Logical Methods in Computer Science
Vol. 14(1:9)2018, pp. 1–46
https://lmcs.episciences.org/

Submitted Feb. 01, 2017
Published Jan. 23, 2018

# A LOAD-BUFFER SEMANTICS FOR TOTAL STORE ORDERING *

PAROSH AZIZ ABDULLA [a], MOHAMED FAOUZI ATIG [b], AHMED BOUAJJANI [c], AND TUAN PHONG NGO [d]

[a,b,d] Uppsala University, Sweden
*e-mail address*: parosh@it.uu.se
*e-mail address*: mohamed_faouzi.atig@it.uu.se
*e-mail address*: tuan-phong.ngo@it.uu.se

[c] IRIF Université Paris Diderot - Paris 7, France
*e-mail address*: abou@liafa.univ-paris-diderot.fr

Abstract. We address the problem of verifying safety properties of concurrent programs running over the Total Store Order (TSO) memory model. Known decision procedures for this model are based on complex encodings of store buffers as lossy channels. These procedures assume that the number of processes is fixed. However, it is important in general to prove the correctness of a system/algorithm in a parametric way with an arbitrarily large number of processes.

In this paper, we introduce an alternative (yet equivalent) semantics to the classical one for the TSO semantics that is more amenable to efficient algorithmic verification and for the extension to parametric verification. For that, we adopt a *dual* view where *load buffers* are used instead of store buffers. The flow of information is now from the memory to load buffers. We show that this new semantics allows (1) to simplify drastically the safety analysis under TSO, (2) to obtain a spectacular gain in efficiency and scalability compared to existing procedures, and (3) to extend easily the decision procedure to the parametric case, which allows obtaining a new decidability result, and more importantly, a verification algorithm that is more general and more efficient in practice than the one for bounded instances.

*2012 ACM CCS:* [**Software and its engineering**]: Software organization and properties—Software functional properties—Formal methods—Software verification.

*Key words and phrases:* Total Store Order, Weak Memory Models, Reachability Problem, Parameterized Systems, Well-quasi-ordering.

* A preliminary version of this paper appeared as at CONCUR'16 [AABN16].

This work was supported in part by the Swedish Research Council and carried out within the Linnaeus centre of excellence UPMARC, Uppsala Programming for Multicore Architectures Research Center.

LOGICAL METHODS IN COMPUTER SCIENCE
DOI:10.23638/LMCS-14(1:9)2018

## 1. Introduction

Most modern processor architectures execute instructions in an out-of-order manner to gain efficiency. In the context of *sequential* programming, this out-of-order execution is transparent to the programmer since one can still work under the Sequential Consistency (SC) model [Lam79]. However, this is not true when we consider concurrent processes that share the memory. In fact, it turns out that concurrent algorithms such as mutual exclusion and producer-consumer protocols may not behave correctly any more. Therefore, program verification is a relevant (and difficult) task in order to prove correctness under the new semantics. The out-of-order execution of instructions has led to the invention of new program semantics, so called *Weak (or relaxed) Memory Models* (WMMs), by allowing permutations between certain types of memory operations [AG96, DSB86, AH90]. Total Store Ordering (TSO) is one of the the most common models, and it corresponds to the relaxation adopted by Sun's SPARC multiprocessors [WG94] and formalizations of the x86-TSO memory model [OSS09, SSO$^+$10]. These models put an unbounded perfect (non-lossy) *store buffer* between each process and the main memory where a store buffer carries the pending store operations of the process. When a process performs a store operation, it appends it to the end of its buffer. These operations are propagated to the shared memory non-deterministically in a FIFO manner. When a process reads a variable, it searches its buffer for a pending store operation on that variable. If no such a store operation exists, it fetches the value of the variable from the main memory. Verifying programs running on the TSO memory model poses a difficult challenge since the unboundedness of the buffers implies that the state space of the system is infinite even in the case where the input program is finite-state. Decidability of safety properties has been obtained by constructing equivalent models that replace the perfect store buffer by *lossy* channels [ABBM10, ABBM12, AAC$^+$12a]. However, these constructions are complicated and involve several ingredients that lead to inefficient verification procedures. For instance, they require each message inside a lossy channel to carry (instead of a single store operation) a full snapshot of the memory representing a local view of the memory contents by the process. Furthermore, the reductions involve non-deterministic guessing the lossy channel contents. The guessing is then resolved either by consistency checking [ABBM10] or by using explicit pointer variables (each corresponding to one process) inside the buffers [AAC$^+$12a], causing a serious state space explosion problem.

In this paper, we introduce a novel semantics which we call the *Dual TSO* semantics. Our aim is to provide an alternative (and equivalent) semantics that is more amenable for efficient algorithmic verification. The main idea is to have *load buffers* that contain pending load operations (more precisely, values that will potentially be taken by forthcoming load operations) rather than store buffers (that contain store operations). The flow of information will now be in the reverse direction, i.e., store operations are performed by the processes atomically on the main memory, while values of variables are propagated non-deterministically from the memory to the load buffers of the processes. When a process performs a load operation, it can fetch the value of the variable from the head of its load buffer. We show that the Dual TSO semantics is equivalent to the original one in the sense that any given set of processes will reach the same set of local states under both semantics. The Dual TSO semantics allows us to understand the TSO model in a totally different way compared to the classical semantics. Furthermore, the Dual TSO semantics offers several important advantages from the point of view of formal reasoning and program verification. First, the Dual TSO semantics allows transforming the load buffers to *lossy* channels without adding the costly overhead that was necessary in the case of store buffers. This means that we can assume w.l.o.g. that any message in the load buffers (except a finite number of messages) can be lost in non-deterministic manner. Hence, we can apply the theory of *well-structured systems* [Abd10, ACJT96, FS01] in a straightforward manner leading to a much simpler proof of decidability of safety properties. Second, the absence of extra overhead means that we obtain more efficient algorithms and better scalability (as shown by our experimental results). Finally, the Dual TSO semantics allows extending the framework to perform *parameterized verification* which is an important paradigm in concurrent program verification. Here, we consider systems, e.g., mutual exclusion protocols, that consist of an arbitrary number of processes. The aim of parameterized verification is to prove correctness of the system regardless of the number of processes. It is not obvious how to perform parameterized verification under the classical semantics. For instance, extending the framework of [AAC+12a], would involve an unbounded number of pointer variables, thus leading to channel systems with unbounded message alphabets. In contrast, as we show in this paper, the simple nature of the Dual TSO semantics allows a straightforward extension of our verification algorithm to the case of parameterized verification. This is the first time a decidability result is established for the parametrized verification of programs running over WMMs. Notice that this result is taking into account two sources of infinity: the number of processes and the size of the buffers.

Based on our framework, we have implemented a tool and applied it to a large set of benchmarks. The experiments demonstrate the efficiency of the Dual TSO semantics compared to the classical one (by two order of magnitude in average), and the feasibility of parametrized verification in the former case. In fact, besides its theoretical generality, parametrized verification is practically crucial in this setting: as our experiments show, it is much more efficient than verification of bounded-size instances (starting from a number of components of 3 or 4), especially concerning memory consumption (which also is a critical resource).

**Related Work.** There have been a lot of works related to the analysis of programs running under WMMs (e.g., [LNP+12, KVY10, KVY11, DMVY13, AAC+12a, BM08, BSS11, BDM13, BAM07, YGLS04, AALN15, AAC+12b, AAJL16, DMVY17, TW16, LV16, LV15, Vaf15, HVQF16]). Some of these works propose precise analysis techniques for checking safety properties or stability of finite-state programs under WMMs (e.g., [AAC+12a, BDM13, DM14, AAP15, AALN15]). Others propose context-bounded analyzing techniques (e.g., [ABP11, TLI+16, TLF+16, AABN17]) or stateless model-checking techniques (e.g., [AAA+15, ZKW15, DL15, HH16]) for programs under TSO and PSO. Different other techniques based on monitoring and testing have also been developed during these last years (e.g., [BM08, BSS11, LNP+12]). There are also a number of efforts to design bounded model checking techniques for programs under WMMs (e.g., [AKNT13, AKT13, YGLS04, BAM07]) which encode the verification problem in SAT/SMT.

The closest works to ours are those presented in [AAC+12a, ABBM10, AAC+13, ABBM12] which provide precise and sound techniques for checking safety properties for finite-state programs running under TSO. However, as stated in the introduction, these techniques are complicated and can not be extended, in a straightforward manner, to the verification of parameterized systems (as it is the case of the developed techniques for the Dual TSO semantics). In Section 6, we experimentally compare our techniques with

Memorax [AAC+12a, AAC+13] which is the only precise and sound tool for checking safety properties for concurrent programs under TSO.

## 2. Preliminaries

Let $\Sigma$ be a finite alphabet. We use $\Sigma^*$ (resp. $\Sigma^+$) to denote the set of all *words* (resp. non-empty words) over $\Sigma$. Let $\epsilon$ be the empty word. The length of a word $w \in \Sigma^*$ is denoted by $|w|$ (and in particular $|\epsilon| = 0$). For every $i : 1 \leq i \leq |w|$, let $w(i)$ be the symbol at position $i$ in $w$. For $a \in \Sigma$, we write $a \in w$ if $a$ appears in $w$, i.e., $a = w(i)$ for some $i : 1 \leq i \leq |w|$.

Given two words $u$ and $v$ over $\Sigma$, we use $u \preceq v$ to denote that $u$ is a (not necessarily contiguous) subword of $v$, i.e., if there is an injection $h : \{1, \ldots, |u|\} \mapsto \{1, \ldots, |v|\}$ such that: (1) $h(i) < h(j)$ for all $i, j : 1 \leq i < j \leq |u|$ and (2) for every $i : 1 \leq i \leq |u|$, we have $u(i) = v(h(i))$.

Given a subset $\Sigma' \subseteq \Sigma$ and a word $w \in \Sigma^*$, we use $w|_{\Sigma'}$ to denote the projection of $w$ over $\Sigma'$, i.e., the word obtained from $w$ by erasing all the symbols that are not in $\Sigma'$.

Let $A$ and $B$ be two sets and let $f : A \mapsto B$ be a total function from $A$ to $B$. We use $f[a \hookleftarrow b]$ to denote the function $g$ such that $g(a) = b$ and $g(x) = f(x)$ for all $x \neq a$.

A transition system $\mathcal{T}$ is a tuple $\left(\mathtt{C}, \mathtt{Init}, \mathtt{Act}, \cup_{a \in \mathtt{Act}} \xrightarrow{a}\right)$ where $\mathtt{C}$ is a (potentially infinite) set of *configurations*; $\mathtt{Init} \subseteq \mathtt{C}$ is a set of *initial configurations*; $\mathtt{Act}$ is a set of actions; and for every $a \in \mathtt{Act}$, $\xrightarrow{a} \subseteq \mathtt{C} \times \mathtt{C}$ is a *transition relation*. We use $c \xrightarrow{a} c'$ to denote that $(c, c') \in \xrightarrow{a}$. Let $\rightarrow := \cup_{a \in \mathtt{Act}} \xrightarrow{a}$.

A *run* $\pi$ of $\mathcal{T}$ is of the form $c_0 \xrightarrow{a_1} c_1 \xrightarrow{a_2} \cdots \xrightarrow{a_n} c_n$ where $c_i \xrightarrow{a_{i+1}} c_{i+1}$ for all $i : 0 \leq i < n$. Then, we write $c_0 \xrightarrow{\pi} c_n$. We use *target* $(\pi)$ to denote the configuration $c_n$. The run $\pi$ is said to be a *computation* if $c_0 \in \mathtt{Init}$. Two runs $\pi_1 = c_0 \xrightarrow{a_1} c_1 \xrightarrow{a_2} \cdots \xrightarrow{a_m} c_m$ and $\pi_2 = c_{m+1} \xrightarrow{a_{m+2}} c_{m+2} \xrightarrow{a_{m+3}} \cdots \xrightarrow{a_n} c_n$ are *compatible* if $c_m = c_{m+1}$. Then, we write $\pi_1 \bullet \pi_2$ to denote the run

$$\pi = c_0 \xrightarrow{a_1} c_1 \xrightarrow{a_2} \cdots \xrightarrow{a_m} c_m \xrightarrow{a_{m+2}} c_{m+2} \xrightarrow{a_{m+3}} \cdots \xrightarrow{a_n} c_n.$$

For two configurations $c$ and $c'$, we use $c \xrightarrow{*} c'$ to denote that $c \xrightarrow{\pi} c'$ for some run $\pi$. A configuration $c$ is said to be *reachable* in $\mathcal{T}$ if $c_0 \xrightarrow{*} c$ for some $c_0 \in \mathtt{Init}$, and a set $C$ of configurations is said to be *reachable* in $\mathcal{T}$ if some $c \in C$ is reachable in $\mathcal{T}$.

## 3. Concurrent Systems

In this section, we define the syntax we use for *concurrent programs*, a model for representing communication of concurrent processes. Communication between processes is performed through a shared memory that consists of a finite number of shared variables (over finite domains) to which all processes can read and write. Then we recall the classical TSO semantics including the transition system it induces and its reachability problem. Next, we introduce the *Dual TSO* semantics and its induced transition system. Finally, we state the equivalence between the two semantics; i.e., for a given concurrent program, we can reduce its reachability problem under the classical TSO semantics to its reachability problem under Dual TSO semantics and vice-versa.

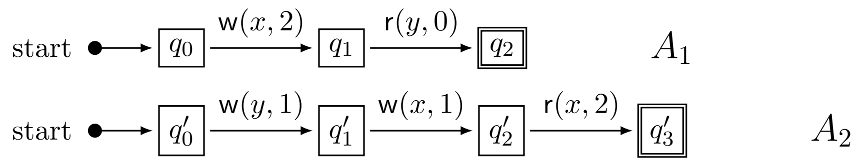


Figure 1: An example of a concurrent system $\mathcal{P} = \{A_1, A_2\}$.

3.1. **Syntax.** Let $\mathbb{V}$ be a finite data domain and $\mathbb{X}$ be a finite set of variables. We assume w.l.o.g. that $\mathbb{V}$ contains the value 0. Let $\Omega(\mathbb{X}, \mathbb{V})$ be the smallest set of memory operations that contains with $x \in \mathbb{X}$ and $v, v' \in \mathbb{V}$:

(1) *"no"* operation $\mathsf{nop}$,
(2) *read* operation $\mathsf{r}(x, v)$,
(3) *write* operation $\mathsf{w}(x, v)$,
(4) *fence* operation $\mathsf{fence}$, and
(5) *atomic read-write* operation $\mathsf{arw}(x, v, v')$.

A *concurrent system* (or a concurrent program) is a tuple $\mathcal{P} = (A_1, A_2, \ldots, A_n)$ where for every $p : 1 \leq p \leq n$, $A_p$ is a finite-state automaton describing the behavior of the process $p$. The automaton $A_p$ is defined as a triple $\left(Q_p, q_p^{init}, \Delta_p\right)$ where $Q_p$ is a finite set of *local states*, $q_p^{init} \in Q_p$ is the *initial* local state, and $\Delta_p \subseteq Q_p \times \Omega(\mathbb{X}, \mathbb{V}) \times Q_p$ is a finite set of *transitions*. We define $\mathbb{P} := \{1, \ldots, n\}$ to be the set of process IDs, $Q := \cup_{p \in \mathbb{P}} Q_p$ to be the set of all local states and $\Delta := \cup_{p \in \mathbb{P}} \Delta_p$ to be the set of all transitions.

**Example 3.1.** Figure 1 shows an example of a concurrent system $\mathcal{P} = \{A_1, A_2\}$ consisting of two concurrent processes, called $p_1$ and $p_2$. Communication between processes is performed through two shared variables $x$ and $y$ to which the processes can read and write. The automaton $A_1$ is defined as a triple $(\{q_0, q_1, q_2\}, \{q_0\}, \{(q_0, \mathsf{w}(x, 2), q_1), (q_1, \mathsf{r}(y, 0), q_2)\})$. Similarly, $A_2 = (\{q'_0, q'_1, q'_2, q'_3\}, \{q'_0\}, \{(q'_0, \mathsf{w}(y, 1), q'_1), (q'_1, \mathsf{w}(x, 1), q'_2), (q'_2, \mathsf{r}(x, 2), q'_3)\})$. $\triangle$

3.2. **Classical TSO Semantics.** In the following, we recall the semantics of concurrent systems under the classical TSO model as formalized in [OSS09, SSO+10]. To do that, we define the set of configurations and the induced transition relation. Let $\mathcal{P} = (A_1, A_2, \ldots, A_n)$ be a concurrent system.

**TSO-configurations.** A *TSO-configuration* $c$ is a triple $(\mathbf{q}, \mathbf{b}, \mathbf{mem})$ where:

(1) $\mathbf{q} : \mathbb{P} \mapsto Q$ is the *global state* of $\mathcal{P}$, mapping each process $p \in \mathbb{P}$ to a local state in $Q_p$ (i.e., $\mathbf{q}(p) \in Q_p$).
(2) $\mathbf{b} : \mathbb{P} \mapsto (\mathbb{X} \times \mathbb{V})^*$ gives the content of the store buffer of each process.
(3) $\mathbf{mem} : \mathbb{X} \mapsto \mathbb{V}$ defines the value of each shared variable.

Observe that the store buffer of each process contains a sequence of write operations, where each write operation is defined by a pair, namely a variable $x$ and a value $v$ that is assigned to $x$.

