in \text{Lock}: \forall (X, L) \in \text{H}_L(o): \forall l_1, l_2: \\
(isXOwnerRespecting_{lo}(X) \land \\
isXLock_{X,o}(l_1) \land \\
isXUnlock_{X,o}(l_2) \land \\
l_1 <_L l_2) \Rightarrow \\
\exists l_1, l_2: \\
isXUnlock_{X,o}(l_1) \land thread_X(l_1) = thread_X(l_1) \land \\
isXLock_{X,o}(l_2) \land thread_X(l_2) = thread_X(l_2) \land \\
l_1 <_L l_2
\]
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,

\[
\forall o \in \text{Lock}: \forall (X, L) \in \mathbb{H}_L(o): \forall l_{u1}, l_{r2}:
\begin{align*}
& (\text{isXOwnerRespecting}_o(X) \land \\
& \text{isXRead}_{X,o}(l_{r2}) \land \text{retv}_{X}(l_{r2}) = \text{false} \land \\
& \text{isXUnlock}_{X,o}(l_{u1}) \land \\
& l_{r2} \prec L l_{u1}) \Rightarrow \\
\exists l_{l1}: \\
& \text{isXLock}_{X,o}(l_{l1}) \land \text{thread}_{X}(l_{l1}) = \text{thread}_{X}(l_{u1}) \land \\
& l_{r2} \prec L l_{l1}
\end{align*}
\]

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,

\[
\forall o \in \text{Lock}: \forall (X, L) \in \mathbb{H}_L(o): \forall l_{l1}, l_{r2}:
\begin{align*}
& (\text{isXOwnerRespecting}_o(X) \land \\
& \text{isXLock}_{X,o}(l_{l1}) \land \\
& \text{isXRead}_{X,o}(l_{r2}) \land \text{retv}_{X}(l_{r2}) = \text{false} \land \\
& l_{l1} \prec L l_{r2}) \Rightarrow \\
\exists l_{u1}: \\
& \text{isXUnlock}_{X,o}(l_{u1}) \land \text{thread}_{X}(l_{l1}) = \text{thread}_{X}(l_{u1}) \land \\
& l_{u1} \prec L l_{r2}
\end{align*}
\]

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,

\[
\forall o \in \text{Lock}: \forall (X, L) \in \mathbb{H}_L(o): \forall l_{l1}, l_{r2}, l_{u1}, l_{r2}:
\begin{align*}
& (\text{isXOwnerRespecting}_o(X) \land \\
& \text{isXLock}_{X,o}(l_{l1}) \land \\
& \text{isXUnlock}_{X,o}(l_{u1}) \land \\
& \text{thread}_{X}(l_{l1}) = \text{thread}_{X}(l_{u1}) \land \\
& \forall l_{u1}': (\text{isXUnlock}_{X,o}(l_{u1}') \land \text{thread}_{X}(l_{l1}) = \text{thread}_{X}(l_{u1}')) \Rightarrow (l_{u1}' \prec X l_{l1} \lor l_{u1}' \preceq X l_{l1}') \land \\
& \text{isXRead}_{X,o}(l_{r2}) \land \\
& l_{l1} \prec L l_{r2} \land l_{r2} \prec L l_{u1}) \Rightarrow \\
\text{retv}_{X}(l_{r2}) = \text{true}
\end{align*}
\]

Try-Lock

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

\[
isXTryLock_{X,o}(l) = (5.32) \\
\text{isXTLock}_{X,o}(l) = (5.33) \\
isXLock_{X,o}(l) \lor (isXTryLock_{X,o}(l) \land \text{retv}_{X}(l) = \text{true})
\]

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.

\[
isXTOwnerRespecting_{o}(X) = (5.34) \\
\forall l: \text{isXUnlock}_{X,o}(l) \Rightarrow \exists l': \text{isXTLock}_{X,o}(l') \land \text{thread}_{X}(l') = \text{thread}_{X}(l) \land l' \prec_{X} l \land \forall l'': \text{isXUnlock}_{X,o}(l'') \land \text{thread}_{X}(l'') = \text{thread}_{X}(l) \Rightarrow l'' \prec_{X} l' \lor l \preceq_{X} l''
\]

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,

$$\forall o \in TryLock: \forall (X, L) \in \mathbb{H}_L(o): \forall l_{u1}, l_{u2}: \tag{5.35}$$

\[
\begin{align*}
& (isXTOwnerRespecting_o(X) \land \\
& isXTLock_{X,o}(l_{u1}) \land \\
& isXUnlock_{X,o}(l_{u2}) \land \\
& l_{u1} \prec_L l_{u2}) \Rightarrow \\
\exists l_{r1}, l_{r2} : \quad \\
& isXUnlock_{X,o}(l_{r1}) \land thread_X(l_{r1}) = thread_X(l_{u1}) \land \\
& isXTLock_{X,o}(l_{r2}) \land thread_X(l_{r2}) = thread_X(l_{u2}) \land \\
& l_{r1} \prec_L l_{r2}
\end{align*}
\]

**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,

$$\forall o \in TryLock: \forall (X, L) \in \mathbb{H}_L(o): \forall l_{u1}, l_{r2}: \tag{5.36}$$

\[
\begin{align*}
& (isXTOwnerRespecting_o(X) \land \\
& isXRead_{X,o}(l_{r2}) \land retv_X(l_{r2}) = false \\
& isXUnlock_{X,o}(l_{u1}) \land \\
& l_{r2} \prec_L l_{u1}) \Rightarrow \\
\exists l_{r1} : \\
& isXTLock_{X,o}(l_{r1}) \land thread_X(l_{r1}) = thread_X(l_{u1}) \land \\
& l_{r1} \prec_L l_{r2}
\end{align*}
\]

**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,

$$\forall o \in TryLock: \forall (X, L) \in \mathbb{H}_L(o): \forall l_{r1}, l_{r2}: \tag{5.37}$$

\[
\begin{align*}
& (isXTOwnerRespecting_o(X) \land \\
& isXTLock_{X,o}(l_{r1}) \land \\
& isXRead_{X,o}(l_{r2}) \land retv_X(l_{r2}) = false \\
& l_{r1} \prec_L l_{r2}) \Rightarrow \\
\exists l_{r1} : \\
& isXUnlock_{X,o}(l_{r1}) \land thread_X(l_{r1}) = thread_X(l_{u1}) \land \\
& l_{r1} \prec_L l_{r2}
\end{align*}
\]

**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,

$$
\forall o \in \text{TryLock}: \forall (X, L) \in H_L(o): \forall l_1, l_u, l_2:
$$

$$
(isXOwnerRespecting_o(X) \land
isXTLock_{X,o}(l_1) \land
isXUnlock_{X,o}(l_u) \land
thread_X(l_1) = thread_X(l_u) \land
\forall l_1: (isXUnlock_{X,o}(l_u) \land thread_X(l_1) = thread_X(l'_u)) \implies (l'_u <_X l_1 \lor l_u \preceq X l'_u)
$$

$$
\implies retv_X(l_2) = true
$$

Seq-Lock

Abstract seq-lock. A seq-lock $l$ is an object that encapsulates a number and an abstract state, acquired $A$ or released $R$. 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 $A$ 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 $R$ to $A$ 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 $A$ to $R$.