The *initial* TSO-configuration $c_{init}$ is defined by the tuple $(\mathbf{q}_{init}, \mathbf{b}_{init}, \mathbf{mem}_{init})$ where, for all $p \in \mathbb{P}$ and $x \in \mathbb{X}$, we have that $\mathbf{q}_{init}(p) = q_p^{init}$, $\mathbf{b}_{init}(p) = \epsilon$ and $\mathbf{mem}_{init}(x) = 0$. In other words, each process is in its initial local state, all the buffers are empty, and all the variables in the shared memory are initialized to 0.

$$\frac{t = (q, \mathsf{nop}, q') \qquad \mathbf{q}(p) = q}{(\mathbf{q}, \mathbf{b}, \mathbf{mem}) \xrightarrow{t}_{\mathsf{TSO}} (\mathbf{q}\,[p \hookleftarrow q']\,, \mathbf{b}, \mathbf{mem})} \quad \text{Nop}$$

$$\frac{t = (q, \mathsf{w}(x,v), q') \qquad \mathbf{q}(p) = q}{(\mathbf{q}, \mathbf{b}, \mathbf{mem}) \xrightarrow{t}_{\mathsf{TSO}} (\mathbf{q}\,[p \hookleftarrow q']\,, \mathbf{b}\,[p \hookleftarrow (x,v)\cdot \mathbf{b}(p)]\,, \mathbf{mem})} \quad \text{Write}$$

$$\frac{t = \mathsf{update}_p}{(\mathbf{q}, \mathbf{b}\,[p \hookleftarrow \mathbf{b}(p)\cdot (x,v)]\,, \mathbf{mem}) \xrightarrow{t}_{\mathsf{TSO}} (\mathbf{q}, \mathbf{b}, \mathbf{mem}\,[x \hookleftarrow v])} \quad \text{Update}$$

$$\frac{t = (q, \mathsf{r}(x,v), q') \qquad \mathbf{q}(p) = q \qquad \mathbf{b}(p)|_{\{x\}\times\mathbb{V}} = (x,v)\cdot w}{(\mathbf{q}, \mathbf{b}, \mathbf{mem}) \xrightarrow{t}_{\mathsf{TSO}} (\mathbf{q}\,[p \hookleftarrow q']\,, \mathbf{b}, \mathbf{mem})} \quad \text{Read-Own-Write}$$

$$\frac{t = (q, \mathsf{r}(x,v), q') \qquad \mathbf{q}(p) = q \qquad \mathbf{b}(p)|_{\{x\}\times\mathbb{V}} = \epsilon \qquad \mathbf{mem}(x) = v}{(\mathbf{q}, \mathbf{b}, \mathbf{mem}) \xrightarrow{t}_{\mathsf{TSO}} (\mathbf{q}\,[p \hookleftarrow q']\,, \mathbf{b}, \mathbf{mem})} \quad \text{Read from Memory}$$

$$\frac{t = (q, \mathsf{arw}(x,v,v'), q') \qquad \mathbf{q}(p) = q \qquad \mathbf{b}(p) = \epsilon \qquad \mathbf{mem}(x) = v}{(\mathbf{q}, \mathbf{b}, \mathbf{mem}) \xrightarrow{t}_{\mathsf{TSO}} (\mathbf{q}\,[p \hookleftarrow q']\,, \mathbf{b}, \mathbf{mem}\,[x \hookleftarrow v'])} \quad \text{ARW}$$

$$\frac{t = (q, \mathsf{fence}, q') \qquad \mathbf{q}(p) = q \qquad \mathbf{b}(p) = \epsilon}{(\mathbf{q}, \mathbf{b}, \mathbf{mem}) \xrightarrow{t}_{\mathsf{TSO}} (\mathbf{q}\,[p \hookleftarrow q']\,, \mathbf{b}, \mathbf{mem})} \quad \text{Fence}$$

Figure 2: The transition relation $\rightarrow_{\mathsf{TSO}}$ under TSO semantics. Here, process $p \in \mathbb{P}$ and transition $t \in \Delta_p \cup \{\mathsf{update}_p\}$ where $\mathsf{update}_p$ is a transition that updates the memory using the oldest message in the buffer of the process $p$.

We use $\mathsf{C}_{\mathsf{TSO}}$ to denote the set of all TSO-configurations.

**TSO-transition Relation.** The *transition relation* $\rightarrow_{\mathsf{TSO}}$ between TSO-configurations is given by a set of rules, described in Figure 2. Here, we informally explain these rules. A *nop* transition $(q, \mathsf{nop}, q') \in \Delta_p$ changes only the local state of the process $p$ from $q$ to $q'$. A *write* transition $(q, \mathsf{w}(x,v), q') \in \Delta_p$ adds a new message $(x,v)$ to the tail of the store buffer of the process $p$. A memory *update* transition $\mathsf{update}_p$ can be performed at any time by removing the (oldest) message at the head of the store buffer of the process $p$ and updating the memory accordingly. For a *read* transition $(q, \mathsf{r}(x,v), q') \in \Delta_p$, if the store buffer of the process $p$ contains some write operations to $x$, then the read value $v$ must correspond to the value of the most recent such a write operation. Otherwise, the value $v$ of $x$ is fetched from the memory. A *fence* transition $(q, \mathsf{fence}, q') \in \Delta_p$ can be performed by the process $p$ only if its store buffer is empty. Finally, an *atomic read-write* transition $(q, \mathsf{arw}(x,v,v'), q') \in \Delta_p$ can be performed by the process $p$ only if its store buffer is empty. This transition checks whether the value of $x$ in the memory is $v$ and then changes it to $v'$.

Let $\Delta' := \{\mathsf{update}_p | \, p \in \mathbb{P}\}$, i.e., $\Delta'$ contains all memory update transitions. We use $c \rightarrow_{\mathsf{TSO}} c'$ to denote that $c \xrightarrow{t}_{\mathsf{TSO}} c'$ for some $t \in \Delta \cup \Delta'$. The transition system induced by $\mathcal{P}$ under the classical TSO semantics is then given by $\mathcal{T}_{\mathsf{TSO}} := (\mathsf{C}_{\mathsf{TSO}}, \{c_{init}\}, \Delta \cup \Delta', \rightarrow_{\mathsf{TSO}})$.

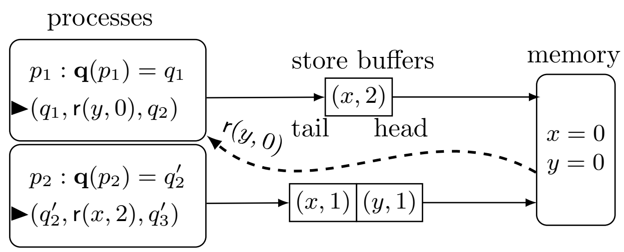


Figure 3: A reachable TSO-configuration of the concurrent system in Figure 1.

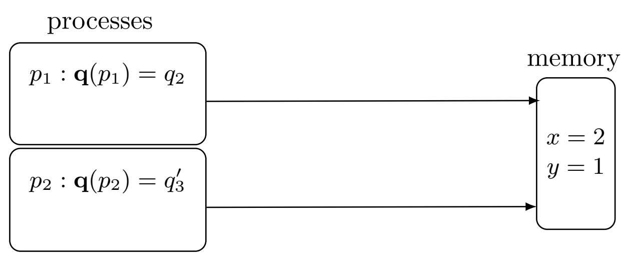


Figure 4: A reachable "empty buffer" TSO-configuration of the concurrent system in Figure 1.

**The TSO Reachability Problem.** A global state $\mathbf{q}_{target}$ is said to be *reachable* in $\mathcal{T}_{\mathsf{TSO}}$ if and only if there is a TSO-configuration $c$ of the form $(\mathbf{q}_{target}, \mathbf{b}, \mathbf{mem})$, with $\mathbf{b}(p) = \epsilon$ for all $p \in \mathbb{P}$, such that $c$ is reachable in $\mathcal{T}_{\mathsf{TSO}}$.

The *TSO reachability problem* for the concurrent system $\mathcal{P}$ under the TSO semantics asks, for a given global state $\mathbf{q}_{target}$, whether $\mathbf{q}_{target}$ is reachable in $\mathcal{T}_{\mathsf{TSO}}$. Observe that, in the definition of the reachability problem, we require that the buffers of the configuration $c$ must be empty instead of being arbitrary. This is only for the sake of simplicity and does not constitute a restriction. Indeed, we can easily show that the "arbitrary buffer" reachability problem is reducible to the "empty buffer" reachability problem.

**Example 3.2.** Figure 3 illustrates a TSO-configuration $c$ that can be reached from the initial configuration $c_{init}$ of the concurrent system in Figure 1. To reach this configuration, the process $p_1$ has executed the write transition $(q_0, \mathsf{w}(x, 2), q_1)$ and appended the message $(x, 2)$ to its store buffer. Meanwhile, the process $p_2$ has executed two write transitions $(q'_0, \mathsf{w}(y, 1), q'_1)$ and $(q'_1, \mathsf{w}(x, 1), q'_2)$. Hence, the store buffer of $p_2$ contains the sequence $(x, 1) \cdot (y, 1)$. Now, the process $p_1$ can perform the read transition $(q_1, \mathsf{r}(y, 0), q_2)$. Since the buffer of $p_1$ does not contain any pending write message on $y$, the read value is fetched from the memory (represented by the dash arrow in Figure 3). Then, $p_1$ and $p_2$ perform the following sequence of update transitions $\mathsf{update}_{p_2} \cdot \mathsf{update}_{p_2} \cdot \mathsf{update}_{p_1}$ to empty their buffers and update the memory to $x = 2$ and $y = 1$. Finally, $p_2$ performs the read transition $(q'_2, \mathsf{r}(x, 2), q'_3)$ (by reading from the memory) to reach to the configuration $c_{target}$ given in Figure 4. Observe that the buffers of both processes are empty in $c_{target}$. Let $\mathbf{q}_{target}$ be the global state in $c_{target}$ defined as follows: $\mathbf{q}_{target}(p_1) = q_2$ and $\mathbf{q}_{target}(p_2) = q'_3$. Therefore, we can say that the global state $\mathbf{q}_{target}$ is reachable in $\mathcal{T}_{\mathsf{TSO}}$. $\triangle$

$$\frac{t = (q, \mathsf{nop}, q') \qquad \mathbf{q}(p) = q}{(\mathbf{q}, \mathbf{b}, \mathbf{mem}) \xrightarrow{t}_{\mathsf{DTSO}} (\mathbf{q}[p \hookleftarrow q'], \mathbf{b}, \mathbf{mem})} \quad \text{Nop}$$

$$\frac{t = (q, \mathsf{w}(x, v), q') \qquad \mathbf{q}(p) = q}{(\mathbf{q}, \mathbf{b}, \mathbf{mem}) \xrightarrow{t}_{\mathsf{DTSO}} (\mathbf{q}[p \hookleftarrow q'], \mathbf{b}[p \hookleftarrow (x, v, own) \cdot \mathbf{b}(p)], \mathbf{mem}[x \hookleftarrow v])} \quad \text{Write}$$

$$\frac{t = \mathsf{propagate}_p^x \qquad \mathbf{mem}(x) = v}{(\mathbf{q}, \mathbf{b}, \mathbf{mem}) \xrightarrow{t}_{\mathsf{DTSO}} (\mathbf{q}, \mathbf{b}[p \hookleftarrow (x, v) \cdot \mathbf{b}(p)], \mathbf{mem})} \quad \text{Propagate}$$

$$\frac{t = \mathsf{delete}_p \qquad |m| = 1}{(\mathbf{q}, \mathbf{b}[p \hookleftarrow \mathbf{b}(p) \cdot m], \mathbf{mem}) \xrightarrow{t}_{\mathsf{DTSO}} (\mathbf{q}, \mathbf{b}, \mathbf{mem})} \quad \text{Delete}$$

$$\frac{t = (q, \mathsf{r}(x, v), q') \qquad \mathbf{q}(p) = q \qquad \mathbf{b}(p)|_{\{x\} \times \mathbb{V} \times \{own\}} = (x, v, own) \cdot w}{(\mathbf{q}, \mathbf{b}, \mathbf{mem}) \xrightarrow{t}_{\mathsf{DTSO}} (\mathbf{q}[p \hookleftarrow q'], \mathbf{b}, \mathbf{mem})} \quad \text{Read-Own-Write}$$

$$\frac{t = (q, \mathsf{r}(x, v), q') \quad \mathbf{q}(p) = q \quad \mathbf{b}(p)|_{\{x\} \times \mathbb{V} \times \{own\}} = \epsilon \quad \mathbf{b}(p) = w \cdot (x, v)}{(\mathbf{q}, \mathbf{b}, \mathbf{mem}) \xrightarrow{t}_{\mathsf{DTSO}} (\mathbf{q}[p \hookleftarrow q'], \mathbf{b}, \mathbf{mem})} \quad \text{Read from Buffer}$$

$$\frac{t = (q, \mathsf{arw}(x, v, v'), q') \qquad \mathbf{q}(p) = q \qquad \mathbf{b}(p) = \epsilon \qquad \mathbf{mem}(x) = v}{(\mathbf{q}, \mathbf{b}, \mathbf{mem}) \xrightarrow{t}_{\mathsf{DTSO}} (\mathbf{q}[p \hookleftarrow q'], \mathbf{b}, \mathbf{mem}[x \hookleftarrow v'])} \quad \text{ARW}$$

$$\frac{t = (q, \mathsf{fence}, q') \qquad \mathbf{q}(p) = q \qquad \mathbf{b}(p) = \epsilon}{(\mathbf{q}, \mathbf{b}, \mathbf{mem}) \xrightarrow{t}_{\mathsf{DTSO}} (\mathbf{q}[p \hookleftarrow q'], \mathbf{b}, \mathbf{mem})} \quad \text{Fence}$$

Figure 5: The induced transition relation $\rightarrow_{\mathsf{DTSO}}$ under the Dual TSO semantics. Here, process $p \in \mathbb{P}$ and transition $t \in \Delta_p \cup \Delta'_p$ where $\Delta'_p := \left\{\mathsf{propagate}_p^x, \mathsf{delete}_p |\ x \in \mathbb{X}\right\}$.

3.3. **Dual TSO Semantics.** In this section, we define the *Dual TSO* semantics. The model has a FIFO *load* buffer between the main memory and each process. This load buffer is used to store *potential read* operations that will be performed by the process. We allow this buffer to *lose* messages at any time by deleting the messages at its head in non-deterministic manner. Each message in the load buffer of a process $p$ is either a pair of the form $(x, v)$ or a triple of the form $(x, v, own)$ where $x \in \mathbb{X}$ and $v \in \mathbb{V}$. A message of the form $(x, v)$ corresponds to the fact that $x$ has had the value $v$ in the shared memory. Meanwhile, a message of the form $(x, v, own)$ corresponds to the fact that the process $p$ has written the value $v$ to $x$. We say that a message $(x, v, own)$ is an *own-message*.

A *write* operation $\mathsf{w}(x, v)$ of the process $p$ immediately updates the shared memory and then appends a new own-message $(x, v, own)$ to the tail of the load buffer of $p$. Read *propagation* is then performed by non-deterministically choosing a variable (let's say $x$ and its value is $v$ in the shared memory) and appending the new message $(x, v)$ to the tail of the load buffer of $p$. This propagation operation speculates on a read operation of $p$ on $x$ that will be performed later on. Moreover, *delete* operation of the process $p$ can be performed at any time by removing the (oldest) message at the head of the load buffer of $p$. A *read* operation $\mathsf{r}(x, v)$ of the process $p$ can be executed if the message at the head of the load buffer of $p$ is of the form $(x, v)$ and there is no pending own-message of the form

$(x, v', own)$. In the case that the load buffer of $p$ contains some own-messages (i.e., of the form $(x, v', own)$), the read value must correspond to the value of the most recent such an own-message. Implicitly, this allows to simulate the Read-Own-Write transitions in the TSO semantics. A *fence* operation means that the load buffer of $p$ must be empty before $p$ can continue. Finally, an *atomic read-write* operation $\mathsf{arw}(x, v, v')$ means that the load buffer of $p$ must be empty and the value of the variable $x$ in the memory is $v$ before $p$ can continue.

**DTSO-configurations.** A *DTSO-configuration* $c$ is a triple $(\mathbf{q}, \mathbf{b}, \mathbf{mem})$ where:

(1) $\mathbf{q} : \mathbb{P} \mapsto Q$ is the *global state* of $\mathcal{P}$.
(2) $\mathbf{b} : \mathbb{P} \mapsto ((\mathbb{X} \times \mathbb{V}) \cup (\mathbb{X} \times \mathbb{V} \times \{own\}))^*$ is the content of the load buffer of each process.
(3) $\mathbf{mem} : \mathbb{X} \mapsto \mathbb{V}$ gives the value of each shared variable.

The *initial* DTSO-configuration $c^D_{init}$ is defined by $(\mathbf{q}_{init}, \mathbf{b}_{init}, \mathbf{mem}_{init})$ where, for all $p \in \mathbb{P}$ and $x \in \mathbb{X}$, we have that $\mathbf{q}_{init}(p) = q_p^{init}$, $\mathbf{b}_{init}(p) = \epsilon$ and $\mathbf{mem}_{init}(x) = 0$.

We use $\mathsf{C}_{\mathsf{DTSO}}$ to denote the set of all DTSO-configurations.

**DTSO-transition Relation.** The *transition relation* $\rightarrow_{\mathsf{DTSO}}$ between DTSO-configurations is given by a set of rules, described in Figure 5. This relation is induced by members of $\Delta \cup \Delta''$ where $\Delta'' := \left\{\mathsf{propagate}_p^x, \mathsf{delete}_p |\ p \in \mathbb{P},\ x \in \mathbb{X}\right\}$.

We informally explain the transition relation rules. The *propagate* transition $\mathsf{propagate}_p^x$ speculates on a read operation of $p$ over $x$ that will be executed later. This is done by appending a new message $(x, v)$ to the tail of the load buffer of $p$ where $v$ is the current value of $x$ in the shared memory. The *delete* transition $\mathsf{delete}_p$ removes the (oldest) message at the head of the load buffer of the process $p$. A *write* transition $(q, \mathsf{w}(x, v), q') \in \Delta_p$ updates the memory and appends a new own-message $(x, v, own)$ to the tail of the load buffer. A *read* transition $(q, \mathsf{r}(x, v), q') \in \Delta_p$ checks first if the load buffer of $p$ contains an own-message of the form $(x, v', own)$. In that case, the read value $v$ should correspond to the value of the most recent such an own-message. If there is no such message on the variable $x$ in the load buffer of $p$, then the value $v$ of $x$ is fetched from the (oldest) message at the head of the load buffer of $p$.

We use $c \rightarrow_{\mathsf{DTSO}} c'$ to denote that $c \xrightarrow{t}_{\mathsf{DTSO}} c'$ for some $t \in \Delta \cup \Delta''$. The transition system induced by $\mathcal{P}$ under the Dual TSO semantics is then given by $\mathcal{T}_{\mathsf{DTSO}} = \left(\mathsf{C}_{\mathsf{DTSO}}, \{c^D_{init}\}, \Delta \cup \Delta'', \rightarrow_{\mathsf{DTSO}}\right)$.

**The DTSO Reachability Problem.** The *DTSO reachability problem* for $\mathcal{P}$ under the Dual TSO semantics is defined in a similar manner to the case of the TSO semantics. A global state $\mathbf{q}_{target}$ is said to be *reachable* in $\mathcal{T}_{\mathsf{DTSO}}$ if and only if there is a DTSO-configuration $c$ of the form $(\mathbf{q}_{target}, \mathbf{b}, \mathbf{mem})$, with $\mathbf{b}(p) = \epsilon$ for all $p \in \mathbb{P}$, such that $c$ is reachable in $\mathcal{T}_{\mathsf{DTSO}}$. Then, the *DTSO reachability problem* consists in checking whether $\mathbf{q}_{target}$ is reachable in $\mathcal{T}_{\mathsf{DTSO}}$.

**Example 3.3.** Figure 6 illustrates a DTSO-configuration $c'$ that can be reached from the initial configuration $c^D_{init}$ of the concurrent system in Figure 1. To reach this configuration, a propagation operation is performed by appending the message $(y, 0)$ into the load buffer of $p_1$. Then, the process $p_2$ executes two write transitions $(q'_0, \mathsf{w}(y, 1), q'_1)$ and $(q'_1, \mathsf{w}(x, 1), q'_2)$ that update the shared memory to $x = 1$ and $y = 1$ and add two own-messages to the tail of the load buffer of $p_2$. Hence, the load buffer of $p_2$ contains the sequence $(x, 1, own) \cdot (y, 1, own)$. Then, the process $p_1$ executes the write transition $(q_0, \mathsf{w}(x, 2), q_1)$ which updates the shared memory and appendes the own-message $(x, 2, own)$ to the tail of the load buffer of $p_1$. After

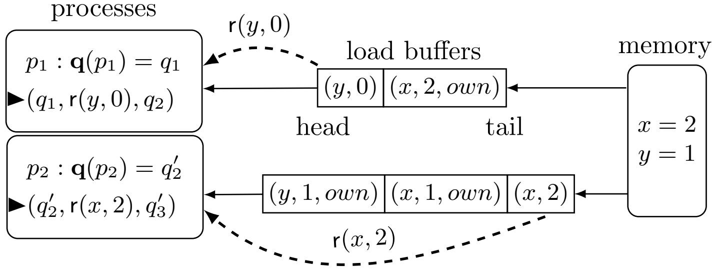


Figure 6: A reachable DTSO-configuration of the concurrent system in Figure 1.