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 \( S\text{Counter} \) 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,

\[
\forall c \in S\text{Counter}: \forall (X, C) \in \mathbb{H}_{L}(c): \forall l, l':
\]

\[
l \in X \land l' \in X \land name_X(l') = \text{iaf} \land l \prec_C l'
\]

\[
\Rightarrow
\]

\[
\text{retv}_X(l) < \text{retv}_X(l')
\]

**Set**

A set \( s \) is an object that represents a set of values and supports the following methods: \( \text{add} \): The method call \( s.\text{add}(v) \) adds value \( v \) to set \( s \). \( \text{contains} \): The method call \( s.\text{contains}(v) \) 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 \( \text{add} \) and \( \text{contains} \) method calls on \( s \) where every \( \text{contains} \) method call returns \( true \) if there is a preceding \( \text{add} \) 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 \( B\text{asicSet} \) denote the type of basic sets.

Let us define

\[
isXContains_{X,s}(l) =
\]

\[
l \in X \land \text{obj}_X(l) = s \land name_X(l) = \text{contains}
\]

\[
isXAdd_{X,s}(l) =
\]

\[
l \in X \land \text{obj}_X(l) = s \land name_X(l) = \text{add}
\]

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

\[
\forall s \in B\text{asicSet}: \forall X \in \mathbb{H}_B(s): X \in \text{Sequential} \Rightarrow
\]

\[
\forall l_c: isXContains_{X,s}(l_c) \land \text{retv}_X(l_c) = true \Rightarrow
\]

\[
\exists l_a: isXAdd_{X,s}(l_a) \land arg1(l_a) = arg1(l_c) \land l_a \prec_X l_c
\]

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

\[
\forall s \in B\text{asicSet}: \forall X \in \mathbb{H}_B(s): X \in \text{Sequential} \Rightarrow
\]

\[
\forall l_c, l_a:
\]

\[
isXContains_{X,s}(l_c) \land
\]

\[
isXAdd_{X,s}(l_a) \land
\]

\[
arg1(l_a) = arg1(l_c) \land l_a \prec_X l_c
\]

\[
\Rightarrow
\]

\[
\text{retv}_X(l_c) = true
\]
**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 $v$ ($v \neq \perp$) 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 $\perp$ 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 $\perp$ 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 $\text{BasicMap}$ denote the type of basic maps.

Let us define

$$
\text{isXGet}_{X,m}(l) = \begin{cases} 
1 \in X & \text{obj}_X(l) = m \land \text{name}_X(l) = \text{get} \\
\end{cases}
$$

$$
\text{isXPut}_{X,m}(l) = \begin{cases} 
l \in X & \text{obj}_X(l) = m \land \text{name}_X(l) = \text{put} \\
\end{cases}
$$

$$
\text{isXPutter}_{X,m}(l_p, l_g) \iff \begin{cases} 
\text{isXPut}_{X,m}(l_p) \land \text{arg}_1 X(l_p) = \text{arg}_1 X(l_g) \land l_p \prec X \ l_g \\
\forall l'_p: \text{isXPut}_{X,m}(l'_p) \land \text{arg}_1 X(l'_p) = \text{arg}_1 X(l_g) \Rightarrow (l'_p \preceq X l_p \lor l_g \prec X l'_p)
\end{cases}
$$

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

$$
\forall m \in \text{BasicMap}: \forall X \in H_B(m): X \in \text{Sequential} \Rightarrow
\forall l_g: \text{isXGet}_{X,m}(l_g) \land \lnot (\text{retv}_X(l_g) = \perp) \Rightarrow
\exists l_p: \text{isPutter}_{X,m}(l_p, l_g) \land
\text{arg}_2 X(l_p) = \text{retv}_X(l_g)
$$

**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,

$$
\forall m \in \text{BasicMap}: \forall X \in H_B(m): X \in \text{Sequential} \Rightarrow
\forall l_g, l_p:
\text{isXGet}_{X,m}(l_g) \land
\text{isPutter}_{X,m}(l_p, l_g) \land
\Rightarrow
\text{retv}_X(l_g) = \text{arg}_2 X(l_p)
$$

**Proof Sketches.**

Lemma [19]
Straightforward from $\prec_X \subseteq L$.

Lemma 20:
We have

1. $l \prec_L l'$
From 1, we have

2. $rEv(l) \prec_L iEv(l')$
From the well-formedness of the history $O$, we have

3. $iEv(l) \prec_L rEv(l)$
4. $iEv(l') \prec_L rEv(l')$
From 3, 2 and 4, we have

5. $iEv(l) \prec_L rEv(l')$
From 5, we have

6. $\neg(rEv(l') \prec_L iEv(l))$
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

$l \prec_L l' \lor l' \prec_L l \lor l = l'$

Lemma 23:
Straightforward from the fact that $L$ is equivalent to $X$.
We have

1. $X \equiv L$
From (3), we have
(4) $l \in L$
(5) $l' \in L$
From (2) on (4) and (5), we have
(6) $\text{obj}_L(l) = o$
(7) $\text{obj}_L(l') = o$
From (1) on (4) and (5), we have
(l) $l \in X$
(l') $l' \in X$
From (1) on (6) and (7), we have
$\text{obj}_X(l) = o$
$\text{obj}_X(l') = o$

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 [1]. Derivable from the semantics of linearizable objects (Definition 2) and the sequential specification of register (Definition 3).

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

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

Lemma 30:
Derived 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 fromLemma 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

<table>
<thead>
<tr>
<th>Function</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>def <code>init_t()</code></td>
<td><code>\texttt{rver} = \texttt{clock.iaf}()</code>, <code>\texttt{C07 \triangleright wver = clock.iaf()}</code></td>
</tr>
<tr>
<td><code>\texttt{R01 \triangleright snap = clock.read()}</code>, <code>\texttt{R02 \triangleright rver[t].write(snap)}</code>, <code>\texttt{R03 \triangleright return ok}</code></td>
<td><code>\texttt{C01 \triangleright foreach (i \in \texttt{wset}[t])}</code></td>
</tr>
<tr>
<td><code>\texttt{R01 \triangleright pv = wset[t].get(i)}</code>, <code>\texttt{R02 \triangleright return pv}</code></td>
<td><code>\texttt{C02_i \triangleright locked = lock[i].trylock()}</code>, <code>\texttt{C03_i \triangleright if (~locked) lset.add(i) else}</code></td>
</tr>
<tr>
<td><code>\texttt{R03 \triangleright s_1 = ver[i].read()}</code>, <code>\texttt{R04 \triangleright v = reg[i].read()</code></td>
<td><code>\texttt{C04_i \triangleright foreach (j \in \texttt{lset})}</code></td>
</tr>
<tr>
<td><code>\texttt{R05 \triangleright l = lock[i].read()}</code>, <code>\texttt{R06 \triangleright s_2 = ver[i].read()}</code>, <code>\texttt{R07 \triangleright sver = rver[t].read()</code>, <code>\texttt{R08 \triangleright return A_i}</code></td>
<td><code>\texttt{C05_{ij} \triangleright lock[j].unlock()}</code>, <code>\texttt{C06_i \triangleright return A_i};</code></td>
</tr>
<tr>
<td><code>\texttt{R09 \triangleright rver[t].add(i) \{R03 \rightarrow R04, R04 \rightarrow R05, R05 \rightarrow R06\}},</code></td>
<td><code>\texttt{C07 \triangleright wver = clock.iaf()}</code></td>
</tr>
<tr>
<td><code>\texttt{W01 \triangleright wset[t].put(i, v)}</code>, <code>\texttt{W02 \triangleright return ok}</code></td>
<td><code>\texttt{C08 \triangleright sver = rver[t].read()</code>, <code>\texttt{C09 \triangleright foreach (i \in \texttt{lset})}</code></td>
</tr>
<tr>
<td><code>\texttt{def \textit{abort}_t()} \{C01 \rightarrow C07, C10 \rightarrow C11, C09 \rightarrow C15},</code></td>
<td><code>\texttt{C10_i \triangleright l = lock[i].read()}</code>, <code>\texttt{C11_i \triangleright s = ver[i].read()</code></td>
</tr>
<tr>
<td><code>\texttt{A01 \triangleright return A_i} \{C01 \rightarrow C07, C10 \rightarrow C11, C09 \rightarrow C15},</code></td>
<td><code>\texttt{C12_i \triangleright foreach (j \in \texttt{lset})}</code></td>
</tr>
<tr>
<td><code>\texttt{C13_{ij} \triangleright lock[j].unlock()}</code>, <code>\texttt{C14_i \triangleright return A_i},</code></td>
<td><code>\texttt{C15 \triangleright foreach ((i, v) \in \texttt{wset}[t])}</code></td>
</tr>
<tr>
<td><code>\texttt{C16_i \triangleright reg[i].write(v)},</code></td>
<td><code>\texttt{C17_i \triangleright ver[i].write(wver)},</code></td>
</tr>
<tr>
<td><code>\texttt{C18_i \triangleright lock[i].unlock()},</code></td>
<td><code>\texttt{C19 \triangleright return C},</code></td>
</tr>
<tr>
<td><code>\texttt{C19 \triangleright return C},</code></td>
<td><code>\texttt{C16 \rightarrow C17, C17 \rightarrow C18},</code></td>
</tr>
</tbody>
</table>

\textbf{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 \textit{reg}: an array of basic registers. Version registers \textit{ver}: an array of try-locks that are initially released. The arrays are of size memory location count \(|I|\). Global version clock \textit{clock}: a strong counter with the initial value 0. Locks \textit{lock}: an array of try-locks that are initially released. The \(\emptyset\) thread-local basic set that is initially \(\emptyset\). A strong counter provides two methods in its interface: \textit{iaf} (inc-and-fetch) that increments the counter and returns the counter value and \textit{read} that returns the counter value. Read version \textit{rver}: a thread-local basic register. Read set \textit{rset}: a thread-local basic set that is initially \(\emptyset\). Write set \textit{wset}: a thread-local basic map that is initially \(\emptyset\). Lock set \textit{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 \textit{ok}, \(A\), \(C\) 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 \textit{wset}[t] at \(W01\) and is written back to register \textit{reg}[i] at \(C16_i\) while \(t\) is committing. TL2 records a version in the register \textit{ver}[i] for the value stored in the register \textit{reg}[i]. The version register \textit{ver}[i] is updated to ascending numbers at \(C17_i\) after new values are written back to \textit{reg}[i] at \(C16_i\). The try-lock \textit{lock}[i] is used for exclusive access to the registers for location \(i\). At commit, the lock \textit{lock}[i] of each location \(i\) in the write set \textit{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 \textit{clock} at \(C07\). Then, for each location in the read set \textit{rset}[t], first \textit{lock}[i] and then \textit{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 \textit{wset}[t] is written back at \(C15_i\) to \(C18_i\). For each pair in the write set \textit{wset}[t], the following three operations are executed in order. First, the buffered value is written back to \textit{reg}[i], then \textit{ver}[i] is updated, and then \textit{lock}[i] is released. In the \textit{init} method, each transaction \(t\) reads the current snapshot version from \textit{clock} at \(I01\) and writes it to the read version register \textit{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 \textit{ver}[i], \textit{reg}[i], \textit{lock}[i] and again \textit{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 \textit{rset}[t] and the read value is returned.

---

2As 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 $l_1 \preceq_H l_2$ to denote that $l_1$ is executed before or concurrently to $l_2$. We use $\prec_{\text{clock}}, \prec_{\text{ver}[i]}$ and $\prec_{\text{lock}[i]}$ to denote the linearization order of $\text{clock}, \text{ver}[i]$ and $\text{lock}[i]$ 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 $\text{initOf}_H(T)$ and $\text{commitOf}_H(T)$ to denote the $\text{init}$ and $\text{commit}$ 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 $\text{clock}$. 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 $\text{reg}[i]$ register. The read method reads $\text{reg}[i]$ at $R04$ and a writer transaction writes to $\text{reg}[i]$ at $C16_i$.

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

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

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

\[
\begin{align*}
\{(T,T') | T,T' \in \text{Trans}(H) \land \text{Eff}(T) \prec_{\text{clock}} \text{Eff}(T')\} \cup \\
\{(T,R) | \exists i: R \in \text{GlobalTReads}(H), i = \text{arg}_1(R), T \in \text{Writers}_H(i) \land \text{writeAcc}(T,i) \prec_H \text{readAcc}(R)\} \cup \\
\{(R,T) | \exists i: R \in \text{GlobalTReads}(H), i = \text{arg}_1(R), T \in \text{Writers}_H(i) \land \text{readAcc}(R) \preceq_H \text{writeAcc}(T,i)\}
\end{align*}
\]

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 $l_2$. 

39
Lemma 45 TL2 preserves reads of aborted transactions (part 1).

∀H ∈ H(TL2):
∀R ∈ GlobalTReads(H): Let i = arg1_H(R), T = trans_H(R):
T ∈ Aborted(H) ⇒ NoWriterBetween_H,i(R, ⊆, T)

Proof Sketch.

<table>
<thead>
<tr>
<th>T</th>
<th>T’</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>C02_i ⊲ lock[i].trylock()</td>
</tr>
<tr>
<td></td>
<td>C07 ⊲ wver = clock.iaf()</td>
</tr>
<tr>
<td>I01 ⊲</td>
<td>snap = clock.read()</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td>R03 ⊲</td>
<td>s1 = ver[i].read()</td>
</tr>
<tr>
<td>R04 ⊲</td>
<td>v = reg[i].read()</td>
</tr>
<tr>
<td></td>
<td>C16_i ⊲ reg[i].write(v)</td>
</tr>
<tr>
<td></td>
<td>C17_i ⊲ ver[i].write(wver)</td>
</tr>
<tr>
<td>R05 ⊲</td>
<td>l = lock[i].read()</td>
</tr>
<tr>
<td>R06 ⊲</td>
<td>s2 = ver[i].read()</td>
</tr>
<tr>
<td>R07 ⊲</td>
<td>sver = rver[t].read()</td>
</tr>
<tr>
<td>if (∼(¬l ∧ s1 = s2 ∧ s2 ≤ sver))</td>
<td>return A</td>
</tr>
</tbody>
</table>

Figure 5: Case T ∈ Aborted(H) ∧ R ⊆ T’ ⊆ 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’ ⊆ R

and
T’ takes effect before T
that is
[2] T’ ⊆ T

We show that
TL2 aborts R.

Figure 5 depicts the two transactions.