Figure 7: A reachable "empty buffer" DTSO-configuration of the concurrent system in Figure 1.

that, a propagation operation appending the message $(x, 2)$ into the load buffer of $p_2$ is performed. Hence, the value of $x$ (resp. $y$ ) is 2 (resp. 1) in the shared memory. Furthermore, the load buffer of $p_1$ (resp. $p_2$) contains the following sequence $(x, 2, own) \cdot (y, 0)$ (resp, $(x, 2) \cdot (x, 1, own) \cdot (y, 1, own)$). Now from the configuration $c'$ (given in Figure 6), the process $p_1$ can perform a read transition $(q_1, \mathsf{r}(y, 0), q_2)$. Since there is no pending own-message of the form $(y, v, own)$ for some $v \in \mathbb{V}$ in the load buffer of $p_1$, $p_1$ reads from the message at the head of its load buffer, i.e. the message $(y, 0)$ (represented by the dash arrow for $p_1$). Then, $p_2$ performs two delete transitions $\mathsf{delete}_{p_2}$ to remove two own-messages at the head of its load buffer. Now, the process $p_2$ can perform the read transition $(q_2', \mathsf{r}(x, 2), q_3')$ to read from its load buffer. Finally, $p_1$ and $p_2$ performs a sequence of delete transitions $\mathsf{delete}_{p_1} \cdot \mathsf{delete}_{p_1} \cdot \mathsf{delete}_{p_2}$ to empty their load buffers, reaching to the configuration $c'_{target}$ given in Figure 7. Let $\mathbf{q}_{target}$ be the global state in $c'_{target}$ defined as follows: $\mathbf{q}_{target}(p_1) = q_2$ and $\mathbf{q}_{target}(p_2) = q_3'$. Therefore, we can say that the global state $\mathbf{q}_{target}$ in $c'_{target}$ is reachable in $\mathcal{T}_{\mathsf{DTSO}}$. △

3.4. **Relation between TSO and DTSO Reachability Problems.** The following theorem states the equivalence of the reachability problems under the TSO and Dual TSO semantics.

**Theorem 3.4** (TSO-DTSO reachability equivalence)**.** *A global state* $\mathbf{q}_{target}$ *is reachable in* $\mathcal{T}_{\mathsf{TSO}}$ *iff* $\mathbf{q}_{target}$ *is reachable in* $\mathcal{T}_{\mathsf{DTSO}}$.

*Proof.* The proof of this theorem can be found in Appendix A. □

**Example 3.5.** In the Example 3.2 and Example 3.3, we have shown that the global state $\mathbf{q}_{target}$ (defined by $\mathbf{q}_{target}(p_1) = q_2$ and $\mathbf{q}_{target}(p_2) = q_3'$) is both reachable in $\mathcal{T}_{\mathsf{TSO}}$ and $\mathcal{T}_{\mathsf{DTSO}}$ for the concurrent system given in Figure 1. $\triangle$

## 4. The DTSO Reachability Problem

In this section, we show the *decidability* of the DTSO reachability problem by making use of the framework of *Well-Structured Transition Systems* (Wsts) [ACJT96, FS01]. First, we briefly recall the framework of Wsts. Then, we instantiate it to show the decidability of the DTSO reachability problem. Following Theorem 3.4, we also obtain the decidability of the TSO reachability problem.

4.1. **Well-structured Transition Systems.** Let $\mathcal{T} = \left(\mathtt{C}, \mathtt{Init}, \mathtt{Act}, \cup_{a\in\mathtt{Act}} \xrightarrow{a}\right)$ be a transition system. Let $\sqsubseteq$ be a *well-quasi-ordering* on $\mathtt{C}$. Recall that a well-quasi-ordering on $\mathtt{C}$ is a binary relation over $\mathtt{C}$ that is reflexive and transitive; and for every infinite sequence $(c_i)_{i\geq 0}$ of elements in $\mathtt{C}$, there exist $i, j \in \mathbb{N}$ such that $i < j$ and $c_i \sqsubseteq c_j$.

A set $\mathtt{U} \subseteq \mathtt{C}$ is called *upward closed* if for every $c \in \mathtt{U}$ and $c' \in \mathtt{C}$ with $c \sqsubseteq c'$, we have $c' \in \mathtt{U}$. It is known that every upward closed set $\mathtt{U}$ can be characterised by a finite *minor set* $\mathtt{M} \subseteq \mathtt{U}$ such that: (i) for every $c \in \mathtt{U}$, there is $c' \in \mathtt{M}$ such that $c' \sqsubseteq c$; and (ii) if $c, c' \in \mathtt{M}$ and $c \sqsubseteq c'$, then $c = c'$. We use $\mathtt{min}(\mathtt{U})$ to denote for a given upward closed set $\mathtt{U}$ its minor set.

Let $\mathtt{D} \subseteq \mathtt{C}$. The upward closure of $\mathtt{D}$ is defined as $\mathtt{D}\uparrow := \{c' \in \mathtt{C} |\ \exists c \in \mathtt{D} \text{ with } c \sqsubseteq c'\}$. We also define the set of *predecessors* of $\mathtt{D}$ as $\mathtt{Pre}_{\mathcal{T}}(\mathtt{D}) := \left\{c |\ \exists c_1 \in \mathtt{D}, a \in \mathtt{Act}, c \xrightarrow{a} c_1\right\}$. For a finite set of configurations $\mathtt{M} \subseteq \mathtt{C}$, we use $\mathsf{minpre}(\mathtt{M})$ to denote $\mathtt{min}(\mathtt{Pre}_{\mathcal{T}}(\mathtt{M}\uparrow) \cup \mathtt{M}\uparrow)$.

The transition relation $\to$ is said to be *monotonic* wrt. the ordering $\sqsubseteq$ if, given $c_1, c_2, c_3 \in \mathtt{C}$ where $c_1 \to c_2$ and $c_1 \sqsubseteq c_3$, we can compute a configuration $c_4 \in \mathtt{C}$ and a run $\pi$ such that $c_3 \xrightarrow{\pi} c_4$ and $c_2 \sqsubseteq c_4$. The pair $(\mathcal{T}, \sqsubseteq)$ is called a *monotonic transition system* if $\to$ is *monotonic* wrt. $\sqsubseteq$.

Given a *finite* set of configurations $\mathtt{M} \subseteq \mathtt{C}$, the *coverability problem* of $\mathtt{M}$ in the monotonic transition system $(\mathcal{T}, \sqsubseteq)$ asks whether the set $\mathtt{M}\uparrow$ is *reachable* in $\mathcal{T}$; i.e. there exist two configurations $c_1$ and $c_2$ such that $c_1 \in \mathtt{M}$, $c_1 \sqsubseteq c_2$, and $c_2$ is reachable in $\mathcal{T}$.

For the *decidability* of this problem, the following **three conditions** are sufficient:

(1) For every two configurations $c_1$ and $c_2$, it is decidable whether $c_1 \sqsubseteq c_2$.
(2) For every $c \in \mathtt{C}$, we can check whether $\{c\}\uparrow \cap \mathtt{Init} \neq \emptyset$.
(3) For every $c \in \mathtt{C}$, the set $\mathsf{minpre}(\{c\})$ is finite and computable.

The solution for the coverability problem as suggested in [ACJT96, FS01] is based on a backward analysis approach. It is shown that starting from a finite set $\mathtt{M}_0 \subseteq \mathtt{C}$, the sequence $(\mathtt{M}_i)_{i\geq 0}$ with $\mathtt{M}_{i+1} := \mathsf{minpre}(\mathtt{M}_i)$, for $i \geq 0$, reaches a fixpoint and it is computable.

4.2. **DTSO-transition System is a Wsts.** In this section, we instantiate the framework of Wsts to show the following result:

**Theorem 4.1** (Decidability of DTSO reachability problem)**.** *The DTSO reachability problem is decidable.*

*Proof.* The rest of this section is devoted to the proof of the above theorem. Let $\mathcal{P} = (A_1, A_2, \ldots, A_n)$ be a concurrent system (as defined in Section 3). Moreover, let $\mathcal{T}_{\mathsf{DTSO}} = \left(\mathsf{C}_{\mathsf{DTSO}}, \{c^D_{init}\}, \Delta \cup \Delta'', \rightarrow_{\mathsf{DTSO}}\right)$ be the transition system induced by $\mathcal{P}$ under the Dual TSO semantics (as defined in Section 3.3).

In the following, we will show that the DTSO-transition system $\mathcal{T}_{\mathsf{DTSO}}$ is monotonic wrt. an ordering $\sqsubseteq$. Then, we will show the three sufficient conditions for the decidability of the coverability problem for $(\mathcal{T}_{\mathsf{DTSO}}, \sqsubseteq)$ (as stated in Section 4.1).

(1) We first define the ordering $\sqsubseteq$ on the set of DTSO-configurations $\mathsf{C}_{\mathsf{DTSO}}$ (see Section 4.2.1).
(2) Then, we show that the transition system $\mathcal{T}_{\mathsf{DTSO}}$ induced under the Dual TSO semantics is monotonic wrt. the ordering $\sqsubseteq$ (see Lemma 4.2).
(3) For the first sufficient condition, we show that $\sqsubseteq$ is a well-quasi-ordering; and that for every two configurations $c_1$ and $c_2$, it is decidable whether $c_1 \sqsubseteq c_2$ (see Lemma 4.3).
(4) The second sufficient condition (i.e., checking whether the upward closed set $\{c\}\uparrow$, with $c$ is a DTSO-configuration, contains the initial configuration $c^D_{init}$) is trivial. This check boils down to verifying whether $c$ is the initial configuration $c^D_{init}$.
(5) For the third sufficient condition, we show that we can calculate the set of minimal DTSO-configurations for the set of predecessors of any upward closed set (see Lemma 4.4).
(6) Finally, we will also show that the DTSO reachability problem for $\mathcal{P}$ can be reduced to the coverability problem in the monotonic transition system $(\mathcal{T}_{\mathsf{DTSO}}, \sqsubseteq)$ (see Lemma 4.5). Observe that this reduction is needed since we are requiring that the load buffers are empty when defining the DTSO reachability problem.

This concludes the proof of Theorem 4.1. □

4.2.1. **Ordering $\sqsubseteq$.** In the following, we define an ordering $\sqsubseteq$ on the set of DTSO-configurations $\mathsf{C}_{\mathsf{DTSO}}$. Let us first introduce some notations and definitions.

Consider a word $w \in ((\mathbb{X} \times \mathbb{V}) \cup (\mathbb{X} \times \mathbb{V} \times \{own\}))^*$ representing the content of a load buffer. We define an operation that divides $w$ into a number of fragments according to the most-recent own-message concerning each variable. We define

$$[w]_{own} := (w_1, (x_1, v_1, own), w_2, \ldots, w_m, (x_m, v_m, own), w_{m+1})$$

where the following conditions are satisfied:

(1) $x_i \neq x_j$ for all $i, j : i \neq j$ and $1 \leq i, j \leq m$.
(2) If $(x, v, own) \in w_i$ for some $i : 1 < i \leq m + 1$, then $x = x_j$ for some $j : 1 \leq j < i$, i.e., the most recent own-message on variable $x_j$ occurs at the $(2j)^{th}$ fragment of $[w]_{own}$.
(3) $w = w_1 \cdot (x_1, v_1, own) \cdot w_2 \cdots w_m \cdot (x_m, v_m, own) \cdot w_{m+1}$, i.e., the divided fragments correspond to the given word $w$.

Let $w, w' \in ((\mathbb{X} \times \mathbb{V}) \cup (\mathbb{X} \times \mathbb{V} \times \{own\}))^*$ be two words. Let us assume that:

$$[w]_{own} = (w_1, (x_1, v_1, own), w_2, \ldots, w_r, (x_r, v_r, own), w_{r+1})$$
$$[w']_{own} = (w'_1, (x'_1, v'_1, own), w'_2, \ldots, w'_m, (x'_m, v'_m, own), w'_{m+1}).$$

We write $w \sqsubseteq w'$ to denote that the following conditions are satisfied:

(1) $r = m$,
(2) $x'_i = x_i$ and $v'_i = v_i$ for all $i : 1 \leq i \leq m$, and
(3) $w_i \preceq w'_i$ for all $i : 1 \leq i \leq m + 1$.

Consider two DTSO-configurations $c = (\mathbf{q}, \mathbf{b}, \mathbf{mem})$ and $c' = (\mathbf{q}', \mathbf{b}', \mathbf{mem}')$, we extend the ordering $\sqsubseteq$ to configurations as follows: We write $c \sqsubseteq c'$ if and only if the following conditions are satisfied:

(1) $\mathbf{q} = \mathbf{q}'$,
(2) $\mathbf{b}(p) \sqsubseteq \mathbf{b}'(p)$ for all process $p \in \mathbb{P}$, and
(3) $\mathbf{mem}' = \mathbf{mem}$.

4.2.2. **Monotonicity.** Let $c_1 = (\mathbf{q}_1, \mathbf{b}_1, \mathbf{mem}_1), c_2 = (\mathbf{q}_2, \mathbf{b}_2, \mathbf{mem}_2), c_3 = (\mathbf{q}_3, \mathbf{b}_3, \mathbf{mem}_3) \in \mathsf{C}_{\mathsf{DTSO}}$ such that $c_1 \xrightarrow{t}_{\mathsf{DTSO}} c_2$ for some $t \in \Delta_p \cup \left\{\mathsf{propagate}_p^x, \mathsf{delete}_p |\ x \in \mathbb{X}\right\}$ with $p \in \mathbb{P}$, and $c_1 \sqsubseteq c_3$. We will show that it is possible to compute a configuration $c_4 \in \mathsf{C}_{\mathsf{DTSO}}$ and a run $\pi$ such that $c_3 \xrightarrow{\pi}_{\mathsf{DTSO}} c_4$ and $c_2 \sqsubseteq c_4$.

To that aim, we first show that it is possible from $c_3$ to reach a configuration $c_3'$, by performing a certain number of $\mathsf{delete}_p$ transitions, such that the process $p$ will have the same last message in its load buffer in the configurations $c_1$ and $c_3'$ while $c_1 \sqsubseteq c_3'$. Then, from the configuration $c_3'$, the process $p$ can perform the same transition $t$ as $c_1$ did (to reach $c_2$) in order to reach the configuration $c_4$ such that $c_2 \sqsubseteq c_4$. Let us assume that $[\mathbf{b}_1(p)]_{own}$ is of the form

$$(w_1, (x_1, v_1, own), w_2, \ldots, w_m, (x_m, v_m, own), w_{m+1})$$

and $[\mathbf{b}_3(p)]_{own}$ is of the form

$$\left(w_1', (x_1', v_1', own), w_2', \ldots, w_m', (x_m', v_m', own), w_{m+1}'\right).$$

We define the word $w \in ((\mathbb{X} \times \mathbb{V}) \cup (\mathbb{X} \times \mathbb{V} \times \{own\}))^*$ to be the longest word such that $w_{m+1}' = w' \cdot w$ with $w_{m+1} \preceq w'$. Observe that in this case we have either $w_{m+1} = w' = \epsilon$ or $w'(|w'|) = w_{m+1}(|w_{m+1}|)$. Then, after executing a certain number $|w|$ of $\mathsf{delete}_p$ transitions from the configuration $c_3$, one can obtain a configuration $c_3' = (\mathbf{q}_3, \mathbf{b}_3', \mathbf{mem}_3)$ such that $\mathbf{b}_3 = \mathbf{b}_3' \left[p \hookleftarrow \mathbf{b}_3'(p) \cdot w\right]$. As a consequence, we have $c_1 \sqsubseteq c_3'$. Furthermore, since $c_1$ and $c_3'$ have the same global state, the same memory valuation, the same sequence of most-recent own-messages concerning each variable, and the same last message in the load buffers of $p$, $c_3'$ can perform the transition $t$ and reaches a configuration $c_4$ such that $c_2 \sqsubseteq c_4$.

The following lemma shows that $(\mathcal{T}_{\mathsf{DTSO}}, \sqsubseteq)$ is a monotonic transition system.

**Lemma 4.2** (DTSO monotonic transition system)**.** *The transition relation $\rightarrow_{\mathsf{DTSO}}$ is monotonic wrt. the ordering $\sqsubseteq$.*

*Proof.* The proof of the lemma is given in Appendix B. □

4.2.3. **Conditions of Decidability.** We show the first and the third conditions of the three conditions for the decidability of the coverability problem for $(\mathcal{T}_{\mathsf{DTSO}}, \sqsubseteq)$ (as stated in Section 4.1). The second condition has been shown to be trivial in the main proof of Theorem 4.1.

The following lemma shows that the ordering $\sqsubseteq$ is indeed a well-quasi-ordering.

**Lemma 4.3** (Well-quasi-ordering $\sqsubseteq$)**.** *The ordering $\sqsubseteq$ is a well-quasi-ordering over $\mathsf{C}_{\mathsf{DTSO}}$. Furthermore, for every two DTSO-configurations $c_1$ and $c_2$, it is decidable whether $c_1 \sqsubseteq c_2$.*

*Proof.* The proof of the lemma is given in Appendix C. □

The following lemma shows that we can calculate the set of minimal DTSO-configurations for the set of predecessors of any upward closed set.

**Lemma 4.4** (Computable minimal predecessor set)**.** *For any DTSO-configuration $c$, we can compute* $\mathsf{minpre}(\{c\})$.

*Proof.* The proof of lemma is given in Appendix D. ☐

4.2.4. **From DTSO Reachability to Coverability.** Let $\mathbf{q}_{target}$ be a global state of a concurrent program $\mathcal{P}$ and let $\mathtt{M}_{target}$ be the set of DTSO-configurations of the form $(\mathbf{q}_{target}, \mathbf{b}, \mathbf{mem})$ with $\mathbf{b}(p) = \epsilon$ for all $p \in \mathbb{P}$ where $\mathbb{P}$ be the set of process IDs in $\mathcal{P}$. We recall that $\mathbf{q}_{target}$ in $\mathcal{T}_{\mathsf{DTSO}}$ if and only if $\mathtt{M}_{target}$ is reachable in $\mathcal{T}_{\mathsf{DTSO}}$ (see Section 2 for the definition of a reachable set of configurations). Then by Lemma 4.5, we have that the reachability problem of $\mathbf{q}_{target}$ in $\mathcal{T}_{\mathsf{DTSO}}$ can be reduced to the coverability problem of $\mathtt{M}_{target}$ in $(\mathcal{T}_{\mathsf{DTSO}}, \sqsubseteq)$.

**Lemma 4.5** (DTSO reachability to coverability)**.** $\mathtt{M}_{target}\!\uparrow$ *is reachable in* $\mathcal{T}_{\mathsf{DTSO}}$ *iff* $\mathtt{M}_{target}$ *is reachable in* $\mathcal{T}_{\mathsf{DTSO}}$.