By Definition 13 on [1], we have
[3] R04 ≺_H C16_i

By Definition 13 on [2], we have
[4] C07 ≺_{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 A in both cases.

Case 1:
[5] R05 ≺_{lock[i]} C18_i
By P2X on the algorithm, we have
(6) \( C_{02} \prec_H C_{07} \)
(7) \( I_{01} \prec_H R_{05} \)
By the Lemma XLTrans on [6], [4] and [7], we have
\( C_{02} \prec_H R_{05} \)
thus, by the Lemma X2L, we have
(8) \( C_{02} \prec_{lock[i]} R_{05} \)
By the Lemma TryLockReadM on [8] and [5], we have that
\( R_{05} \) returns true i.e. \( l = true \)
Thus,
The validation check fails and \( R \) returns \( A \).

Case 2:
(9) \( C_{18} \prec_{lock[i]} R_{05} \)
By P2X on the algorithm, we have
(10) \( C_{17} \prec_H C_{18} \)
(11) \( R_{05} \prec_H R_{06} \)
By the Lemma XLTrans on [10], [9] and [11], we have
\( C_{17} \prec_H R_{06} \)
Thus, by the Lemma X2L, we have
(12) \( C_{17} \prec_{ver[i]} R_{06} \)
By Lemma 54 on [12], we have
(13) \( w_{ver} \leq s_2 \)
By P2X on the algorithm, we have
(14) \( R_{03} \prec_H R_{04} \)
(15) \( C_{16} \prec_H C_{17} \)
By the Lemma XXTrans on [14], [3] and [15], we have
\( R_{03} \prec_H C_{17} \)
Thus, by the Lemma X2L, we have
(16) \( R_{03} \prec_{ver[i]} C_{17} \)
By Lemma 54 on [16], we have
(17) \( s_1 < w_{ver} \)
From [13] and [17], we have
\( \neg(s_1 = s_2) \)
Thus,
The validation check fails and \( R \) returns \( A \) in this case too.

\[ \]

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

\( \forall H \in \mathbb{H}(TL2) : \)
\( \forall R \in \text{GlobalTReads}(H) : \) Let \( i = \text{arg}_1(H)(R), T = \text{trans}_H(R) : \)
\( T \in \text{Aborted}(H) \Rightarrow \text{NoWriterBetween}_H,i(T, \sqsubseteq, R) \)

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

\[ \]
\begin{center}
\begin{tabular}{|c|l|}
\hline
$T$ & $T'$
\hline
$I01 \triangleright \quad$ snap = \texttt{clock.read()} & $C02_i \triangleright \quad$ lock[i].trylock()
\hline
$I02 \triangleright \quad$ rver[t].write(snap) &
\hline
& $C07 \triangleright \quad$ wver = \texttt{clock.iaf}()
\hline
& $C16_i \triangleright \quad$ reg[i].write(v)
\hline
$R04 \triangleright \quad$ v = reg[i].read() & $C17_i \triangleright \quad$ ver[i].write(wver)
\hline
$R05 \triangleright \quad$ l = lock[i].read() & $C18_i \triangleright \quad$ lock[i].unlock()
\hline
$R06 \triangleright \quad$ s2 = ver[i].read() &
\hline
$R07 \triangleright \quad$ sver = rver[t].read() &
\hline
\multicolumn{2}{|c|}{\textbf{if} \: (\neg (\neg l \land s_1 = s_2 \land s_2 \leq \textit{sver}))} \\
\multicolumn{2}{|c|}{\quad \textbf{return} \: \texttt{A}}
\hline
\end{tabular}
\end{center}

Figure 6: Case $T \in \textit{Aborted}(H) \land T \not\subset T' \subset R$

$T'$ takes effect after $T$

that is

1. $T \subset T'$

and

$T'$ accesses $i$ before $R$

that is

2. $T' \subset R$

We show that

TL2 aborts $R$.

Figure 6 depicts the two transactions.

By Definition 13 on [1], we have

3. $I01 \prec_{\text{clock}} C07$

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 $\texttt{A}$ in both cases.

Case 1:

5. $R05 \prec_{\text{lock[i]}} C18_i$

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. $C02_i \prec_{\text{lock[i]}} R05$

By the Lemma TRYLOCKREADM on 8 and 5, we have that

$R05$ returns true i.e. $l = \text{true}$.

Thus,

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

42
Case 2:

<table>
<thead>
<tr>
<th>9</th>
<th>C18_i ≺_lock[i] R05</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>C17_i ≼_H C18_i</td>
</tr>
<tr>
<td>11</td>
<td>R05 ≼_H R06</td>
</tr>
</tbody>
</table>

By the Lemma XLTRANS on 10, 9 and 11, we have

C17_i ≼_H R06

Thus, by the Lemma X2L, we have

| 12 | C17_i ≼_ver[i] R06 |

By Lemma 53 on 12, we have

| 13 | wver ≤ s2 |

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 > s2

Thus,

The validation check fails and R returns A in this case too.

\[ \square \]

**Lemma 47** TL2 preserves reads of aborted transactions.

\[ \forall H \in \Pi(TL2): \forall R \in \text{GlobalTReads}(H): \text{Let } i = \text{arg1}_H(R), T = \text{trans}_H(R): \]

\[ T \in \text{Aborted}(H) \Rightarrow \]

\[ \text{NoWriterBetween}_{H,i}(R, \subseteq, T) \land \text{NoWriterBetween}_{H,i}(T, \subseteq, R) \]

**Proof.** Immediate from Lemma 45 and Lemma 46. \[ \square \]

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

\[ \forall H \in \Pi(TL2): \forall R \in \text{GlobalTReads}(H): \text{Let } i = \text{arg1}_H(R), T = \text{trans}_H(R): \]

\[ T \in \text{Committed}(H) \Rightarrow \]

\[ \text{NoWriterBetween}_{H,i}(R, \subseteq, T) \]

**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

| 1 | R ⊆ T' |

that is

T' accesses i after R

and

T' takes effect before T

that is
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) $R_{04} \prec_{H} C_{16i}$

By Definition 13 on [2], we have

(4) $C_{07}' \prec_{\text{clock}} C_{07}$

The method calls $I_{01}$ and $C_{07}'$ are on the object $\text{clock}$. We consider two cases for the linearization order of them.

Case 1:

(5) $C_{07}' \prec_{\text{clock}} I_{01}$

From [3] and [3],

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

Case 2:

(6) $I_{01} \prec_{\text{clock}} C_{07}'$

By the Lemma SCounter on [4], we have

(7) $wver' < wver$

By the Lemma SCounter on [6], we have

(8) $\text{snap} < wver'$

The value of $\text{swer}$ is read at $R_{07}$ from $\text{rver}$.

The thread-local register $\text{rver}$ is only assigned at $I_{02}$ to $\text{snap}$.

Thus, we have
From [8] and [9], we have
\[ \text{snap} = \text{svver} \]
From [10] and [7], we have
\[ \text{svver} < \text{wver}' \]
Thus,
\[ \text{wver} \neq \text{svver} + 1 \]

The if branch is taken.
The method calls \( C_{10i} \) and \( C_{18i}' \) are on the object \( \text{lock}[i] \).
We consider two cases for the linearization order of them.

**Case 2.1:**
\[ C_{10i} \prec_{\text{lock}[i]} C_{18i}' \]
By P2X on the algorithm, we have
\[ C_{02i}' \prec_H C_{07}' \]
\[ C_{07} \prec_H C_{10i} \]
By the Lemma XLTRANS on [13], [4] and [14], we have
\[ C_{02i}' \prec_H C_{10i} \]
thus, by the Lemma X2L, we have
\[ C_{02i}' \prec_{\text{lock}[i]} C_{10i} \]
By the Lemma TRYLOCKREADM on [15] and [12], we have that
\( R_{05} \) returns \text{true} i.e. \( l = \text{true} \)
Thus,
\[ \text{The validation check fails and } R \text{ returns } A. \]

**Case 2.2:**
\[ C_{18i}' \prec_{\text{lock}[i]} C_{10i} \]
By P2X on the algorithm, we have
\[ C_{17i}' \prec_H C_{18i}' \]
\[ C_{10i} \prec_H C_{11i} \]
By the Lemma XLTRANS on [17], [16] and [18], we have
\[ C_{17i}' \prec_H C_{11i} \]
Thus, by the Lemma X2L, we have
\[ C_{17i}' \prec_{\text{wver}[i]} C_{11i} \]
By Lemma 54 on [19], we have
\[ \text{wver}' \leq s \]
From [10], [20], we have
\[ \text{svver} < s \]
Thus,
\[ \text{The validation check fails and } R \text{ returns } A \text{ in this case too.} \]

---

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

\[
\forall H \in \mathbb{H}(TL2):
\forall R \in \text{Global}T\text{Reads}(H): \text{Let } i = \text{arg}_1 H(R), T = \text{trans}_H(R):
\quad T \in \text{Committed}(H) \Rightarrow
\quad \text{NoWriterBetween}_{H,i}(T, \subseteq, R)
\]
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

1. $T' \sqsupseteq T'$

We show that

2. $R \sqsupseteq T'$

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. $C07 \prec_{\text{clock}} C07'$

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.

$\forall H \in \mathbb{H}(TL2) :$

$\forall R \in \text{GlobalTReads}(H) :$ Let $i = \text{arg}1_H(R), T = \text{trans}_H(R) :$

$T \in \text{Committed}(H) \Rightarrow$

$\text{NoWriterBetween}_{H,i}(R, \subseteq, T) \land \text{NoWriterBetween}_{H,i}(T, \subseteq, R)$

Proof. Immediate from Lemma 48 and Lemma 49.

Lemma 51 TL2 is read-preserving.

$\forall H \in \mathbb{H}(TL2) :$ ReadPres$(H, \subseteq)$

Proof. Immediate from Lemma 47 and Lemma 50.

46
Lemma 52  Version registers are updated to ascending numbers.

Let $C_{17_i}^1$ denote the method call at line $C_{17_i}$ executed by a transaction $T_1$ and let $wver^1$ denote its argument. Similarly, let $C_{17_i}^2$ denote the method call at line $C_{17_i}$ executed by a transaction $T_2$ and let $wver^2$ denote its argument. If $C_{17_i}^1 \prec_{ver} C_{17_i}^2$, then $wver^1 < wver^2$.

Proof Sketch.

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>$C_{02_i}^1 \triangleright locked^1 = lock[i].trylock()$</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>$C_{07_i}^1 \triangleright wver^1 = clock.iaf()$</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>$C_{17_i}^1 \triangleright ver[i].write(wver^1)$</td>
<td>$C_{17_i}^2 \triangleright wver^2 = clock.iaf()$</td>
</tr>
<tr>
<td>$C_{18_i}^1 \triangleright lock[i].unlock()$</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>$C_{02_i}^2 \triangleright locked^2 = lock[i].trylock()$</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>$C_{07_i}^2 \triangleright wver^2 = clock.iaf()$</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>$C_{17_i}^2 \triangleright ver[i].write(wver^2)$</td>
<td></td>
</tr>
<tr>
<td>$C_{18_i}^2 \triangleright lock[i].unlock()$</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

We have that

1. $C_{17_i}^1 \prec_{ver} C_{17_i}^2$

We show that

$wver^1 < wver^2$

By P2X on the algorithm, we have

2. $C_{02_i}^1 \prec_H C_{17_i}^1$
3. $C_{17_i}^2 \prec_H C_{18_i}^2$

By the Lemma XLTrans on 2, 1 and 3, we have
4. $C_{02_i}^1 \prec_H C_{18_i}^2$

Thus, by the Lemma X2L, we have
5. $C_{02_i}^1 \prec_{lock} C_{18_i}^2$

From the algorithm,
6. The ownership of $lock[i]$ is respected.

By the Lemma TryLock on 6 and 5, we have
7. $C_{18_i}^2 \prec_{lock} C_{02_i}^2$

By P2X on the algorithm, we have

8. $C_{07_i}^1 \prec_H C_{18_i}^1$
9. $C_{02_i}^2 \prec_H C_{07_i}^2$

By the Lemma XLTrans on 8, 7, and 9, we have
10. $C_{07_i}^1 \prec_{H} C_{07_i}^2$

By the Lemma X2L on 10, we have
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 \( R \).
The argument of $W'$ is $v''$. By the Lemma AReg' on [6], [7], and [5], we have $v' = v''$. From [3] and [6], we have $W' \prec_{v[e]} W$. By the algorithm and [1] and [6], we have $W$ and $W''$ are both at $C_{17}$. By Lemma 52 on [10], [9], [4] and [7], we have $v'' < v$. From [8] and [11], we have $v' < v$. 

□
Lemma 55 TL2 is global-write-observant.

\[ \forall H \in \mathbb{H}(TL2) : \forall R \in \text{GlobalTReads}(H) : \exists W \in \text{GlobalTWrites}(H) : \text{Let } T' = \text{trans}_H(W) : \]
\[ \text{LastPreAccessor}_{H} (T', R) \land \]
\[ \text{arg}_{1H}(R) = \text{arg}_{1H}(W) \land \text{retv}_{H}(R) = \text{arg}_{2H}(W) \]

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 \( R_{04} \).
We first show that
\[ 3 \] The read method call from \( \text{reg}[i] \) at \( R_{04} \) is race-free.
We assume that there is a write method call on \( \text{reg}[i] \) concurrent to it and show that TL2 aborts \( R \).
Figure 10 depicts this situation.