*Proof.* Let us assume that $\mathtt{M}_{target}\!\uparrow$ is reachable in $\mathcal{T}_{\mathsf{DTSO}}$. This means that there is a configuration $c \in \mathtt{M}_{target}\!\uparrow$ which is reachable in $\mathcal{T}_{\mathsf{DTSO}}$. Let us assume that $c$ is of the form $(\mathbf{q}_{target}, \mathbf{b}, \mathbf{mem})$. Then, from the configuration $c$, it is possible to reach the configuration $c' = (\mathbf{q}_{target}, \mathbf{b}', \mathbf{mem})$, with $\mathbf{b}'(p) = \epsilon$ for all $p \in \mathbb{P}$, by performing a sequence of delete transitions to empty the load buffer of each process. It is then easy to see that $c' \in \mathtt{M}_{target}$ and so $\mathtt{M}_{target}$ is reachable in $\mathcal{T}_{\mathsf{DTSO}}$. The other direction of the lemma is trivial since $\mathtt{M}_{target} \subseteq \mathtt{M}_{target}\!\uparrow$. ☐

## 5. Parameterized Concurrent Systems

In this section, we give the definitions for *parameterized concurrent systems*, a model for representing unbounded number of communicating concurrent processes under the Dual TSO semantics, and its induced transition system. Then, we define the DTSO reachability problem for the case of parameterized concurrent systems.

5.1. **Definitions for Parameterized Concurrent Systems.** Let $\mathbb{V}$ be a finite data domain and $\mathbb{X}$ be a finite set of variables ranging over $\mathbb{V}$. A *parameterized concurrent system* (or simply a *parameterized system*) consists of an unbounded number of *identical* processes running under the Dual TSO semantics. Communication between processes is performed through a shared memory that consists of a finite number of the shared variables $\mathbb{X}$ over the finite domain $\mathbb{V}$. Formally, a parameterized system $\mathcal{S}$ is defined by an extended finite-state automaton $A = \left(Q, q^{init}, \Delta\right)$ uniformly describing the behavior of each process.

An *instance* of $\mathcal{S}$ is a concurrent system $\mathcal{P} = (A_1, A_2, \ldots, A_n)$, for some $n \in \mathbb{N}$, where for each $p : 1 \leq p \leq n$, we have $A_p = A$. In other words, it consists of a finite set of processes each running the same code defined by $A$. We use $\mathit{Inst}(\mathcal{S})$ to denote all possible instances of $\mathcal{S}$. We use $\mathcal{T}_{\mathcal{P}} := (\mathtt{C}_{\mathcal{P}}, \mathtt{Init}_{\mathcal{P}}, \mathtt{Act}_{\mathcal{P}}, \rightarrow_{\mathcal{P}})$ to denote the transition system induced by an instance $\mathcal{P}$ of $\mathcal{S}$ under the Dual TSO semantics.

A *parameterized configuration* $\alpha$ is a pair $(\mathbb{P}, c)$ where $\mathbb{P} = \{1, \ldots, n\}$, with $n \in \mathbb{N}$, is the set of process IDs and $c$ is a DTSO-configuration of an instance $\mathcal{P} = (A_1, A_2, \ldots, A_n)$ of $\mathcal{S}$.

The parameterized configuration $\alpha = (\mathbb{P}, c)$ is said to be *initial* if $c$ is an initial configuration of $\mathcal{P}$ (i.e., $c \in \mathtt{Init}_{\mathcal{P}}$). We use $\mathtt{C}$ (resp. $\mathtt{Init}$) to denote the set of all the parameterized configurations (resp. all the initial configurations) of $\mathcal{S}$.

Let $\mathtt{Act}$ denote the set of *actions* of all possible instances of $\mathcal{S}$ (i.e., $\mathtt{Act} = \cup_{\mathcal{P} \in Inst(\mathcal{S})} \mathtt{Act}_{\mathcal{P}}$). We define a *transition relation* $\rightarrow$ on the set $\mathtt{C}$ of all parameterized configurations such that given two configurations $(\mathbb{P}, c)$ and $(\mathbb{P}', c')$, we have $(\mathbb{P}, c) \xrightarrow{t} (\mathbb{P}', c')$ for some action $t \in \mathtt{Act}$ iff $\mathbb{P}' = \mathbb{P}$ and there is an instance $\mathcal{P}$ of $\mathcal{S}$ such that $t \in \mathtt{Act}_{\mathcal{P}}$ and $c \xrightarrow{t}_{\mathcal{P}} c'$. The *transition system* induced by $\mathcal{S}$ is given by $\mathcal{T} := (\mathtt{C}, \mathtt{Init}, \mathtt{Act}, \rightarrow)$.

**The Parameterized DTSO Reachability Problem.** A global state $\mathbf{q}_{target} : \mathbb{P}' \mapsto Q$ is said to be *reachable* in $\mathcal{T}$ if and only if there exists a parameterized configuration $\alpha = (\mathbb{P}, (\mathbf{q}, \mathbf{b}, \mathbf{mem}))$, with $\mathbf{b}(p) = \epsilon$ for all $p \in \mathbb{P}$, such that $\alpha$ is reachable in $\mathcal{T}$ and $\mathbf{q}_{target}(1) \cdots \mathbf{q}_{target}(|\mathbb{P}'|) \preceq \mathbf{q}(1) \cdots \mathbf{q}(|\mathbb{P}|)$.

The parameterized DTSO reachability problem consists in checking whether $\mathbf{q}_{target}$ is reachable in $\mathcal{T}$. In other words, the DTSO reachability problem for parameterized systems asks whether there is an instance of the parameterized system that reaches a configuration with a number of processes in certain given local states.

5.2. **Decidability of the Parameterized Reachability Problem.** We prove hereafter the following theorem:

**Theorem 5.1** (Decidability of parameterized DTSO reachability problem)**.** *The parameterized DTSO reachability problem is decidable.*

*Proof.* Let $\mathcal{S} = \left(Q, q^{init}, \Delta\right)$ be a parameterized system and $\mathcal{T} = (\mathtt{C}, \mathtt{Init}, \mathtt{Act}, \rightarrow)$ be its induced transition system. The proof of the theorem is done by instantiating the framework of WSTS. In more detail, we will show that the parameterized transition system $\mathcal{T}$ is monotonic wrt. an ordering $\trianglelefteq$. Then, we will show the three sufficient conditions for the decidability of the coverability problem for $(\mathcal{T}, \trianglelefteq)$ (as stated in Section 4.1).

(1) We first define the ordering $\trianglelefteq$ on the set $\mathtt{C}$ of all parameterized configurations (see Section 5.2.1).
(2) Then, we show that the transition system $\mathcal{T}$ is monotonic wrt. the ordering $\trianglelefteq$ (see Lemma 5.2).
(3) For the first sufficient condition, we show that the ordering $\trianglelefteq$ is a well-quasi-ordering; and that for every two parameterized configurations $\alpha$ and $\alpha'$, it is decidable whether $\alpha \sqsubseteq \alpha'$ (see Lemma 5.3).
(4) The second sufficient condition (i.e., checking whether the upward closed set $\{\alpha\}\uparrow$, with $\alpha$ is a parameterized configuration, contains an initial configuration) for the decidability of the coverability problem is trivial. This check boils down to verifying whether the configuration $\alpha$ is initial.
(5) For the third sufficient condition, we show that we can calculate the set of minimal parameterized configurations for the set of predecessors of any upward closed set (see Lemma 5.4).
(6) Finally, we will show that the parameterized DTSO reachability problem for the parameterized system $\mathcal{S}$ can be reduced to the coverability problem in the monotonic transition system $(\mathcal{T}, \trianglelefteq)$ (see Lemma 5.5).

This concludes the proof of Theorem 5.1. □

5.2.1. **Ordering $\trianglelefteq$.** Let $\alpha = (\mathbb{P}, (\mathbf{q}, \mathbf{b}, \mathbf{mem}))$ and $\alpha' = (\mathbb{P}', (\mathbf{q}', \mathbf{b}', \mathbf{mem}'))$ be two parameterized configurations. We define the ordering $\trianglelefteq$ on the set $\mathtt{C}$ of parameterized configurations as follows: We write $\alpha \trianglelefteq \alpha'$ if and only if the following conditions are satisfied:

(1) $\mathbf{mem} = \mathbf{mem}'$.
(2) There is an injection $h : \{1, \ldots, |\mathbb{P}|\} \mapsto \{1, \ldots, |\mathbb{P}'|\}$ such that
    (i) for all $p, p' \in \mathbb{P}$, $p < p'$ implies $h(p) < h(p')$; and
    (ii) for every $p \in \{1, \ldots, |\mathbb{P}|\}$, $\mathbf{q}(p) = \mathbf{q}'(h(p))$ and $\mathbf{b}(p) \sqsubseteq \mathbf{b}'(h(p))$.

5.2.2. **Monotonicity.** We assume that three configurations $\alpha_1 = (\mathbb{P}, (\mathbf{q}_1, \mathbf{b}_1, \mathbf{mem}_1))$, $\alpha_2 = (\mathbb{P}, (\mathbf{q}_2, \mathbf{b}_2, \mathbf{mem}_2))$ and $\alpha_3 = (\mathbb{P}', (\mathbf{q}_3, \mathbf{b}_3, \mathbf{mem}_3))$ are given. Furthermore, we assume that $\alpha_1 \trianglelefteq \alpha_3$ and $\alpha_1 \xrightarrow{t} \alpha_2$ for some transition $t$. We will show that it is possible to compute a parameterized configuration $\alpha_4$ and a run $\pi$ such that $\alpha_3 \xrightarrow{\pi} \alpha_4$ and $\alpha_2 \trianglelefteq \alpha_4$.

Since $\alpha_1 \trianglelefteq \alpha_3$, there is an injection function $h : \{1, \ldots, |\mathbb{P}|\} \mapsto \{1, \ldots, |\mathbb{P}'|\}$ such that:

(1) For all $p, p' \in \mathbb{P}$, $p < p'$ implies $h(p) < h(p')$.
(2) For every $p \in \{1, \ldots, |\mathbb{P}|\}$, $\mathbf{q}_1(p) = \mathbf{q}_3(h(p))$ and $\mathbf{b}_1(p) \sqsubseteq \mathbf{b}_3(h(p))$.

We define the parameterized configuration $\alpha'$ from $\alpha_3$ by only keeping the local states and load buffers of processes in $h(\mathbb{P})$. Formally, $\alpha' = (\mathbb{P}, (\mathbf{q}', \mathbf{b}', \mathbf{mem}'))$ is defined as follows:

(1) $\mathbf{mem}' = \mathbf{mem}_3$.
(2) For every $p \in \{1, \ldots, |\mathbb{P}|\}$, $\mathbf{q}'(p) = \mathbf{q}_3(h(p))$ and $\mathbf{b}'(p) = \mathbf{b}_3(h(p))$.

We observe that $(\mathbf{q}_1, \mathbf{b}_1, \mathbf{mem}_1) \sqsubseteq (\mathbf{q}', \mathbf{b}', \mathbf{mem}')$. Since the transition relation $\rightarrow_{\mathsf{DTSO}}$ is monotonic wrt. the ordering $\sqsubseteq$ (see Lemma 4.2), there is a DTSO-configuration $(\mathbf{q}'', \mathbf{b}'', \mathbf{mem}'')$ such that $(\mathbf{q}', \mathbf{b}', \mathbf{mem}') \rightarrow^*_{\mathsf{DTSO}} (\mathbf{q}'', \mathbf{b}'', \mathbf{mem}'')$ and $(\mathbf{q}_2, \mathbf{b}_2, \mathbf{mem}_2) \sqsubseteq (\mathbf{q}'', \mathbf{b}'', \mathbf{mem}'')$.

Consider now the parameterized configuration $\alpha_4 = (\mathbb{P}', (\mathbf{q}_4, \mathbf{b}_4, \mathbf{mem}_4))$ such that:

(1) $\mathbf{mem}'' = \mathbf{mem}_4$.
(2) For every $p \in \{1, \ldots, |\mathbb{P}|\}$, $\mathbf{q}''(p) = \mathbf{q}_4(h(p))$ and $\mathbf{b}''(p) = \mathbf{b}_4(h(p))$.
(3) For every $p \in (\{1, \ldots, |\mathbb{P}'|\} \setminus \{h(1), \ldots, h(|\mathbb{P}|)\})$, we have $\mathbf{q}_4(p) = \mathbf{q}_3(p)$ and $\mathbf{b}_4(p) = \mathbf{b}_3(p)$.

It is easy then to see that $\alpha_2 \trianglelefteq \alpha_4$ and $\alpha_3 \rightarrow^* \alpha_4$.

The following lemma shows that $(\mathcal{T}, \trianglelefteq)$ is a monotonic transition system.

**Lemma 5.2** (Parameterized monotonic transition system)**.** *The transition relation $\rightarrow$ is monotonic wrt. the ordering $\trianglelefteq$.*

*Proof.* The proof of the lemma is given in Appendix E. □

5.2.3. **Conditions for Decidability.** We show the first and the third conditions of the three conditions for the decidability of the coverability problem for $(\mathcal{T}, \trianglelefteq)$ (as stated in Section 4.1). The second condition has been shown to be trivial in the main proof of Theorem 5.1.

The following lemma states that the ordering $\trianglelefteq$ is indeed a well-quasi-ordering:

**Lemma 5.3** (Parameterized well-quasi-ordering $\trianglelefteq$)**.** *The ordering $\trianglelefteq$ is a well-quasi-ordering over $\mathtt{C}$. Furthermore, for every two parameterized configurations $\alpha$ and $\alpha'$, it is decidable whether $\alpha \trianglelefteq \alpha'$.*

*Proof.* The lemma follows a similar argument as in the proof of Lemma 4.3. □

The following lemma shows that we can calculate the set of minimal parameterized configurations for the set of predecessors of any upward closed set.

**Lemma 5.4** (Computable minimal parameterized predecessor set)**.** *For any parameterized configuration* $\alpha$*, we can compute* $\mathsf{minpre}(\{\alpha\})$*.*

*Proof.* The proof of the lemma is given in Appendix F. □

5.2.4. **From Parameterized DTSO Reachability to Coverability.** Let $\mathbf{q}_{target} : \mathbb{P}' \mapsto Q$ be a global state. Let $\mathtt{M}_{target}$ be the set of parameterized configurations of the form $\alpha = (\mathbb{P}', (\mathbf{q}_{target}, \mathbf{b}, \mathbf{mem}))$ with $\mathbf{b}(p) = \epsilon$ for all $p \in \mathbb{P}'$. Lemma 5.5 shows that the parameterized reachability problem of $\mathbf{q}_{target}$ in the transition system $\mathcal{T}$ can be reduced to the coverability problem of $\mathtt{M}_{target}$ in $(\mathcal{T}, \trianglelefteq)$.

**Lemma 5.5** (Parameterized DTSO reachability to coverability)**.** $\mathbf{q}_{target}$ *is reachable in* $\mathcal{T}$ *iff* $\mathtt{M}_{target}\uparrow$ *is reachable in* $\mathcal{T}$*.*

*Proof.* To prove the lemma, we first show that $\mathtt{M}_{target}\uparrow$ is reachable in $\mathcal{T}$ if and only if there is a parameterized configuration $\alpha = (\mathbb{P}, (\mathbf{q}, \mathbf{b}, \mathbf{mem}))$, with $\mathbf{b}(p) = \epsilon$ for all $p \in \mathbb{P}$, such that $\alpha$ is reachable in $\mathcal{T}$ and $\mathbf{q}_{target}(1) \cdots \mathbf{q}_{target}(|\mathbb{P}'|) \preceq \mathbf{q}(1) \cdots \mathbf{q}(|\mathbb{P}|)$. Then as a consequence, the lemma holds.

Let us assume that there is a parameterized configuration $\alpha = (\mathbb{P}, (\mathbf{q}, \mathbf{b}, \mathbf{mem}))$, with $\mathbf{b}(p) = \epsilon$ for all $p \in \mathbb{P}$, such that $\alpha$ is reachable in $\mathcal{T}$ and $\mathbf{q}_{target}(1) \cdots \mathbf{q}_{target}(|\mathbb{P}'|) \preceq \mathbf{q}(1) \cdots \mathbf{q}(|\mathbb{P}|)$. It is then easy to show that $\alpha \in \mathtt{M}_{target}\uparrow$.

Now let us assume that there is a parameterized configuration $\alpha' = (\mathbb{P}'', (\mathbf{q}', \mathbf{b}', \mathbf{mem}')) \in \mathtt{M}_{target}\uparrow$ which is reachable in $\mathcal{T}$. From the configuration $\alpha'$, it is possible to reach the configuration $\alpha'' = (\mathbb{P}'', (\mathbf{q}', \mathbf{b}'', \mathbf{mem}'))$, with $\mathbf{b}''(p) = \epsilon$ for all $p \in \mathbb{P}''$, by performing a sequence of $\mathsf{delete}_p$ transitions to empty the load buffer of each process. Since $\alpha' \in \mathtt{M}_{target}\uparrow$, we have $\mathbf{q}_{target}(1) \cdots \mathbf{q}_{target}(|\mathbb{P}'|) \preceq \mathbf{q}'(1) \cdots \mathbf{q}'(|\mathbb{P}''|)$. Hence, $\alpha''$ is a witness of the parameterized reachability problem of $\mathbf{q}_{target}$ in the transition system $\mathcal{T}$. □

## 6. Experimental Results

We have implemented our techniques described in Section 4 and Section 5 in an open-source tool called Dual-TSO[1]. The tool checks the state reachability problems (c.f. Section 3.3 and Section 5.1) for (parameterized) concurrent systems under the Dual TSO semantics. We emphasize that besides checking the reachability for a global state, Dual-TSO can check the reachability for a set of global states. Moreover, Dual-TSO accepts a more general input class of parameterized concurrent systems. Instead of requiring that the behavior of each process is described by a *unique* extended finite-state automaton as defined in Section 5, Dual-TSO allows that the behavior of a process can be presented by an extended finite-state automaton from a *fixed* set of predefined automata. If the tool finds a witness for a given reachability problem, we say that the concurrent system is unsafe (wrt. the reachability problem). After finding the first witness for a given reachability problem, the tool terminates its execution. In the case that no witness is encountered, Dual-TSO declares that the given concurrent program is safe (wrt. the reachability problem) after it reaches a fixpoint in

[1] Tool webpage: https://www.it.uu.se/katalog/tuang296/dual-tso

| **Program** | **#P** | **Safe under** | | Dual-TSO($\sqsubseteq$) | | Memorax | |
|---|---|---|---|---|---|---|---|
| | | **SC** | **TSO** | **#T(s)** | **#C** | **#T(s)** | **#C** |
| SB | 5 | yes | no | 0.3 | 10 641 | 559.7 | 10 515 914 |
| LB | 3 | yes | yes | 0.0 | 2 048 | 71.4 | 1 499 475 |
| WRC | 4 | yes | yes | 0.0 | 1 507 | 63.3 | 1 398 393 |
| ISA2 | 3 | yes | yes | 0.0 | 509 | 21.1 | 226 519 |
| RWC | 5 | yes | no | 0.1 | 4 277 | 61.5 | 1 196 988 |
| W+RWC | 4 | yes | no | 0.0 | 1 713 | 83.6 | 1 389 009 |
| IRIW | 4 | yes | yes | 0.0 | 520 | 34.4 | 358 057 |
| MP | 4 | yes | yes | 0.0 | 883 | $t/o$ | • |
| Simple Dekker | 2 | yes | no | 0.0 | 98 | 0.0 | 595 |
| Dekker | 2 | yes | no | 0.1 | 5 053 | 1.1 | 19 788 |
| Peterson | 2 | yes | no | 0.1 | 5 442 | 5.2 | 90 301 |
| Repeated Peterson | 2 | yes | no | 0.2 | 7 632 | 5.6 | 100 082 |
| Bakery | 2 | yes | no | 2.6 | 82 050 | $t/o$ | • |
| Dijkstra | 2 | yes | no | 0.2 | 8 324 | $t/o$ | • |
| Szymanski | 2 | yes | no | 0.6 | 29 018 | 1.0 | 26 003 |
| Ticket Spin Lock | 3 | yes | yes | 0.9 | 18 963 | $t/o$ | • |
| Lamport's Fast Mutex | 3 | yes | no | 17.7 | 292 543 | $t/o$ | • |
| Burns | 4 | yes | no | 124.3 | 2 762 578 | $t/o$ | • |
| NBW-W-WR | 2 | yes | yes | 0.0 | 222 | 10.7 | 200 844 |
| Sense Reversing Barrier | 2 | yes | yes | 0.1 | 1 704 | 0.8 | 20 577 |

Table 1: Comparison between Dual-TSO($\sqsubseteq$) and Memorax: The columns *Safe under SC* and *Safe under TSO* indicate that whether the benchmark is safe under SC and TSO wrt. its reachability problem respectively. The columns $\#P$, $\#T$ and $\#C$ give the number of processes, the running time in seconds and the number of generated configurations, respectively. If a tool runs out of time, we put $t/o$ in the $\#T$ column and • in the $\#C$ column.

calculation. Dual-TSO always ends its execution by reporting the running time (in seconds) and the total number of generated configurations. Observe that the number of generated configurations gives a rough estimation of the memory consumption of our tool.