<table>
<thead>
<tr>
<th>( T )</th>
<th>( T' )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( C_{02i} ) ( \triangleright ) locked = \text{lock}[i].trylock()</td>
<td></td>
</tr>
<tr>
<td>( R_{03} ) ( \triangleright ) ( s_1 = \text{ver}[i].\text{read}() )</td>
<td></td>
</tr>
<tr>
<td>( R_{04} ) ( \triangleright ) ( v = \text{reg}[i].\text{read}() )</td>
<td></td>
</tr>
<tr>
<td>( R_{05} ) ( \triangleright ) ( \text{lock}[i].\text{read}() )</td>
<td></td>
</tr>
<tr>
<td>( R_{06} ) ( \triangleright ) ( s_2 = \text{ver}[i].\text{read}() )</td>
<td></td>
</tr>
<tr>
<td>( R_{07} ) ( \triangleright ) ( \text{sv}er = \text{rver}[t].\text{read}() )</td>
<td></td>
</tr>
<tr>
<td>if ( (-l \land s_1 = s_2 \land s_2 \leq \text{sv}er) )</td>
<td></td>
</tr>
<tr>
<td>return ( A )</td>
<td></td>
</tr>
</tbody>
</table>

Figure 10: \( R_{04} \) is race-free

We assume that there a race between \( R_{04} \) and \( C_{16i} \). Thus,
\[ 4 \] \( R_{04} \sim C_{16i} \)
The method calls \( R_{05} \) and \( C_{18i} \) are on the object \( \text{lock}[i] \).
We consider two cases for the linearization order of them and prove that \( R \) returns \( A \) in both cases.
We consider two cases
Case 1:
\[ 5 \] \( R_{04} \prec_{\text{lock}[i]} C_{18i} \)
By P2X and the algorithm, we have
\[ 6 \] \( C_{02i} \prec_{H} C_{16i} \)
\[ 7 \] \( R_{04} \prec_{H} R_{05} \)
By the Lemma XXTRANS on \( 6 \), \( 4 \), and \( 7 \), we have
\[ 8 \] \( C_{02i} \prec_{H} R_{05} \)
By the Lemma X2L on \( 8 \), we have
\[ 9 \] \( C_{02i} \prec_{\text{lock}[i]} R_{05} \)
By the Lemma TRYLOCKREADM on \( 9 \) and \( 5 \), we have that \( R_{05} \) returns true i.e. \( l = true \)

50
Thus, the validation check fails and \( R \) returns \( A \).

Case 2:

\begin{align*}
(10) \quad & C_{18i} \prec_{\text{lock}[i]} R_{04} \\
\text{By P2X and the algorithm, we have} & \\
(11) \quad & R_{03} \prec_{H} R_{04} \\
(12) \quad & R_{05} \prec_{H} R_{06} \\
(13) \quad & C_{16i} \prec_{H} C_{17i} \\
(14) \quad & C_{17i} \prec_{H} C_{18i} \\
\text{By the Lemma XXTrans on (11), (4), and (13), we have} & \\
(15) \quad & R_{03} \prec_{H} C_{17i} \\
\text{By Lemma 54 on (15), we have} & \\
(16) \quad & s_{1} \prec_{w} \text{wver} \\
\text{By the Lemma XLTrans on (14), (10), and (12), we have} & \\
(17) \quad & C_{17i} \prec_{H} R_{06} \\
\text{By Lemma 53 on (17), we have} & \\
(18) \quad & s_{2} \prec_{w} \text{wver} \\
\text{From (15) and (17), we have} & \\
(19) \quad & s_{1} \neq s_{2} \\
\text{Thus,} & \\
\text{The validation check fails and } R \text{ returns } A.
\end{align*}

Second, we show that

\begin{enumerate}
\item[(20)] The register \( reg[i] \) is sequentially-written i.e. no two write methods on \( reg[i] \) are concurrent.
\end{enumerate}

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

Figure 11 depicts this situation.

<table>
<thead>
<tr>
<th>( T )</th>
<th>( T' )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( C'_{02i} \succ locked = lock[i].trylock() )</td>
<td>( C'_{02i} \succ locked' = lock[i].trylock() )</td>
</tr>
<tr>
<td>( \ldots )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( C'_{16i} \succ \ v = reg[i].write(v) )</td>
<td>( C'_{16i} \succ \ v' = reg[i].write(v') )</td>
</tr>
<tr>
<td>( \ldots )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>( C'_{18i} \succ lock[i].unlock() )</td>
<td>( C'_{18i} \succ lock[i].unlock() )</td>
</tr>
</tbody>
</table>

Figure 11: \( reg[i] \) is sequentially-written

\begin{enumerate}
\item[(21)] \( C_{16i} \sim C_{16'i} \) \\
\text{By P2X and the algorithm, we have} \\
\item[(22)] \( C_{02i} \prec_{H} C_{16i} \) \\
\item[(23)] \( C_{16'i} \prec_{H} C_{18'i} \) \\
\text{By the Lemma XXTrans on (22), (21), and (23), we have} \\
\item[(24)] \( C_{02i} \prec_{H} C_{18'i} \) \\
\text{By the Lemma X2L on (8), we have} \\
\item[(25)] \( C_{02i} \prec_{lock[i]} C_{18'i} \) \\
\text{By the Lemma TryLock on (25), we have that}
\end{enumerate}
By P2X and the algorithm, we have

\( C_{18i} \prec_{\text{lock}[i]} C_{02_i} \)

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

\( C_{16i} \prec H C_{18i} \)

\( C_{02_i} \prec H C_{16_i} \)

That is a contradiction to [21].

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

\( \text{There is a write method call } w \text{ on } \text{reg}[i] \text{ such that} \)

The argument of \( w \) is equal to the return value of \( R_{04} \).

The last write method call on \( \text{reg}[i] \) that is executed before \( R_{04} \) is \( w \).

By the algorithm, we have

\( \text{The register } \text{reg}[i] \text{ is written only at } C_{16_i}. \)

From [28] and [29], we have

\( \text{There is a transaction } T'} \text{ such that} \)

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

\( \text{The argument of } C_{16_i}' \text{ is equal to the return value of } R_{04}. \)

\( \text{The last write method call on } \text{reg}[i] \text{ that is executed before } R_{04} \text{ is } C_{16_i}'. \)

By the algorithm, we have

\( \text{The argument of } C_{16_i}' \text{ is the value of the key } i \text{ in the map } wset[T'] \text{ in the commit.} \)

\( \text{The map } wset[T'] \text{ is updated only at } W_{01} \text{ in a write of } T' \text{ such that} \)

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

\( \text{There exists a write } W \text{ of } T' \)

\( \text{The first argument of } W \text{ is equal to } i. \)

\( W \text{ is the last write of } T' \text{ with the first argument equal to } i. \)

\( \text{The second argument of } W \text{ is equal to the argument of } C_{16_i}'. \)

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

\( \text{The first argument of } R \text{ is the first argument of } W. \)

From [12], [32], and [39], we have

\( \text{The return value of } R \text{ is the second argument of } W. \)

From [38], we have

\( W \text{ is a global write.} \)

We show that

\( \text{The transaction } T' \text{ is the last pre-accessor of } R. \)

From [33], we have

\( C_{16_i}' \prec H R_{04} \)

By Definition [13] on [44], we have

\( T' \sqsubset R \)

Now, we show that

\( \text{Every transaction } T'' \text{ other than } T' \text{ that accesses } i \text{ before } R, \text{ takes effect before } T'. \)

We assume that

\( T'' \neq T' \)
We should show that $T'' \subseteq T'$

By Definition 13 on 48, we have

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

From 33, 33, and 49, we have

Consider Figure 12.

<table>
<thead>
<tr>
<th>$T''$</th>
<th>$T'$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C02_i'' \triangleright locked'' = lock[i].tryLock()$</td>
<td>$C02_i' \triangleright locked' = lock[i].tryLock()$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$C07'' \triangleright wver'' = clock.iaf()$</td>
<td>$C07' \triangleright wver' = clock.iaf()$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$C16_i'' \triangleright reg[i].write(v'')$</td>
<td>$C16_i' \triangleright reg[i].write(v')$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$C18_i'' \triangleright lock[i].unlock()$</td>
<td>$C18_i' \triangleright lock[i].unlock()$</td>
</tr>
</tbody>
</table>

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) $C07'' \prec_H C18_i''$

(57) $C07_i' \prec_H C07_i''$

By the Lemma XLTRANS on 56, 55, and 57, we have

(58) $C07'' \prec_H C07_i'$

By Definition 13 on 58, we have

$T'' \subseteq T'$.