We compare our tool with Memorax [AAC+12a, AAC+13] which is the *only precise and sound tool* for deciding the state reachability problem of concurrent systems running under TSO. Observe that Memorax cannot handle the class of parameterized concurrent systems. We use Dual-TSO($\sqsubseteq$) and Dual-TSO($\trianglelefteq$) to denote Dual-TSO when applied to concurrent systems and parameterized concurrent systems, respectively.

In the following, we present two sets of results. The first set concerns the comparison of Dual-TSO($\sqsubseteq$) with Memorax (see Table 1). The second set shows the benefit of the parameterized verification compared to the use of the state reachability when increasing the number of processes (see Table 2 and Figure 8). Our example programs are from [AAC+12a, AMT14, BDM13, AAP15, LNP+12]. In all experiments, we set up the time out to 600 seconds (10 minutes). We perform all experiments on an Intel x86-32 Core2 2.4 Ghz machine and 4GB of RAM.

**Verification of Concurrent Systems.** Table 1 presents a comparison between Dual-TSO($\sqsubseteq$) and Memorax on 20 benchmarks. In all these benchmarks, Dual-TSO($\sqsubseteq$) and

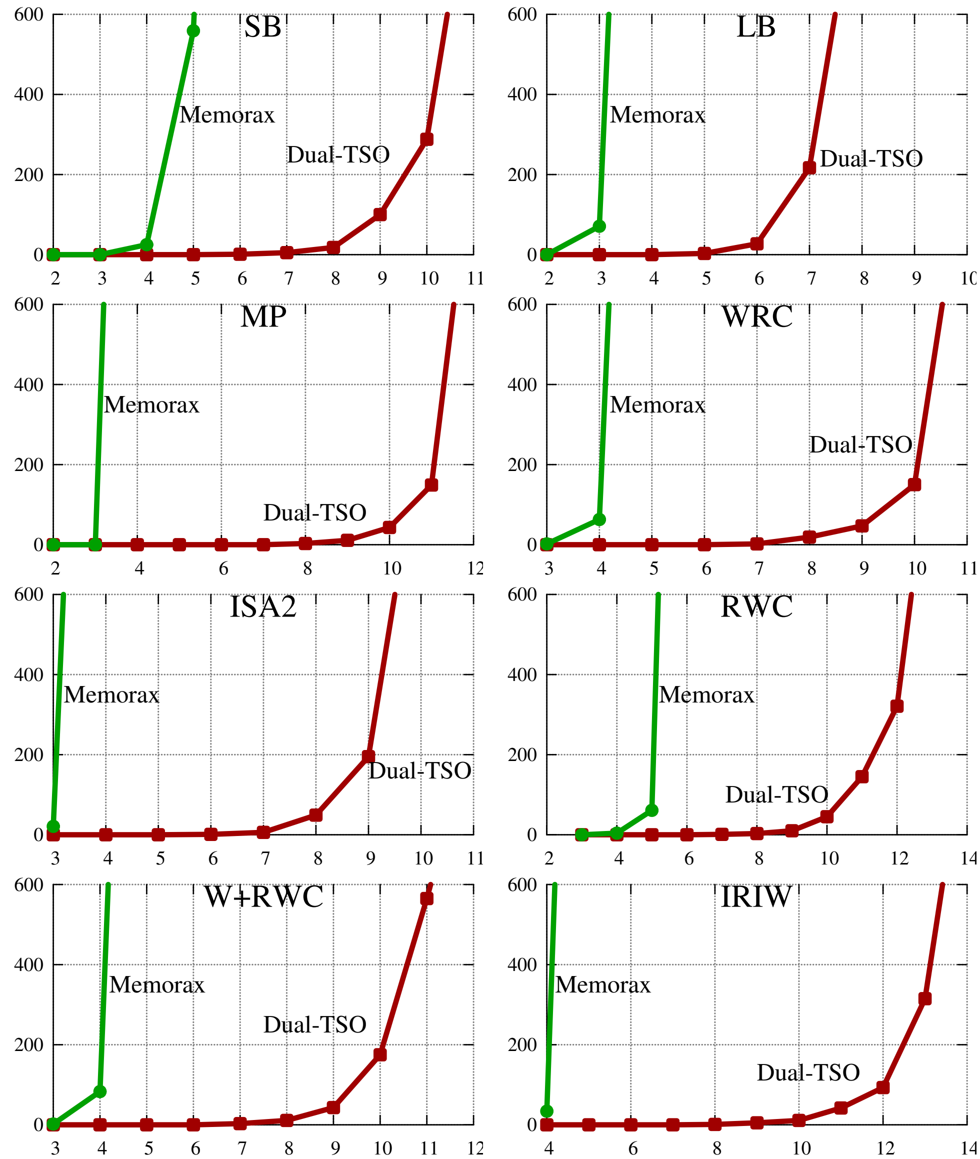


Figure 8: Running time of Memorax and Dual-TSO($\sqsubseteq$) by increasing number of processes. The x axis is number of processes and the y axis is running time in seconds.

Memorax return the same results for the state reachability problems (except 6 examples where Memorax runs out of time). In the benchmarks where the two tools return, Dual-TSO($\sqsubseteq$) out-performs Memorax and generates fewer configurations (and so uses less memory). Indeed, Dual-TSO($\sqsubseteq$) is 600 times faster than Memorax and generates 277 times fewer minimal configurations on average. The experimental results confirm the correlation between the running time and the memory consumption (i.e., the tool who generates less configurations is often the fastest).

**Verification of Parameterized Concurrent Systems.** The second set compares the scalability of Memorax and Dual-TSO while increasing the number of processes. The results

| **Program** | **Safe under TSO** | Dual-TSO($\trianglelefteq$) #**T**(**s**) | #**C** |
|---|---|---|---|
| SB | no | 0.0 | 147 |
| LB | yes | 0.6 | 1 028 |
| MP | yes | 0.0 | 149 |
| WRC | yes | 0.8 | 618 |
| ISA2 | yes | 4.3 | 1 539 |
| RWC | no | 0.2 | 293 |
| W+RWC | no | 1.5 | 828 |
| IRIW | yes | 4.6 | 648 |

Table 2: Parameterized verification with Dual-TSO($\trianglelefteq$).

are given in Figure 8. We observe that although the algorithms implemented by Dual-TSO($\sqsubseteq$) and Memorax have the same (non-primitive recursive) lower bound (in theory), Dual-TSO($\sqsubseteq$) scales better than Memorax in all these benchmarks. In fact, Memorax can only handle benchmarks with at most 5 processes while Dual-TSO can handle benchmarks with more processes. We conjecture that this is due to the important advantages of the Dual TSO semantics. In fact, the Dual TSO semantics transforms the load buffers into lossy channels without adding the costly overhead of memory snapshots that was necessary in the case of Memorax. The absence of this extra overhead means that our tool generates less configurations (due to the ordering) and this results in a better performance and scalability.

Table 2 presents the running time and the number of generated configurations when checking the state reachability problem for the parameterized versions of the benchmarks in Figure 8 with Dual-TSO($\trianglelefteq$). It should be emphasized that Dual-TSO($\trianglelefteq$) and Dual-TSO($\sqsubseteq$) have the same results for the reachability problems in these benchmarks. We observe that the verification of these parameterized systems is much more efficient than verification of bounded-size instances (starting from a number of processes of 3 or 4), especially concerning memory consumption (which is given in terms of number of generated configurations). The reason behind is that the size of the generated minor sets in the analysis of a parameterized system are usually smaller than the size of the generated minor sets during the analysis of an instance of the system with a large number of processes. In fact, during the analysis of a parameterised concurrent system, the number of considered processes in the generated minimal configurations is usually very small. Observe that, in the case of concurrent systems, the number of considered processes in the generated minimal configurations is equal to the number of processes in the given system.

## 7. Conclusion

In this paper, we have presented an alternative (yet equivalent) semantics to the classical one for the TSO memory model that is more amenable for efficient algorithmic verification and for extension to parametric verification. This new semantics allows us to understand the TSO memory model in a totally different way compared to the classical semantics. Furthermore, the proposed semantics offers several important advantages from the point of view of formal reasoning and program verification. First, the Dual TSO semantics allows transforming the load buffers to *lossy* channels (in the sense that the processes can lose any message situated at the head of any load buffer in non-deterministic manner) without adding the costly overhead that was necessary in the case of store buffers. This means that we can apply the theory of *well-structured systems* [Abd10, ACJT96, FS01] in a straightforward manner leading to a much simpler proof of decidability of safety properties. Second, the absence of extra overhead means that we obtain more efficient algorithms and better scalability (as shown by our experimental results). Finally, the Dual TSO semantics allows extending the framework to perform *parameterized verification* which is an important paradigm in concurrent program verification.

In the future, we plan to apply our techniques to other memory models and to combine with predicate abstraction for handling programs with unbounded data domain.

## References


[AAA+15] P. Abdulla, S. Aronis, M.F. Atig, B. Jonsson, C. Leonardsson, and K. Sagonas. Stateless model checking for TSO and PSO. In *TACAS*, volume 9035 of *LNCS*, pages 353–367. Springer, 2015.

[AABN16] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Ahmed Bouajjani, and Tuan Phong Ngo. The benefits of duality in verifying concurrent programs under TSO. In *CONCUR*, volume 59 of *LIPIcs*, pages 5:1–5:15. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016.

[AABN17] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Ahmed Bouajjani, and Tuan Phong Ngo. Context-bounded analysis for POWER. In *TACAS 2017*, pages 56–74, 2017.

[AAC+12a] P.A. Abdulla, M.F. Atig, Y.F. Chen, C. Leonardsson, and A. Rezine. Counter-example guided fence insertion under TSO. In *TACAS 2012*, pages 204–219, 2012.

[AAC+12b] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Carl Leonardsson, and Ahmed Rezine. Automatic fence insertion in integer programs via predicate abstraction. In *SAS 2012*, pages 164–180, 2012.

[AAC+13] P.A. Abdulla, M.F. Atig, Y.F. Chen, C. Leonardsson, and A. Rezine. Memorax, a precise and sound tool for automatic fence insertion under TSO. In *TACAS*, pages 530–536, 2013.

[AAJL16] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, and Carl Leonardsson. Stateless model checking for POWER. In *CAV*, volume 9780 of *LNCS*, pages 134–156. Springer, 2016.

[AALN15] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Magnus Lång, and Tuan Phong Ngo. Precise and sound automatic fence insertion procedure under PSO. In *NETYS 2015*, pages 32–47, 2015.

[AAP15] P.A. Abdulla, M.F. Atig, and N.T. Phong. The best of both worlds: Trading efficiency and optimality in fence insertion for TSO. In *ESOP 2015*, pages 308–332, 2015.

[ABBM10] M. F. Atig, A. Bouajjani, S. Burckhardt, and M. Musuvathi. On the verification problem for weak memory models. In *POPL*, 2010.

[ABBM12] M.F. Atig, A. Bouajjani, S. Burckhardt, and M. Musuvathi. What's decidable about weak memory models? In *ESOP*, volume 7211 of *LNCS*, pages 26–46. Springer, 2012.

[Abd10] Parosh Aziz Abdulla. Well (and better) quasi-ordered transition systems. *Bulletin of Symbolic Logic*, 16(4):457–515, 2010.

[ABP11] M.F. Atig, A. Bouajjani, and G. Parlato. Getting rid of store-buffers in TSO analysis. In *CAV*, volume 6806 of *LNCS*, pages 99–115. Springer, 2011.

[ACJT96] P.A. Abdulla, K. Cerans, B. Jonsson, and Y.K. Tsay. General decidability theorems for infinite-state systems. In *LICS'96*, pages 313–321. IEEE Computer Society, 1996.

[AG96] S. Adve and K. Gharachorloo. Shared memory consistency models: a tutorial. *Computer*, 29(12), 1996.

[AH90] S. Adve and M. D. Hill. Weak ordering - a new definition. In *ISCA*, 1990.

[AKNT13] J. Alglave, D. Kroening, V. Nimal, and M. Tautschnig. Software verification for weak memory via program transformation. In *ESOP*, volume 7792 of *LNCS*, pages 512–532. Springer, 2013.

[AKT13] J. Alglave, D. Kroening, and M. Tautschnig. Partial orders for efficient bounded model checking of concurrent software. In *CAV*, volume 8044 of *LNCS*, pages 141–157, 2013.

[AMT14] Jade Alglave, Luc Maranget, and Michael Tautschnig. Herding cats: Modelling, simulation, testing, and data mining for weak memory. *ACM TOPLAS*, 36(2):7:1–7:74, 2014.

[BAM07] S. Burckhardt, R. Alur, and M. M. K. Martin. CheckFence: checking consistency of concurrent data types on relaxed memory models. In *PLDI*, pages 12–21. ACM, 2007.

[BDM13] Ahmed Bouajjani, Egor Derevenetc, and Roland Meyer. Checking and enforcing robustness against TSO. In *ESOP*, volume 7792 of *LNCS*, pages 533–553. Springer, 2013.

[BM08] Sebastian Burckhardt and Madanlal Musuvathi. Effective program verification for relaxed memory models. In *CAV*, volume 5123 of *LNCS*, pages 107–120. Springer, 2008.

[BSS11] Jacob Burnim, Koushik Sen, and Christos Stergiou. Testing concurrent programs on relaxed memory models. In *ISSTA*, pages 122–132. ACM, 2011.

[DL15] Brian Demsky and Patrick Lam. Satcheck: Sat-directed stateless model checking for SC and TSO. In *OOPSLA 2015*, pages 20–36. ACM, 2015.

[DM14] Egor Derevenetc and Roland Meyer. Robustness against Power is PSpace-complete. In *ICALP (2)*, volume 8573 of *LNCS*, pages 158–170. Springer, 2014.

[DMVY13] A. Marian Dan, Y. Meshman, M. T. Vechev, and E. Yahav. Predicate abstraction for relaxed memory models. In *SAS*, volume 7935 of *LNCS*, pages 84–104. Springer, 2013.

[DMVY17] Andrei Dan, Yuri Meshman, Martin Vechev, and Eran Yahav. Effective abstractions for verification under relaxed memory models. *Computer Languages, Systems and Structures*, 47, Part 1:62–76, 2017.

[DSB86] M. Dubois, C. Scheurich, and F. A. Briggs. Memory access buffering in multiprocessors. In *ISCA*, 1986.

[FS01] A. Finkel and Ph. Schnoebelen. Well-structured transition systems everywhere! *Theor. Comput. Sci.*, 256(1-2):63–92, 2001.

[HH16] Shiyou Huang and Jeff Huang. Maximal causality reduction for TSO and PSO. In *OOPSLA 2016*, pages 447–461, 2016.

[Hig52] G. Higman. Ordering by divisibility in abstract algebras. *Proc. London Math. Soc. (3)*, 2(7):326–336, 1952.

[HVQF16] Mengda He, Viktor Vafeiadis, Shengchao Qin, and João F. Ferreira. Reasoning about fences and relaxed atomics. In *24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2016, Heraklion, Crete, Greece, February 17-19, 2016*, pages 520–527, 2016.

[KVY10] Michael Kuperstein, Martin T. Vechev, and Eran Yahav. Automatic inference of memory fences. In *FMCAD*, pages 111–119. IEEE, 2010.

[KVY11] Michael Kuperstein, Martin T. Vechev, and Eran Yahav. Partial-coherence abstractions for relaxed memory models. In *PLDI*, pages 187–198. ACM, 2011.

[Lam79] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comp.*, C-28(9), 1979.

[LNP+12] Feng Liu, Nayden Nedev, Nedyalko Prisadnikov, Martin T. Vechev, and Eran Yahav. Dynamic synthesis for relaxed memory models. In *PLDI '12*, pages 429–440, 2012.

[LV15] Ori Lahav and Viktor Vafeiadis. Owicki-gries reasoning for weak memory models. In *Automata, Languages, and Programming - 42nd International Colloquium, ICALP 2015, Kyoto, Japan, July 6-10, 2015, Proceedings, Part II*, pages 311–323, 2015.

[LV16] Ori Lahav and Viktor Vafeiadis. Explaining relaxed memory models with program transformations. In *FM 2016*, pages 479–495, 2016.

[OSS09] S. Owens, S. Sarkar, and P. Sewell. A better x86 memory model: x86-tso. In *TPHOL*, 2009.

[SSO+10] P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen. x86-tso: A rigorous and usable programmer's model for x86 multiprocessors. *CACM*, 53, 2010.

[TLF+16] Ermenegildo Tomasco, Truc Nguyen Lam, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. Embedding weak memory models within eager sequentialization. October 2016.

[TLI+16] Ermenegildo Tomasco, Truc Nguyen Lam, Omar Inverso, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. Lazy sequentialization for tso and pso via shared memory abstractions. In *FMCAD16*, pages 193–200, 2016.

[TW16] Oleg Travkin and Heike Wehrheim. Verification of concurrent programs on weak memory models. In *ICTAC 2016*, pages 3–24, 2016.

[Vaf15] Viktor Vafeiadis. Separation logic for weak memory models. In *Proceedings of the Programming Languages Mentoring Workshop, PLMW@POPL 2015, Mumbai, India, January 14, 2015*, page 11:1, 2015.

[WG94] D. Weaver and T. Germond, editors. *The SPARC Architecture Manual Version 9*. PTR Prentice Hall, 1994.
[YGLS04] Y. Yang, G. Gopalakrishnan, G. Lindstrom, and K. Slind. Nemos: A framework for axiomatic and executable specifications of memory consistency models. In *IPDPS*. IEEE, 2004.
[ZKW15] N. Zhang, M. Kusano, and C. Wang. Dynamic partial order reduction for relaxed memory models. In *PLDI*, pages 250–259. ACM, 2015.

## Appendix A. Proof of Theorem 3.4

We prove the theorem by showing its *if direction* and then *only if direction*. In the following, for a TSO (DTSO)-configuration $c = (\mathbf{q}, \mathbf{b}, \mathbf{mem})$, we use $\texttt{states}\,(c)$, $\texttt{buffers}\,(c)$, and $\texttt{mem}\,(c)$ to denote $\mathbf{q}$, $\mathbf{b}$, and $\mathbf{mem}$ respectively.

A.1. **From Dual TSO to TSO.** We show the *if direction* of Theorem 3.4. Consider a DTSO-computation

$$\pi_{\mathsf{DTSO}} = c_0 \xrightarrow{t_1}_{\mathsf{DTSO}} c_1 \xrightarrow{t_2}_{\mathsf{DTSO}} c_2 \cdots \xrightarrow{t_{n-1}}_{\mathsf{DTSO}} c_{n-1} \xrightarrow{t_n}_{\mathsf{DTSO}} c_n$$

where $c_0 = c^D_{init}$ and $c_i$ is of the form $(\mathbf{q}_i, \mathbf{b}_i, \mathbf{mem}_i)$ for all $i : 1 \leq i \leq n$ with $\mathbf{q}_n = \mathbf{q}_{target}$ and $\mathbf{b}_n(p) = \epsilon$ for all $p \in \mathbb{P}$. We will derive a TSO-computation $\pi_{\mathsf{TSO}}$ such that $target\,(\pi_{\mathsf{TSO}})$ is a configuration of the form $(\texttt{states}\,(c_n)\,, \mathbf{b}, \texttt{mem}\,(c_n))$ where $\mathbf{b}(p) = \epsilon$ for all $p \in \mathbb{P}$.

First, we define some functions that we will use in the construction of the computation $\pi_{\mathsf{TSO}}$. Then, we define a sequence of TSO-configurations that appear in $\pi_{\mathsf{TSO}}$. Finally, we show that the TSO-computation $\pi_{\mathsf{TSO}}$ exists. In particular, the target configuration $target\,(\pi_{\mathsf{TSO}})$ has the same local states as the target $c_n$ of the DTSO-computation $\pi_{\mathsf{DTSO}}$.

Let $1 \leq i_1 < i_2 < \cdots < i_k \leq n$ be the sequence of indices such that $t_{i_1} t_{i_2} \ldots t_{i_k}$ is the sequence of write or atomic read-write operations occurring in the computation $\pi_{\mathsf{DTSO}}$. In the following, we assume that $i_0 = 0$.

For each $j : 0 \leq j \leq n$, we associate a mapping function $\mathsf{index}_j : \mathbb{P} \to \{0, \ldots, k\}^*$ that associates for each process $p \in \mathbb{P}$ and each message at the position $\ell : 1 \leq \ell \leq |\texttt{buffers}\,(c_j)\,(p)\,|$ in the load buffer $\texttt{buffers}\,(c_j)\,(p)$ the index $\mathsf{index}_j(p)(\ell)$, i.e., the index of the last write or atomic write-read operations at the moment this message has been added to the load buffer. Formally, we define $\mathsf{index}_j$ as follows:

(1) $\mathsf{index}_0(p) := \epsilon$ for all $p \in \mathbb{P}$.
(2) Consider j such that $0 \leq j < n$. Recall that $c_j \xrightarrow{t_{j+1}}_{\mathsf{DTSO}} c_{j+1}$ with $t_{j+1} \in \Delta_p \cup \Delta'_p$. We define $\mathsf{index}_{j+1}$ based on $\mathsf{index}_j$:
   - Nop, read, fence, arw: If $t_{j+1}$ is of the following forms $(q, \mathsf{nop}, q')$, $(q, \mathsf{r}(x, v), q')$, $(q, \mathsf{fence}, q')$, or $(q, \mathsf{arw}(x, v, v'), q')$, then $\mathsf{index}_{j+1} := \mathsf{index_j}$.
   - Write: If $t_{j+1}$ is of the form $(q, \mathsf{w}(x, v), q')$, then $\mathsf{index}_{j+1} := \mathsf{index}_j\,[p \hookleftarrow r \cdot \mathsf{index}_j(p)]$ with $i_r = j + 1$.
   - Propagate: If $t_{j+1}$ is of the form $\mathsf{propagate}^x_p$, then $\mathsf{index}_{j+1} := \mathsf{index}_j\,[p \hookleftarrow r \cdot \mathsf{index}_j(p)]$ where $r : 0 \leq r \leq k$ is the maximal index such that $i_r \leq j + 1$.
   - Delete: If $t_{j+1}$ is of the form $\mathsf{delete}_p$, then $\mathsf{index}_j := \mathsf{index}_{j+1}\,[p \hookleftarrow \mathsf{index}_{j+1}(p) \cdot r]$ with $r = \mathsf{index}_j(p)(|\mathsf{index}_j(p)|)$.

Next, we associate for each process $p \in \mathbb{P}$ and $j : 0 \leq j \leq n$, the memory view $\mathsf{view}_p(c_j)$ of the process $p$ in the configuration $c_j$ as follows:

(1) If $\texttt{buffers}\,(c_j)\,(p) = \epsilon$, then $\mathsf{view}_p(c_j) := r$ where $r : 0 \leq r \leq k$ is the maximal index such that $i_r \leq j$.

(2) If $\texttt{buffers}\,(c_j)\,(p) \neq \epsilon$, then $\mathsf{view}_p(c_j) := \mathsf{index}_j(p)(|\mathsf{index}_\mathsf{j}(p)|)$.

**Example A.1.** We give an example of how to calculate the functions `index` and `view` for a DTSO-computation. Let consider the following DTSO-computation

$$\pi_{\mathsf{DTSO}} = c_0 \xrightarrow{t_1}_{\mathsf{DTSO}} c_1 \xrightarrow{t_2}_{\mathsf{DTSO}} c_2 \xrightarrow{t_3}_{\mathsf{DTSO}} c_3 \xrightarrow{t_4}_{\mathsf{DTSO}} c_4 \xrightarrow{t_5}_{\mathsf{DTSO}} c_5$$

containing only transitions of a process $p$ with two variables $x$ and $y$ where $c_i = (\mathbf{q}_i, \mathbf{b}_i, \mathbf{mem}_i)$ for all $i : 0 \leq i \leq n = 5$ such that:

$$\begin{array}{llll}
\mathbf{q}_0(p) = q_0, & \mathbf{b}_0(p) = \epsilon, & \mathbf{mem}_0(x) = 0, \mathbf{mem}_0(y) = 0, & t_1 = (q_0, \mathsf{w}(x,1), q_1)\,, \\
\mathbf{q}_1(p) = q_1, & \mathbf{b}_1(p) = (x, 1, \text{OWN})\,, & \mathbf{mem}_1(x) = 1, \mathbf{mem}_1(y) = 0, & t_2 = \mathsf{propagate}_p^y, \\
\mathbf{q}_2(p) = q_1, & \mathbf{b}_2(p) = (y, 0) \cdot (x, 1, \text{OWN})\,, & \mathbf{mem}_2(x) = 1, \mathbf{mem}_2(y) = 0, & t_3 = \mathsf{delete}_p, \\
\mathbf{q}_3(p) = q_1, & \mathbf{b}_3(p) = (y, 0), & \mathbf{mem}_3(x) = 1, \mathbf{mem}_3(y) = 0, & t_4 = (q_1, \mathsf{r}(y,0), q_2), \\
\mathbf{q}_4(p) = q_2, & \mathbf{b}_4(p) = (y, 0), & \mathbf{mem}_4(x) = 1, \mathbf{mem}_4(y) = 0, & t_5 = \mathsf{delete}_p, \\
\mathbf{q}_5(p) = q_2, & \mathbf{b}_5(p) = \epsilon, & \mathbf{mem}_5(x) = 1, \mathbf{mem}_5(y) = 0. &
\end{array}$$

We note that $n = 5$ and $\pi_{\mathsf{DTSO}}$ contains only transitions of the process $p$. We also note that $k = 1$ and $i_1 = 1$ is the index of the only write transition $t_1$ occurring in the computation $\pi_{\mathsf{DTSO}}$. Following the above definitions of `index` and `view`, we define the functions `index` and `view` as follows:

(1) For each $j : 0 \leq j \leq n = 5$, we define the mapping function $\mathsf{index}_j(p)$:

$$\begin{array}{lll}
\mathsf{index}_0(p) = \epsilon, & \mathsf{index}_1(p) = 1, & \mathsf{index}_2(p) = 1.1, \\
\mathsf{index}_3(p) = 1, & \mathsf{index}_4(p) = 1, & \mathsf{index}_5(p) = \epsilon.
\end{array}$$

(2) For each $j : 0 \leq j \leq n = 5$, we define the memory view $\mathsf{view}_p(c_j)$:

$$\begin{array}{lll}
\mathsf{view}_p(c_0) = 0, & \mathsf{view}_p(c_1) = 1, & \mathsf{view}_p(c_2) = 1, \\
\mathsf{view}_p(c_3) = 1, & \mathsf{view}_p(c_4) = 1, & \mathsf{view}_p(c_5) = 1.
\end{array}$$

△

Now, let $\prec$ be an arbitrary total order on the set of processes and let $p_{min}$ and $p_{max}$ be the smallest and largest elements of $\prec$ respectively. For $p \neq p_{max}$, we define $succ(p)$ to be the successor of $p$ wrt. $\prec$, i.e., $p \prec succ(p)$ and there is no $p'$ with $p \prec p' \prec succ(p)$. We define $prev(p)$ for $p \neq p_{min}$ analogously.

The computation $\pi_{\mathsf{TSO}}$ will consist of $k+1$ phases (henceforth referred to as the phases $0, 1, 2, \ldots, k$). In fact, $\pi_{\mathsf{TSO}}$ will have the *same* sequence of memory updates as $\pi_{\mathsf{DTSO}}$. At the phase $r$, the computation $\pi_{\mathsf{TSO}}$ *simulates* the movements of the processes where their memory view index is $r$. The order in which the processes are simulated during phase $r$ is defined by the ordering $\prec$. First, process $p_{min}$ will perform a sequence of transitions. This sequence is derived from the sequence of transitions it performs in $\pi_{\mathsf{DTSO}}$ where its memory view index is $r$, including "no", write, read, fence transitions. Then, the next process performs its transitions. This continues until $p_{max}$ has made all its transitions. When all processes have performed their transitions in phase $r$, we execute exactly one update transition (possibly with a write transition) or one atomic read-write transition in order to move to phase $r+1$. We start the phase $r+1$ by letting $p_{min}$ execute its transitions, and so on.

Formally, we define a *scheduling function* $\alpha(r, p, \ell)$ that gives for each $r : 0 \leq r \leq k$, $p \in \mathbb{P}$, and $\ell \geq 1$ a natural number $j : 0 \leq j \leq n$ such that process $p$ executes the transition

$t_j$ as its $\ell^{th}$ transition during phase $r$. The scheduling function $\alpha$ is defined as follows where $r : 0 \leq r \leq k$, $p \in \mathbb{P}$, and $\ell \geq 0$:

(1) $\alpha(r, p, \ell + 1)$ is defined to be the smallest $j$ such that $\alpha(r, p, \ell) < j$, $t_j \in \Delta_p$ and $\mathsf{view}_p(c_j) = r$. Intuitively, the $(\ell + 1)^{th}$ transition of process $p$ during phase $r$ is defined by the next transition from $t_{\alpha(r,p,\ell)}$ that belongs to $\Delta_p$. Notice that $\alpha(r, p, \ell + 1)$ is defined only for finitely many $\ell$.

(2) If $\{j \mid \mathsf{view}_p(c_j) = r\} \neq \emptyset$, we define $\alpha(r, p, 0) := min\{j \mid \mathsf{view}_p(c_j) = r\}$. Otherwise, we define $\alpha(r, p, 0) := \alpha(r - 1, p, \sharp(r - 1, p))$ where

$$\sharp(r, p) := max\{\ell \mid \ell \geq 0, \alpha(r, p, \ell) \text{ is defined}\}.$$

Intuitively, phase $r$ starts for process $p$ at the point where its memory view index becomes equal to $r$. Notice that $\alpha(0, p, 0) = 0$ for all $p \in \mathbb{P}$ since all processes are initially in phase 0.

**Example A.2.** In the following, we show how to calculate the scheduling function $\alpha(r, p, \ell)$ and $\sharp(r, p)$ where $r : 0 \leq r \leq k$, $p \in \mathbb{P}$, and $\ell \geq 0$ for the DTSO-computation $\pi_{\mathsf{DTSO}}$ given in Example A.1. We recall that $k = 1$, $n = 5$ and the definitions of the two functions `index` and `view` are given in Example A.1. We also recall that $\pi_{\mathsf{DTSO}}$ contains only transitions of the process $p$. The constructed TSO-computation $\pi_{\mathsf{TSO}}$ from $\pi_{\mathsf{DTSO}}$ will consist of $k + 1 = 2$ phases, referred as the phase 0 and the phase 1. In order to define the transitions of the process $p$ in different phases, for each $r : 0 \leq r \leq k = 1$ and $\ell \geq 0$, the scheduling function $\alpha(r, p, \ell)$ and $\sharp(r, p)$ is defined as follows:

$$\alpha(0, p, 0) = 0, \qquad \alpha(1, p, 0) = 1, \qquad \alpha(1, p, 1) = 4,$$
$$\sharp(0, p) = 0, \qquad \sharp(1, p) = 1. \qquad \triangle$$

In order to define $\pi_{\mathsf{TSO}}$, we first define the set of configurations that will appear in $\pi_{\mathsf{TSO}}$. In more detail, for each $r : 0 \leq r \leq k$, $p \in \mathbb{P}$, and $\ell : 0 \leq \ell \leq \sharp(r, p)$, we define a TSO-configuration $d_{r,p,\ell}$ based on the DTSO-configurations that are appearing in $\pi_{\mathsf{DTSO}}$. We will define $d_{r,p,\ell}$ by defining its local states, buffer contents, and memory state.

(1) We define the *local states* of the processes as follows:
- $\mathtt{states}\,(d_{r,p,\ell})\,(p) := \mathtt{states}\left(c_{\alpha(r,p,\ell)}\right)(p)$. After process $p$ has performed its $\ell^{th}$ transition during phase $r$, its local state is identical to its local state in the corresponding DTSO-configuration $c_{\alpha(r,p,\ell)}$.
- If $p' \prec p$ then $\mathtt{states}\,(d_{r,p,\ell})\,(p') := \mathtt{states}\left(c_{\alpha(r,p',\sharp(r,p'))}\right)(p')$, i.e. the state of $p'$ will not change while $p$ is making its moves. This state is given by the local state of $p'$ after it made its last move during phase $r$.
- If $p \prec p'$ then $\mathtt{states}\,(d_{r,p,\ell})\,(p') := \mathtt{states}\left(c_{\alpha(r,p',0)}\right)(p')$, i.e. the local state of $p'$ will not change while $p$ is making its moves. This state is given by the local state of $p'$ when it entered phase $r$ (before it has made any moves during phase $r$).

(2) To define the *buffer contents*, we give more definitions. For a DTSO-message $a$ of the form $(x, v)$, we define $\mathtt{DTSO2TSO}\,(a)$ to be $\epsilon$. For a DTSO-message $a$ of the form $(x, v, \textsc{own})$, we define $\mathtt{DTSO2TSO}\,(a)$ to be $(x, v)$. From that, we define $\mathtt{DTSO2TSO}\,(\epsilon) = \epsilon$ and $\mathtt{DTSO2TSO}\,(a_1 a_2 \cdots a_n) := \mathtt{DTSO2TSO}\,(a_1) \cdot \mathtt{DTSO2TSO}\,(a_2) \cdots \mathtt{DTSO2TSO}\,(a_n)$, i.e., we concatenate the results of applying the operation individually on each $a_i$ with $1 \leq i \leq n$. We define $\mathtt{DTSO2TSO}_+\,(w)$ for a word $w \in ((\mathbb{X} \times \mathbb{V}) \cup (\mathbb{X} \times \mathbb{V} \times \{\textsc{own}\}))^*$ as follows: If $|w| = 0$ then $\mathtt{DTSO2TSO}_+\,(w) := \epsilon$, else $\mathtt{DTSO2TSO}_+\,(w) := \mathtt{DTSO2TSO}\,(w(1)w(2) \cdots w(|w| - 1))$. In the following, we give the definition of the buffer contents of $d_{r,p,\ell}$:

- $\texttt{buffers}\left(d_{r,p,\ell}\right)(p) := \texttt{DTSO2TSO}_{+}\left(\texttt{buffers}\left(c_{\alpha(r,p,\ell)}\right)(p)\right)$. After process $p$ has performed its $\ell^{th}$ transition during phase $r$, the content of its buffer is defined by considering the buffer of the corresponding DTSO-configuration $c_{\alpha(r,p,\ell)}$ and only messages belong to $p$ (i.e., of the form $(x, v, \textsc{own})$).
- If $p' \prec p$ then $\texttt{buffers}\left(d_{r,p,\ell}\right)(p') := \texttt{buffers}\left(c_{\alpha(r,p',\sharp(r,p'))}\right)(p')$. In a similar manner to the case of states, if $p' \prec p$ then the buffer of $p'$ will not change while $p$ is making its moves.
- If $p \prec p'$ then $\texttt{buffers}\left(d_{r,p,\ell}\right)(p') := \texttt{buffers}\left(c_{\alpha(r,p',0)}\right)(p')$. In a similar manner to the case of states, if $p \prec p'$ then the buffer of $p'$ will not be changed while $p$ is making its moves.

(3) We define the *memory state* as follows:
- $\texttt{mem}\left(d_{r,p,\ell}\right) := \texttt{mem}\left(c_{i_r}\right)$. This definition is consistent with the fact that all processes have identical views of the memory when they are in the same phase $r$. This view is defined by the memory component of $c_{i_r}$.

**Example A.3.** In the following, we give the configurations $d_{r,p,\ell}$ for all $r : 0 \le r \le k$, $p \in \mathbb{P}$, and $\ell : 0 \le \ell \le \sharp(r,p)$ that will appear in the constructed TSO-computation $\pi_{\mathsf{TSO}}$ from $\pi_{\mathsf{DTSO}}$ given in Example A.1. We call that $k = 1$, $n = 5$, and $\pi_{\mathsf{DTSO}}$ contains only transitions of the process $p$. We also recall that the scheduling function $\alpha(r,p,\ell)$ and $\sharp(r,p)$ are given in Example A.2.