The conclusion is

36, 42, 40, 41, and 43

\[ \square \]

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

\[
\forall H \in \mathbb{H}(TL2):
\forall R \in LocalTReads(H): \text{Let } T = trans_H(R), i = \text{arg}_1H(R), H' = H|T|i:
\exists W \in T\text{Writes}(H'):\nW \prec_{H'} R \land \text{NoWriteBetween}_{H'}(W, R) \land \text{retv}_{H'}(R) = \text{arg}_2H'(W)
\]
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**.

\[ \Box \]

**Lemma 57** \( TL2 \) is write-observing.

\[ \forall H \in \mathbb{H}(TL2): WriteObs(H, \sqsubseteq) \]

**Proof.** Immediate from Lemma 56 and Lemma 55. \[ \Box \]
Lemma 58  TL2 is real-time-preserving.

\[ \forall H \in \mathbb{H}(TL2): \text{RealTimePres}(H, \sqsubseteq) \]

Proof Sketch.
We assume that
1. \( T \preceq_H T' \)
We show that
\( T \sqsubseteq T' \)
By the definition of \( \preceq_H \), 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.

\[ \forall H \in \mathbb{H}(TL2): \sqsubseteq \in \text{Marking}(H) \]

Proof Sketch.
Consider Definition 13.

By the totality of the linearization order \( \prec_{\text{clock}} \), 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 \( i \).

\[ \square \]
Lemma 60  $TL_2$ is markable.

\[ \forall H \in \mathbb{H}(TL_2): H \in FinalStateMarkable \]

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

Theorem 61  $TL_2$ is opaque.

\[ \forall H \in \mathbb{H}(TL_2): H \in FinalStateOpaque \]

Proof.
Immediate from Lemma 60 and Theorem 18.
7 Marking DSTM (visible reads)

<table>
<thead>
<tr>
<th>T:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loc {</td>
</tr>
<tr>
<td>writer: BasicRegister,</td>
</tr>
<tr>
<td>rset: BasicSet,</td>
</tr>
<tr>
<td>oldVal: BasicRegister,</td>
</tr>
<tr>
<td>newVal: BasicRegister },</td>
</tr>
<tr>
<td>state: AtomicCASRegister[],</td>
</tr>
<tr>
<td>start: AtomicCASRegister[]</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>D:</th>
</tr>
</thead>
<tbody>
<tr>
<td>def init_t()</td>
</tr>
<tr>
<td>I01 ▷ state[t].write((R)),</td>
</tr>
<tr>
<td>I02 ▷ return ok,</td>
</tr>
<tr>
<td>def read_t(i)</td>
</tr>
<tr>
<td>R01 ▷ (r_1 = \text{start}[i].\text{read}()),</td>
</tr>
<tr>
<td>R02 ▷ (v = \text{currentValue}_t(r_1)),</td>
</tr>
<tr>
<td>R03 ▷ (r_2 = \text{clone}(r_1)),</td>
</tr>
<tr>
<td>R04 ▷ (r_2.\text{rset}.\text{add}(t)),</td>
</tr>
<tr>
<td>R05 ▷ (rd = \text{start}[i].\text{cas}(r_1, r_2)),</td>
</tr>
<tr>
<td>R06 ▷ (s = \text{state}[t].\text{read}()),</td>
</tr>
<tr>
<td>if ((\neg rd \lor (s = A)))</td>
</tr>
<tr>
<td>R07 ▷ return A</td>
</tr>
<tr>
<td>else</td>
</tr>
<tr>
<td>R08 ▷ return v,</td>
</tr>
<tr>
<td>{R05 \rightarrow R06},</td>
</tr>
<tr>
<td>def commit_t()</td>
</tr>
<tr>
<td>C01 ▷ (c = \text{state}[t].\text{cas}(R, C)),</td>
</tr>
<tr>
<td>if (c)</td>
</tr>
<tr>
<td>C02 ▷ return C</td>
</tr>
<tr>
<td>else</td>
</tr>
<tr>
<td>C03 ▷ return A,</td>
</tr>
<tr>
<td>def currentValue_t(r)</td>
</tr>
<tr>
<td>V01 ▷ (t_2 = r.\text{writer}.\text{read}()),</td>
</tr>
<tr>
<td>if ((\neg (t_2 = t)))</td>
</tr>
<tr>
<td>V02 ▷ (\text{state}[t_2].\text{cas}(R, A)),</td>
</tr>
<tr>
<td>V03 ▷ (s = \text{state}[t_2].\text{read}()),</td>
</tr>
<tr>
<td>if (s = A)</td>
</tr>
<tr>
<td>V04 ▷ return r.oldVal</td>
</tr>
<tr>
<td>else</td>
</tr>
<tr>
<td>V05 ▷ return r.newVal,</td>
</tr>
<tr>
<td>def write_t(i, v)</td>
</tr>
<tr>
<td>W01 ▷ (r_1 = \text{start}[i].\text{read}()),</td>
</tr>
<tr>
<td>W02 ▷ (w = r_1.\text{writer}.\text{read}()),</td>
</tr>
<tr>
<td>if ((w = t))</td>
</tr>
<tr>
<td>W03 ▷ (r_1.\text{newVal}.\text{write}(v)),</td>
</tr>
<tr>
<td>W04 ▷ return ok,</td>
</tr>
<tr>
<td>W05 ▷ (v_2 = \text{currentValue}_t(r_1)),</td>
</tr>
<tr>
<td>W06 ▷ foreach (t_2 \in r_1.\text{rset})</td>
</tr>
<tr>
<td>W07 ▷ (\text{state}[t_2].\text{cas}(R, A)),</td>
</tr>
<tr>
<td>W08 ▷ (r_2 = \text{new \text{Loc}}()),</td>
</tr>
<tr>
<td>W09 ▷ (r_2.\text{writer}.\text{write}(t)),</td>
</tr>
<tr>
<td>W10 ▷ (r_2.\text{oldVal}.\text{write}(v_2)),</td>
</tr>
<tr>
<td>W11 ▷ (r_2.\text{newVal}.\text{write}(v)),</td>
</tr>
<tr>
<td>W12 ▷ (wd = \text{start}[i].\text{cas}(r_1, r_2)),</td>
</tr>
<tr>
<td>if (wd)</td>
</tr>
<tr>
<td>W13 ▷ return ok</td>
</tr>
<tr>
<td>else</td>
</tr>
<tr>
<td>W14 ▷ return A</td>
</tr>
<tr>
<td>{W06 \rightarrow W12}</td>
</tr>
</tbody>
</table>

Figure 13: DSTMV is DSTM (visible reads) Algorithm Specification
**Notation.** Let us remind the notation. Consider an execution history \( H \).

We write \( e_1 \triangleleft_H e_2 \) 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 \( l_1 \preceq_H l_2 \) to denote that \( l_1 \) is executed before or concurrently to \( l_2 \).