For each $r : 0 \le r \le k = 1$ and $\ell : 0 \le \ell \le \sharp(r,p)$, we define the TSO-configurations $d_{r,p,\ell} = (\mathbf{q}_{r,p,\ell}, \mathbf{b}_{r,p,\ell}, \mathbf{mem}_{r,p,\ell})$ based on the DTSO-configurations that are appearing in $\pi_{\mathsf{DTSO}}$ as follows:

$$\begin{array}{llll} d_{0,p,0}: & \mathbf{q}_{0,p,0}(p) = q_0, & \mathbf{b}_{0,p,0}(p) = \epsilon, & \mathbf{mem}_{0,p,0}(x) = 0, \mathbf{mem}_{0,p,0}(y) = 0, \\ d_{1,p,0}: & \mathbf{q}_{1,p,0}(p) = q_1, & \mathbf{b}_{1,p,0}(p) = \epsilon, & \mathbf{mem}_{1,p,0}(x) = 1, \mathbf{mem}_{1,p,0}(y) = 0, \\ d_{1,p,1}: & \mathbf{q}_{1,p,0}(p) = q_2, & \mathbf{b}_{1,p,1}(p) = \epsilon, & \mathbf{mem}_{1,p,1}(x) = 1, \mathbf{mem}_{1,p,1}(y) = 0. \end{array}$$

Finally, we construct the TSO-computation

$$\pi_{\mathsf{TSO}} = d_{0,p,0} \xrightarrow{t'_1}_{\mathsf{TSO}} d'_{0,p,0} \xrightarrow{t'_2}_{\mathsf{TSO}} d_{1,p,0} \xrightarrow{t'_3}_{\mathsf{TSO}} d_{1,p,1}$$

where

$$\begin{aligned} d'_{0,p,0} &= \left(\mathbf{q}'_{0,p,0}, \mathbf{b}'_{0,p,0}, \mathbf{mem}'_{0,p,0}\right), \\ t'_1 &= (q_0, \mathsf{w}(x,1), q_1), \\ t'_2 &= \mathsf{update}_p, \\ t'_3 &= (q_1, \mathsf{r}(y,0), q_2), \end{aligned}$$

and:

$$d'_{0,p,0}: \quad \mathbf{q}'_{0,p,0}(p) = q_1, \quad \mathbf{b}'_{0,p,0}(p) = (x,1), \quad \mathbf{mem}'_{0,p,0}(x) = 0, \mathbf{mem}'_{0,p,0}(y) = 0.$$

Since there is only one update transition in both two computations $\pi_{\mathsf{DTSO}}$ and $\pi_{\mathsf{TSO}}$, it is easy to see that $\pi_{\mathsf{TSO}}$ has the same sequence of memory updates as $\pi_{\mathsf{DTSO}}$. It is also easy to see that $d_{0,p,0} = c_{init}$ and $d_{1,p,\sharp(1,p)} = d_{1,p,1} = \left(\texttt{states}\left(c_5\right), \mathbf{b}, \texttt{mem}\left(c_5\right)\right)$ where $\mathbf{b}(p) = \texttt{buffers}\left(c_5\right)(p) = \epsilon$. Therefore, $\pi_{\mathsf{TSO}}$ is a witness of the construction. $\triangle$

The following lemma shows the existence of a TSO-computation $\pi_{\mathsf{TSO}}$ that starts from the initial TSO-configuration and whose target has the same local state definitions as the target $c_n$ of the DTSO-computation $\pi_{\mathsf{DTSO}}$. This concludes the proof of the if direction of Theorem 3.4.

**Lemma A.4.** *$d_{0,p_{min},0} \xrightarrow{\pi_{\mathsf{TSO}}}_{\mathsf{TSO}} d_{k,p_{max},\sharp(k,p_{max})}$ for some TSO-computation $\pi_{\mathsf{TSO}}$. Furthermore, $d_{0,p_{min},0}$ is the initial TSO-configuration and*

$$d_{k,p_{max},\sharp(k,p_{max})} = \left(\mathtt{states}\left(c_n\right), \mathbf{b}, \mathtt{mem}\left(c_n\right)\right)$$

*where $\mathtt{buffers}\,(p) = \mathtt{buffers}\left(c_n\right)(p) = \epsilon$ for all $p \in \mathbb{P}$.*

*Proof.* Lemmas A.8–A.11 show that the existence of the computation $\pi_{\mathsf{TSO}}$. Lemma A.13 and Lemma A.12 show the conditions on the initial and target configurations. □

First, we start by establishing Lemma A.5, Lemma A.6, and Lemma A.7 that we will use later.

**Lemma A.5.** *For every $j : 0 \leq j \leq n$ and process $p \in \mathbb{P}$, the following properties hold:*

1. $|\mathsf{index}_j(p)| = |\mathtt{buffers}\left(c_j\right)(p)|$.
2. *For every $\ell_1, \ell_2 : 1 \leq \ell_1 \leq \ell_2 \leq |\mathsf{index}_j(p)|$, $\mathsf{index}_j(p)(\ell_2) \leq \mathsf{index}_j(p)(\ell_1) \leq j$.*
3. *For every $\ell_1, \ell_2 : 1 \leq \ell_1 < \ell_2 \leq |\mathsf{index}_j(p)|$ such that $\mathtt{buffers}\left(c_j\right)(p)\left(\ell_1\right)$ is of the form $(x, v, \textsc{own})$, $\mathsf{index}_j(p)(\ell_2) < \mathsf{index}_j(p)(\ell_1)$.*
4. *For every $\ell : 1 \leq \ell \leq |\mathtt{buffers}\left(c_j\right)(p)|$, if $r = \mathsf{index}_j(p)(\ell)$ and $\mathtt{buffers}\left(c_j\right)(p)\left(\ell\right)$ is of the form $(x, v, \textsc{own})$, then $t_{i_r} \in \Delta_p$ and it is of the form $(q, \mathsf{w}(x, v), q')$.*
5. *For every $\ell : 1 \leq \ell \leq |\mathtt{buffers}\left(c_j\right)(p)|$, if $r = \mathsf{index}_j(p)(\ell)$ and $\mathtt{buffers}\left(c_j\right)(p)\left(\ell\right)$ is of the form $(x, v)$, then $\mathtt{mem}\left(c_{i_r}\right)(x) = v$.*
6. *For every $r_1, r_2 : 0 \leq r_1 \leq r_2 \leq k$ such that $r_1 = \textsc{min}\{\mathsf{index_j(p)}(\ell) \,|\, \ell : 1 \leq \ell \leq |\mathsf{index}_j(p)|\}$, $t_{r_2} \in \Delta_p$, and $t_{r_2}$ is of the form $(q, \mathsf{w}(x, v), q')$, then there is an index $\ell : 1 \leq \ell \leq |\mathsf{index}_j(c_j)(p)|$ such that $\mathsf{index}_j(p)(\ell) = r_2$ and $\mathtt{buffers}\left(c_j\right)(p)\left(\ell\right) = (x, v, \textsc{own})$.*

*Proof.* The lemma holds following an immediate consequence of the definition of $\mathsf{index}_j$. □

**Lemma A.6.** *For every process $p \in \mathbb{P}$ and index $j : 0 \leq j < n$, $\mathsf{view}_p(c_j) \leq \mathsf{view}_p(c_{j+1})$. Furthermore, $i_{\mathsf{view}_p(c_j)} \leq j$ and $i_{\mathsf{view}_p(c_{j+1})} \leq j+1$.*

*Proof.* The lemma holds following an immediate consequence of the definitions of $\mathsf{view}_p$ and $\mathsf{index}_j$. □

**Lemma A.7.** *For every natural number $j$ such that $\alpha(r, p, \ell) \leq j < \alpha(r, p, \ell+1) - 1$, $\mathtt{DTSO2TSO}_+\left(\mathtt{buffers}\left(c_j\right)(p)\right) = \mathtt{DTSO2TSO}_+\left(\mathtt{buffers}\left(c_{j+1}\right)(p)\right)$.*

*Proof.* The proof is done by contradiction. Let us assume that there is some $j : \alpha(r, p, \ell) \leq j < \alpha(r, p, \ell+1) - 1$ such that

$$\mathtt{DTSO2TSO}_+\left(\mathtt{buffers}\left(c_j\right)(p)\right) \neq \mathtt{DTSO2TSO}_+\left(\mathtt{buffers}\left(c_{j+1}\right)(p)\right).$$

Observe that the only three operations that can change the content of the load buffer of the process $p$ are write, delete and propagation operations. Since $t_j \notin \Delta_p$ (and so no write operation has been performed) and propagation will append messages of the form $(x, v)$, this implies that $t_j$ is a delete transition of the process $p$ (i.e., $t_j = \mathsf{delete}_p$). Now, the only case when $\mathtt{DTSO2TSO}_+\left(\mathtt{buffers}\left(c_j\right)(p)\right) \neq \mathtt{DTSO2TSO}_+\left(\mathtt{buffers}\left(c_{j+1}\right)(p)\right)$ is where $\mathtt{buffers}\left(c_j\right)(p)$ is of the form $w \cdot (y, v', \textsc{own}) \cdot m$ with $m \in \{(x, v), (x, v, \textsc{own}) \,|\, x \in \mathbb{X}, v \in \mathbb{V}\}$. This implies that $\mathtt{buffers}\left(c_{j+1}\right)(p) = w \cdot (y, v', \textsc{own})$. Now we can use the third case of Lemma A.5 to prove that $\mathsf{view}_p(c_{j+1}) > \mathsf{view}_p(c_j)$. This contradicts the fact that $\mathsf{view}_p(c_{j+1}) \leq \mathsf{view}_p(c_{\alpha(r,p,\ell+1)})$ (see Lemma A.6) since we have $\mathsf{view}_p(c_{\alpha(r,p,\ell)}) =$

$\mathsf{view}_p(c_{\alpha(r,p,\ell+1)}) = r$ (by definition), $\mathsf{view}_p(c_j) \geq \mathsf{view}_p(c_{\alpha(r,p,\ell)})$ (see Lemma A.6) and $\mathsf{view}_p(c_{j+1}) > \mathsf{view}_p(c_j)$. ☐

Now we can start proving the existence of the computation $\pi_{\mathsf{TSO}}$ by showing that we can move from the configuration $d_{r,p,\ell}$ to $d_{r,p,\ell+1}$ using the transition $t_{\alpha(r,p,\ell+1)}$.

**Lemma A.8.** *If* $\ell < \sharp(r,p)$ *then* $d_{r,p,\ell} \xrightarrow{t_{\alpha(r,p,\ell+1)}}_{\mathsf{TSO}} d_{r,p,\ell+1}$.

*Proof.* We recall that $t_{\alpha(r,p,\ell+1)} \in \Delta_p$ by definition. Therefore, $t_{\alpha(r,p,\ell+1)}$ is not a propagation transition nor a delete transition. Furthermore, suppose that $t_{\alpha(r,p,\ell+1)}$ is an atomic read-write transition. It leads to the fact that $\mathsf{view}_p(c_{\alpha(r,p,\ell+1)}) > \mathsf{view}_p(c_{\alpha(r,p,\ell)})$, contradicting to the assumption that we are in phase $r$. Hence, $t_{\alpha(r,p,\ell+1)}$ is not an atomic read-write transition.

Let $t_{\alpha(r,p,\ell+1)} \in \Delta_p$ be of the form $(q, op, q')$. To prove the lemma, we will prove the following properties:

(1) $\mathtt{states}\left(d_{\alpha(r,p,\ell)}\right)(p) = q$ and $\mathtt{states}\left(d_{r,p,\ell+1}\right) = \mathtt{states}\left(d_{\alpha(r,p,\ell)}\right)[p \hookleftarrow q']$,
(2) $\mathtt{states}\left(d_{r,p,\ell+1}\right)(p') = \mathtt{states}\left(d_{r,p,\ell}\right)(p')$ for $p' \neq p$,
(3) $\mathtt{buffers}\left(d_{r,p,\ell+1}\right)(p') = \mathtt{buffers}\left(d_{r,p,\ell}\right)(p')$ for $p' \neq p$,
(4) $\mathtt{mem}\left(d_{r,p,\ell}\right) = \mathtt{mem}\left(d_{r,p,\ell+1}\right) = \mathtt{mem}\left(c_{i_r}\right)$,
(5) The contents of $\mathtt{buffers}\left(d_{r,p,\ell}\right)(p)$ and $\mathtt{buffers}\left(d_{r,p,\ell+1}\right)(p)$ are compatible with the transition $t_{\alpha(r,p,\ell+1)}$. In means that with the properties (1)–(4), the property (5) allows that $d_{r,p,\ell} \xrightarrow{t_{\alpha(r,p,\ell+1)}}_{\mathsf{TSO}} d_{r,p,\ell+1}$.

We prove the property (1). We see from definition of $\alpha$ that $t_j \notin \Delta_p$ for all $j : \alpha(r,p,\ell) < j < \alpha(r,p,\ell+1)$. It follows that $\mathtt{states}\left(c_j\right)(p) = \mathtt{states}\left(c_{\alpha(r,p,\ell)}\right)(p)$ for all $j : \alpha(r,p,\ell) < j < \alpha(r,p,\ell+1)$. In particular, we have $\mathtt{states}\left(c_{\alpha(r,p,\ell+1)-1}\right)(p) = \mathtt{states}\left(c_{\alpha(r,p,\ell)}\right)(p)$. Then, from the fact that $c_{\alpha(r,p,\ell+1)-1} \xrightarrow{t_{\alpha(r,p,\ell+1)}}_{\mathsf{DTSO}} c_{\alpha(r,p,\ell+1)}$ and the definitions of $d_{r,p,\ell}$ and $d_{r,p,\ell+1}$, we know that $\mathtt{states}\left(d_{r,p,\ell}\right)(p) = \mathtt{states}\left(c_{\alpha(r,p,\ell)}\right)(p) = \mathtt{states}\left(c_{\alpha(r,p,\ell+1)-1}\right)(p) = q$. It follows that

$$\mathtt{states}\left(d_{r,p,\ell+1}\right)(p) = \mathtt{states}\left(c_{\alpha(r,p,\ell+1)}\right)(p) = q'.$$

This concludes the property (1).

We prove the property (2). We see from the definitions of $d_{\alpha(r,p,\ell)}$ and $d_{\alpha(r,p,\ell+1)}$ that if $p' \prec p$ then $\mathtt{states}\left(d_{r,p,\ell+1}\right)(p') = \mathtt{states}\left(c_{\alpha(r,p',\sharp(k,p'))}\right)(p') = \mathtt{states}\left(d_{r,p,\ell}\right)(p')$. Moreover, we have if $p \prec p'$ then

$$\mathtt{states}\left(d_{r,p,\ell+1}\right)\left(p'\right) = \mathtt{states}\left(c_{\alpha(r,p',0)}\right)\left(p'\right) = \mathtt{states}\left(d_{r,p,\ell}\right)\left(p'\right).$$

This concludes the property (2).

We prove the properties (3) and (4). In a similar manner to the case of states, we can show the property (3). By the definitions of $d_{\alpha(r,p,\ell)}$ and $d_{\alpha(r,p,\ell+1)}$ and the fact that $\ell < \ell+1 \leq \sharp(r,p)$, we have $\mathtt{mem}\left(d_{r,p,\ell}\right) = \mathtt{mem}\left(c_{i_r}\right) = \mathtt{mem}\left(d_{r,p,\ell+1}\right)$. This concludes the property (4).

Now, it remains to prove the property (5). We consider the cases where $op$ is a write or a read operation. The other cases can be treated in a similar way.

- $op = \mathsf{w}(x,v)$: We see from Lemma A.7 that for all: $j : \alpha(r,p,\ell) < j < \alpha(r,p,\ell+1)$

$$\mathtt{DTSO2TSO}_{+}\left(\mathtt{buffers}\left(c_{\alpha(r,p,\ell)}\right)(p)\right) = \mathtt{DTSO2TSO}_{+}\left(\mathtt{buffers}\left(c_j\right)(p)\right)$$

In particular, we have

$$\texttt{DTSO2TSO}_{+}\left(\texttt{buffers}\left(c_{\alpha(r,p,\ell)}\right)(p)\right) = \texttt{DTSO2TSO}_{+}\left(\texttt{buffers}\left(c_{\alpha(r,p,\ell+1)-1}\right)(p)\right)$$

Then, since $c_{\alpha(r,p,\ell+1)-1} \xrightarrow{t_{\alpha(r,p,\ell+1)}}_{\mathsf{DTSO}} c_{\alpha(r,p,\ell+1)}$, we have $\texttt{buffers}\left(c_{\alpha(r,p,\ell+1)}\right)(p) = (x, v, \text{OWN}) \cdot \texttt{buffers}\left(c_{\alpha(r,p,\ell+1)-1}\right)(p)$.

We will show that $\texttt{buffers}\left(c_{\alpha(r,p,\ell+1)-1}\right)(p) \neq \epsilon$ by contradiction. Let us suppose that $\texttt{buffers}\left(c_{\alpha(r,p,\ell+1)-1}\right)(p) = \epsilon$. By definition, we have $\mathsf{view}_p(c_{\alpha(r,p,\ell+1)}) = r'$ such that $i_{r'} = \alpha(r, p, \ell+1)$. Furthermore, by applying Lemma A.6 to $c_{\alpha(r,p,\ell)}$, we know that $i_r \leq \alpha(r, p, \ell)$. Then, since $\alpha(r, p, \ell) < \alpha(r, p, \ell+1)$ by definition, we have $i_r < i_{r'}$. This contradicts to the fact that $\mathsf{view}_p(c_{\alpha(r,p,\ell+1)}) = r$ by definition. Therefore, we have $\texttt{buffers}\left(c_{\alpha(r,p,\ell+1)-1}\right)(p) \neq \epsilon$.

As a consequence of the fact that $\texttt{buffers}\left(c_{\alpha(r,p,\ell+1)-1}\right)(p) \neq \epsilon$, we know that

$$\texttt{DTSO2TSO}_{+}\left(\texttt{buffers}\left(c_{\alpha(r,p,\ell+1)}\right)(p)\right) = (x, v) \cdot \texttt{DTSO2TSO}_{+}\left(\texttt{buffers}\left(c_{\alpha(r,p,\ell)-1}\right)(p)\right)$$

Then, since

$$\texttt{DTSO2TSO}_{+}\left(\texttt{buffers}\left(c_{\alpha(r,p,\ell)}\right)(p)\right) = \texttt{DTSO2TSO}_{+}\left(\texttt{buffers}\left(c_{\alpha(r,p,\ell+1)-1}\right)(p)\right),$$

it follows that $\texttt{buffers}(d_{r,p,\ell+1}) = (x, v) \cdot \texttt{buffers}(d_{r,p,\ell})$. Hence this implies that $d_{r,p,\ell} \xrightarrow{t_{\alpha(r,p,\ell+1)}}_{\mathsf{TSO}} d_{r,p,\ell+1}$.