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

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 \( \text{initOf}_H(T) \) and \( \text{commitOf}_H(T) \) to denote the \( \text{init} \) and \( \text{commit} \) method calls of the transaction \( T \) in the history \( H \). We use \( \text{LastTRead}_H(T) \) to denote the last read method call by the transaction \( T \) in the history \( H \). We use \( \text{FirstTWrite}_H(T,i) \) 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 \( H \in \mathbb{H}(\text{DSTMVis}) \). Let

\[
\text{Eff}(T) = \begin{cases} 
\text{commitOf}_H(T)'C01 & \text{if } T \in \text{Committed}(H) \\
\text{LastTRead}_H(T)'R05 & \text{if } T \in \text{Aborted}(H) \land \text{TReads}(H) \neq \emptyset \\
\text{initOf}_H(T)'I01 & \text{if } T \in \text{Aborted}(H) \land \text{TReads}(H) = \emptyset 
\end{cases}
\]

\[
\text{readAcc}(R) = R'R05 \\
\text{writeAcc}(T,i) = \text{FirstTWrite}_H(T,i)'W12
\]

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

\[
\{(T,T') | T,T' \in \text{Trans}(H) \land \text{inv(\text{Eff}(T))} \prec_H \text{inv(\text{Eff}(T'))} \} \cup \\
\{(T,R) | \exists i: R \in \text{GlobalTReads}(H), i = \arg1(R), T \in \text{Writers}_H(i) \land \text{writeAcc}(T,i) \prec_{\text{start}[i]} \text{readAcc}(R) \} \cup \\
\{(R,T) | \exists i: R \in \text{GlobalTReads}(H), i = \arg1(R), T \in \text{Writers}_H(i) \land \text{readAcc}(R) \prec_{\text{start}[i]} \text{writeAcc}(T,i) \}
\]

A committed transactions takes effect at the invocation event of \( C01 \), the \( \text{cas} \) 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

\( T: \)
- \( seqLock: \text{SeqLock}, \)
- \( reg: \text{BasicRegister}[] \)
- \( snap: \text{ThreadLocal BasicRegister}, \)
- \( rset: \text{ThreadLocal BasicMap}, \)
- \( wset: \text{ThreadLocal BasicMap}. \)

\( D: \)

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

def read_t(i)
    R01: pv = wset[t].get(i),
        if (pv \neq \bot)
        R02: return pv,
    R03: v = reg[i].read(),
    R04: s1 = snap[t].read(),
    R05: (s2, l2) = seqLock.read(),
        if (s2 = s1 \land \neg l2)
    R06: break,
    R07: b = validate_t(),
        if (\neg b)
    R08: return 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 A

```

```python
def validate_t()
    V01: while (true)
        do
            (s1, l1) = seqLock.read(),
            while (l1)
            foreach ((i, v) \in rset[t])
            V03: v' = reg[i].read(),
                if (v \neq v'),
            V04: return false,
            V05: (s2, l2) = seqLock.read(),
                if (s2 = s1 \land \neg l2)
            V06: snap[t].write(s1),
            V07: return true,
            \{V02 \rightarrow V03, V03 \rightarrow V05\},

def commit_t()
C01: e = wset[t].isEmpty(),
    if (e)
C02: return C,
    do
C03: s = snap[t].read(),
C04: d = seqLock.compareAndLock(s),
    if (d)
C05: break,
C06: b = validate_t(),
    if (\neg b)
            return A,
        while (true),
    foreach ((i, v) \in wset[t])
C07: reg[i].write(v),
C08: seqLock.incAndUnlock(),
C09: return C
    \{C04 \rightarrow C07, C07 \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 $l_1 \preceq_H l_2$ 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 $initOf_H(T)$ and $commitOf_H(T)$ 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

$$REff(T) = \text{The last execution of } I^{01} \text{ or } V^{05}$$
$$Eff(T) = \begin{cases} REff(T) & \text{if } T \in Aborted(H) \lor TWrites(H) = \emptyset \\ commitOf(T)'C^{04} & \text{if } T \in Committed(H) \land TWrites(H) \neq \emptyset \end{cases}$$
$$\text{readAcc}(T,i) = \begin{cases} R'R^{03} & \text{if } REff(T) \prec_H R'R^{03} \\ \text{Let } REff(T) = V'V^{05} \text{ in } V'V^{03}_i & \text{if } R'R^{03} \prec_H REff(T) \end{cases}$$
$$\text{writeAcc}(T,i) = commitOf(T)'C^{07}_i$$

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

$$\{(T,T') \mid T,T' \in Trans(H) \land Eff(T) \prec_{seqLock} Eff(T')\} \cup$$
$$\{(T,R) \mid \exists i: R \in GlobalTReads(H), i = arg_1(R), T \in Writers_H(i) \land writeAcc(T,i) \prec_H readAcc(T,i)\} \cup$$
$$\{(R,T) \mid \exists i: R \in GlobalTReads(H), i = arg_1(R), T \in Writers_H(i) \land readAcc(T,i) \prec_H writeAcc(T,i)\}$$

An aborted transaction or a read-only transaction takes effect at the last execution of $I^{01}$ or $V^{05}$. 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 $C^{04}$.

The access point of a read method call is at $R^{03}$ if the last recent snapshot is read before $R^{03}$; otherwise, it is at $V^{03}_i$ of the latest successful validate method call. The access point of a writer transaction to location $i$ is at $C^{07}_i$. 
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 \( \Omega(|R|) \) where \( R \) 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 \( C \) but has returned \( A \) 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 \( R \) 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 \( k \in R \), 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
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 $A$. 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 $\text{Visible}(S'', T_3) | j = \text{write}_{T_0}(j, v_0) \cdot \text{write}_{T_1}(j, v_1) \cdot \text{read}_{T_3}(j) : v_0 \notin \text{SeqSpec}(j)$. 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, $\text{Visible}(S'', T_3) | j = \text{write}_{T_0}(j, v_0) \cdot \text{write}_{T_1}(i, v_1) \cdot \text{read}_{T_3}(j) : v_0 \notin \text{SeqSpec}(j)$. 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 $\text{Visible}(S'', T_1) | i = \text{write}_{T_0}(i, v_0) \cdot \text{write}_{T_3}(i, v_1) \cdot \text{read}_{T_3}(i) : v_0 \notin \text{SeqSpec}(i)$. 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 $A$.

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 $C$ but has returned $A$ 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 $R$.

1. Invoke $\text{init}_{T_1}()$ and wait for the response.
2. For each $i \in R$:
   2.1. Invoke $\text{read}_{T_1}(i)$ and wait for the response.
3. Invoke $\text{init}_{T_2}(k)$ and wait for the response.
4. Invoke $\text{write}_{T_2}(k, v_1)$ and wait for the response.
5. Invoke $\text{commit}_{T_2}(k)$ and wait for the response.
6. Invoke $\text{init}_{T_3}()$ and wait for the response.
7. Invoke $\text{read}_{T_3}(j)$ and wait for the response.
8. Invoke $\text{abort}_{T_3}()$ and wait for the response.
9. Invoke $\text{write}_{T_1}(j, v_1)$ and wait for the response.
10. Invoke $\text{commit}_{T_1}()$ and wait for the response.

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

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

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 $\Omega(|R|)$ where $R$ 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 $|R|/c$ 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 $k \in R$, 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 $T^k_2, k \in \mathcal{R}$ 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