- $op = \mathsf{r}(x, v)$: We see from Lemma A.7 that for all $j : \alpha(r, p, \ell) < j < \alpha(r, p, \ell+1)$

$$\texttt{DTSO2TSO}_{+}\left(\texttt{buffers}\left(c_{\alpha(r,p,\ell)}\right)(p)\right) = \texttt{DTSO2TSO}_{+}\left(\texttt{buffers}\left(c_j\right)(p)\right)$$

In particular, we have

$$\texttt{DTSO2TSO}_{+}\left(\texttt{buffers}\left(c_{\alpha(r,p,\ell)}\right)(p)\right) = \texttt{DTSO2TSO}_{+}\left(\texttt{buffers}\left(c_{\alpha(r,p,\ell+1)-1}\right)(p)\right)$$

Then, since $c_{\alpha(r,p,\ell+1)-1} \xrightarrow{t_{\alpha(r,p,\ell+1)}}_{\mathsf{DTSO}} c_{\alpha(r,p,\ell+1)}$, we have $\texttt{buffers}\left(c_{\alpha(r,p,\ell+1)}\right)(p) = \texttt{buffers}\left(c_{\alpha(r,p,\ell+1)-1}\right)(p)$. Therefore, $\texttt{buffers}(d_{r,p,\ell+1})(p) = \texttt{buffers}(d_{r,p,\ell})(p)$. We consider two cases about the type of the operation $op$:

– Read own write: We see that there is an $i : 1 \leq i < |\texttt{buffers}\left(c_{\alpha(r,p,\ell+1)-1}\right)(p)|$ such that $\texttt{buffers}\left(c_{\alpha(r,p,\ell+1)-1}\right)(p)(i) = (x, v, \text{OWN})$, and that there are no $j : 1 \leq j < i$ and $v' \in \mathbb{V}$ such that $\texttt{buffers}\left(c_{\alpha(r,p,\ell+1)-1}\right)(p)(j) = (x, v', \text{OWN})$. As a consequence, this implies that there is an $i' : 1 \leq i' \leq |\texttt{DTSO2TSO}_{+}\left(\texttt{buffers}\left(c_{\alpha(r,p,\ell+1)-1}\right)(p)\right)|$ such that $\texttt{DTSO2TSO}_{+}\left(\texttt{buffers}\left(c_{\alpha(r,p,\ell+1)-1}\right)(p)(i')\right) = (x, v)$ and there are no $j' : 1 \leq j' < i'$ and $v' \in \mathbb{V}$ such that $\texttt{DTSO2TSO}_{+}\left(\texttt{buffers}\left(c_{\alpha(r,p,\ell+1)-1}\right)(p)(j)\right) = (x, v')$. From the fact that

$$\texttt{buffers}(d_{r,p,\ell+1})(p) = \texttt{buffers}(d_{r,p,\ell})(p) = \texttt{DTSO2TSO}_{+}\left(\texttt{buffers}\left(c_{\alpha(r,p,\ell+1)-1}\right)\right),$$

we have $d_{r,p,\ell} \xrightarrow{t_{\alpha(r,p,\ell+1)}}_{\mathsf{TSO}} d_{r,p,\ell+1}$.

– Read memory: We consider two cases:

▷ $\texttt{buffers}\left(c_{\alpha(r,p,\ell+1)-1}\right)(p)(i) = (x, v, \text{OWN})$ where $i = |\texttt{buffers}\left(c_{\alpha(r,p,\ell+1)-1}\right)(p)|$ and there are no $j : 1 \leq j < i$ and $v' \in \mathbb{V}$ such that $\texttt{buffers}\left(c_{\alpha(r,p,\ell+1)-1}\right)(p)(j) = (x, v', \text{OWN})$: Since $\mathsf{view}_p(c_{\alpha(r,p,\ell+1)-1}) = r$, this implies from Lemma A.5 that $t_{i_r} \in \Delta_p$ and it is of the form $(q, \mathsf{w}(x, v), q')$. Hence, we see that $\texttt{mem}(c_{i_r})(x) = v$ and thus $\texttt{mem}\left(d_{r,p,\ell}\right)(x) = \texttt{mem}\left(d_{r,p,\ell+1}\right)(x) = v$. Therefore, we have $d_{r,p,\ell} \xrightarrow{t_{\alpha(r,p,\ell+1)}}_{\mathsf{TSO}} d_{r,p,\ell+1}$.

▷ $(x, v', \textsc{own}) \notin \texttt{buffers}\left(c_{\alpha(r,p,\ell+1)-1}\right)(p)$ for all $v' \in \mathbb{V}$: Thus $\texttt{buffers}\left(c_{\alpha(r,p,\ell+1)-1}\right)$ is of the form $w \cdot (x, v)$. Since $\mathsf{view}_p(c_{\alpha(r,p,\ell+1)-1}) = r$, this implies from Lemma A.5 that $\texttt{mem}\left(c_{i_r}\right)(x) = v$ and thus $\texttt{mem}\left(d_{r,p,\ell}\right)(x) = \texttt{mem}\left(d_{r,p,\ell+1}\right)(x) = v$. Therefore, we have $d_{r,p,\ell} \xrightarrow{t_{\alpha(r,p,\ell+1)}}_{\mathsf{TSO}} d_{r,p,\ell+1}$.

This concludes the proof of Lemma A.8. □

**Lemma A.9.** *If $p \prec p_{max}$ then $d_{r,p,\sharp(r,p)} = d_{r,succ(p),0}$.*

*Proof.* To prove the lemma, we will prove the following properties:

(1) $\texttt{states}\left(d_{r,p,\sharp(r,p)}\right)(p') = \texttt{states}\left(d_{r,succ(p),0}\right)(p')$ for all $p' \in \mathbb{P}$,

(2) $\texttt{buffers}\left(d_{r,p,\sharp(r,p)}\right)(p') = \texttt{buffers}\left(d_{r,succ(p),0}\right)(p')$ for all $p' \in \mathbb{P}$,

(3) $\texttt{mem}\left(d_{r,p,\sharp(r,p)}\right) = \texttt{mem}\left(d_{r,succ(p),0}\right)$.

We prove the property (1) by considering four cases:

- $p' = p$: From the definitions of $d_{r,succ(p),\ell}$ and $d_{r,p,\sharp(r,p)}$, we have $\texttt{states}\left(d_{r,succ(p),\ell}\right)(p) = \texttt{states}\left(d_{r,p,\sharp(r,p)}\right)(p)$ for all $\ell : 0 \leq \ell \leq \sharp(r, succ(p))$. In particular, we see that $\texttt{states}\left(d_{r,succ(p),0}\right)(p) = \texttt{states}\left(d_{r,p,\sharp(r,p)}\right)(p)$.
- $p' = succ(p)$: From the definitions of $d_{r,p,\ell}$ and $d_{r,succ(p),0}$, $\texttt{states}\left(d_{r,p,\ell}\right)(succ(p)) = \texttt{states}\left(d_{r,succ(p),0}\right)(succ(p))$ for all $\ell : 0 \leq \ell \leq \sharp(r,p)$. In particular, we see that $\texttt{states}\left(d_{r,p,\sharp(r,p)}\right)(succ(p)) = \texttt{states}\left(d_{r,succ(p),0}\right)(succ(p))$. It follows from $p' = succ(p)$ that $\texttt{states}\left(d_{r,p,\sharp(r,p)}\right)(p') = \texttt{states}\left(d_{r,succ(p),0}\right)(p')$.
- $p' \prec p \prec succ(p)$: From the definitions of $d_{r,succ(p),\ell}$ and $d_{r,p',\sharp(r,p')}$, we know that $\texttt{states}\left(d_{r,succ(p),\ell}\right)(p') = \texttt{states}\left(d_{r,p',\sharp(r,p')}\right)(p')$ for all $\ell : 0 \leq \ell \leq \sharp(r, succ(p))$. In particular, we see that $\texttt{states}\left(d_{r,succ(p),0}\right)(p') = \texttt{states}\left(d_{r,p',\sharp(r,p')}\right)(p')$. Also, by a similar argument, we have $\texttt{states}\left(d_{r,p,\ell}\right)(p') = \texttt{states}\left(d_{r,p',\sharp(r,p')}\right)(p')$ for all $\ell : 0 \leq \ell \leq \sharp(r,p)$. In particular, we see that $\texttt{states}\left(d_{r,p,\sharp(r,p)}\right)(p') = \texttt{states}\left(d_{r,p',\sharp(r,p')}\right)(p')$. Hence, we have $\texttt{states}\left(d_{r,succ(p),0}\right)(p') = \texttt{states}\left(d_{r,p,\sharp(r,p)}\right)(p')$.
- $p \prec succ(p) \prec p'$: From the definitions of $d_{r,succ(p),\ell}$ and $d_{r,p',0}$, we can see that $\texttt{states}\left(d_{r,succ(p),\ell}\right)(p') = \texttt{states}\left(d_{r,p',0}\right)(p')$ for all $\ell : 0 \leq \ell \leq \sharp(r,p)$. In particular, we see that $\texttt{states}\left(d_{r,succ(p),0}\right)(p') = \texttt{states}\left(d_{r,p',0}\right)(p')$. Also, by a similar argument, we have $\texttt{states}\left(d_{r,p,\ell}\right)(p') = \texttt{states}\left(d_{r,p',0}\right)(p')$ for all $\ell : 0 \leq \ell \leq \sharp(r,p)$. In particular, we see that $\texttt{states}\left(d_{r,p,\sharp(r,p)}\right)(p') = \texttt{states}\left(d_{r,p',0}\right)(p')$. Hence, we have $\texttt{states}\left(d_{r,succ(p),0}\right)(p') = \texttt{states}\left(d_{r,p,\sharp(r,p)}\right)(p')$.

We prove the properties (2) and (3). By a similar manner to the case of states, we can show the property (2). Finally, to show the property (3), by the definition of $d_{r,p,\sharp(r,p)}$, it follows that $\texttt{mem}\left(d_{r,p,\sharp(r,p)}\right) = \texttt{mem}\left(c_{i_r}\right)$. Also, by a similar argument, we have $\texttt{mem}\left(d_{r,succ(p),0}\right) = \texttt{mem}\left(c_{i_r}\right)$. Hence, we have $\texttt{mem}\left(d_{r,p,\sharp(r,p)}\right) = \texttt{mem}\left(d_{r,succ(p),0}\right)$.

This concludes the proof of Lemma A.9. □

**Lemma A.10.** *If $r < k$ and $t_{i_{r+1}} \in \Delta_{p^u}$ such that $t_{i_{r+1}}$ is of the form $(q, \mathsf{arw}(x, v, v'), q')$, then $d_{r,p_{max},\sharp(r,p_{max})} \xrightarrow{t_{i_{r+1}}}_{\mathsf{TSO}} d_{r+1,p_{min},0}$.*

*Proof.* To prove the lemma, we will prove the following properties:

(1) $\texttt{states}\left(d_{r,p_{max},\sharp(r,p_{max})}\right)(p) = \texttt{states}\left(d_{r+1,p_{min},0}\right)(p)$ for all $p \neq p^u$,

(2) $\texttt{buffers}\left(d_{r,p_{max},\sharp(r,p_{max})}\right)(p) = \texttt{buffers}\left(d_{r+1,p_{min},0}\right)(p)$ for all $p \neq p^u$,
(3) $\texttt{states}\left(d_{r,p_{max},\sharp(r,p_{max})}\right)(p^u) = q$, and $\texttt{states}\left(d_{r+1,p_{min},0}\right)(p^u) = q'$,
(4) $\texttt{buffers}\left(d_{r,p_{max},\sharp(r,p_{max})}\right)(p^u) = \texttt{buffers}\left(d_{r+1,p_{min},0}\right)(p^u) = \epsilon$,
(5) $\texttt{mem}\left(d_{r,p_{max},\sharp(r,p_{max})}\right)(x) = v$ and $\texttt{mem}\left(d_{r+1,p_{min},0}\right)(x) = v'$.

We show the property (1). Let $p \in \mathbb{P} \setminus \{p^u\}$. From the definition of $d_{r,p_{max},\sharp(r,p_{max})}$ and $d_{r,p,\sharp(r,p)}$, $\texttt{states}\left(d_{r,p_{max},\sharp(r,p_{max})}\right)(p) = \texttt{states}\left(d_{r,p,\sharp(r,p)}\right)(p) = \texttt{states}\left(c_{\alpha(r,p,\sharp(r,p))}\right)(p)$ and $\texttt{states}\left(d_{r+1,p_{min},0}\right)(p) = \texttt{states}\left(d_{r+1,p,0}\right)(p) = \texttt{states}\left(c_{\alpha(r+1,p,0)}\right)(p)$. From the definition of $\alpha$, it follows that $t_j \notin \Delta_p$ for all $j : \alpha(r,p,\sharp(r,p)) \leq j < \alpha(r+1,p,0)$. This implies that $\texttt{states}\left(c_{\alpha(r+1,p,0)-1}\right)(p) = \texttt{states}\left(c_{\alpha(r,p,\sharp(r,p))}\right)(p)$. Now we have two cases:

- $\{j \,|\, \mathsf{view}_p(c_j) = r+1\} = \emptyset$: We see that $\alpha(r+1,p,0) = \alpha(r,p,\sharp(r,p))$, and hence that $\texttt{states}\left(d_{r,p_{max},\sharp(r,p_{max})}\right)(p) = \texttt{states}\left(d_{r+1,p_{min},0}\right)(p)$.
- $\{j \,|\, \mathsf{view}_p(c_j) = r+1\} \neq \emptyset$: Since $\mathsf{view}_p(c_{\alpha(r+1,p,0)-1}) = r$, we can show that $t_{\alpha(r+1,p,0)} \notin \Delta_p$. This is done by contradiction as follows. In fact if $t_{\alpha(r+1,p,0)} \in \Delta_p$, then it is either a write transition or an atomic read-write transition. This implies that in both cases that $\texttt{buffers}\left(c_{\alpha(r+1,p,0)-1}\right)(p) = \epsilon$ and that $\mathsf{view}_p(c_{\alpha(r+1,p,0)}) = \alpha(r+1,p,0)$. Hence, we have $\alpha(r+1,p,0) = r+1$, and this leads to a contradiction since $t_{i_{r+1}} \in \Delta_{p^u}$. Thus, we have $\texttt{states}\left(d_{r,p_{max},\sharp(r,p_{max})}\right)(p) = \texttt{states}\left(d_{r+1,p_{min},0}\right)(p)$.

In a similar manner to the case of states, we can show the property (2). Now we show the properties (3) and (4). Using a similar reasoning as for the process $p$, we know that $\texttt{states}\left(c_{\alpha(r+1,p^u,0)-1}\right)(p^u) = \texttt{states}\left(c_{\alpha(r,p^u,\sharp(r,p^u))}\right)(p^u)$. From the definition of $\pi_{\mathsf{DTSO}}$, it follows that $c_{\alpha(r+1,p^u,0)-1} \xrightarrow{t_{\alpha(r+1,p^u,0)}}_{\mathsf{DTSO}} c_{\alpha(r+1,p^u,0)}$. Furthermore, since $\mathsf{view}_p(c_{\alpha(r+1,p^u,0)}) = r+1$ and $\mathsf{view}_p(c_{\alpha(r+1,p^u,0)-1}) < r+1$, we know that $t_{\alpha(r+1,p^u,0)} = t_{i_{r+1}}$. This implies that $\texttt{buffers}\left(c_{\alpha(r+1,p^u,0)-1}\right)(p^u) = \texttt{buffers}\left(c_{\alpha(r+1,p^u,0)}\right)(p^u) = \epsilon$ and $\texttt{states}\left(c_{\alpha(r+1,p^u,0)-1}\right)(p^u) = q$ and $\texttt{states}\left(c_{\alpha(r+1,p^u,0)}\right)(p^u) = q'$. Now since

$$\texttt{DTSO2TSO}_+\left(\texttt{buffers}\left(c_{\alpha(r,p^u,\sharp(r,p^u))}\right)(p^u)\right) = \texttt{DTSO2TSO}_+\left(\texttt{buffers}\left(c_{\alpha(r+1,p^u,0)-1}\right)(p^u)\right),$$

we see that

$$\texttt{buffers}\left(d_{r,p_{max},\sharp(r,p_{max})}\right)(p^u) = \texttt{buffers}\left(d_{r+1,p_{min},0}\right)(p^u) = \epsilon$$

and that $\texttt{states}\left(d_{r,p_{max},\sharp(r,p_{max})}\right)(p^u) = q$ and $\texttt{states}\left(d_{r+1,p_{min},0}\right)(p^u) = q'$. This concludes the properties (3) and (4).

We show the property (5). From the definition of $\pi_{\mathsf{DTSO}}$, it follows that $\texttt{mem}\left(c_{i_{r+1}}\right) = \texttt{mem}\left(c_{i_r}\right)[x \hookleftarrow v']$ with $\texttt{mem}\left(c_{i_r}\right)(x) = v$. Then from the definitions of $d_{r,p_{max},\sharp(r,p_{max})}$ and $d_{r+1,p_{min},0}$, we have the property (5).

This concludes the proof of Lemma A.10. □

**Lemma A.11.** *If $r < k$ and $t_{i_{r+1}} \in \Delta_{p^u}$ such that $t_{i_{r+1}}$ is of the form $(q, \mathsf{w}(x,v), q')$, then $d_{r,p_{max},\sharp(r,p_{max})} \xrightarrow{*}_{\mathsf{TSO}} d_{r+1,p_{min},0}$.*

*Proof.* To prove the lemma, we will prove the following properties:
(1) $\texttt{states}\left(d_{r,p_{max},\sharp(r,p_{max})}\right)(p) = \texttt{states}\left(d_{r+1,p_{min},0}\right)(p)$ for all $p \neq p^u$,
(2) $\texttt{buffers}\left(d_{r,p_{max},\sharp(r,p_{max})}\right)(p) = \texttt{buffers}\left(d_{r+1,p_{min},0}\right)(p)$ for all $p \neq p^u$,
(3) The contents of buffers $\texttt{states}\left(d_{r,p_{max},\sharp(r,p_{max})}\right)(p^u)$ and $\texttt{states}\left(d_{r+1,p_{min},0}\right)(p^u)$ are compatible, i.e. with the properties (1)–(2), the property (3) allows $d_{r,p_{max},\sharp(r,p_{max})} \xrightarrow{*}_{\mathsf{TSO}} d_{r+1,p_{min},0}$.

We show the property (1). Let $p \in \mathbb{P} \setminus \{p^u\}$. From the definitions of $d_{r,p_{max},\sharp(r,p_{max})}$ and $d_{r,p,\sharp(r,p)}$, $\texttt{states}\left(d_{r,p_{max},\sharp(r,p_{max})}\right)(p) = \texttt{states}\left(d_{r,p,\sharp(r,p)}\right)(p) = \texttt{states}\left(c_{\alpha(r,p,\sharp(r,p))}\right)(p)$ and that $\texttt{states}\left(d_{r+1,p_{min},0}\right)(p) = \texttt{states}\left(d_{r+1,p,0}\right)(p) = \texttt{states}\left(c_{\alpha(r+1,p,0)}\right)(p)$. From the definition of $\alpha$, it follows that $t_j \notin \Delta_p$ for all $j : \alpha(r,p,\sharp(r,p)) \leq j < \alpha(r+1,p,0)$. This implies that $\texttt{states}\left(c_{\alpha(r+1,p,0)-1}\right)(p) = \texttt{states}\left(c_{\alpha(r,p,\sharp(r,p))}\right)(p)$. Now we have two cases:

- $\{j \,|\, \mathsf{view}_p(c_j) = r+1\} = \emptyset$: We see that $\alpha(r+1,p,0) = \alpha(r,p,\sharp(r,p))$, and hence that $\texttt{states}\left(d_{r,p_{max},\sharp(r,p_{max})}\right)(p) = \texttt{states}\left(d_{r+1,p_{min},0}\right)(p)$.
- $\{j \,|\, \mathsf{view}_p(c_j) = r+1\} \neq \emptyset$: Since $\mathsf{view}_p(c_{\alpha(r+1,p,0)-1}) = r$, we can show that $t_{\alpha(r+1,p,0)} \notin \Delta_p$ . This is done by contradiction as follows. In fact if $t_{\alpha(r+1,p,0)} \in \Delta_p$, then it is either a write transition or an atomic read-write transition. This implies that in both cases that $\texttt{buffers}\left(c_{\alpha(r+1,p,0)-1}\right)(p) = \epsilon$ and that $\mathsf{view}_p(c_{\alpha(r+1,p,0)}) = \alpha(r+1,p,0)$. Hence, we have $\alpha(r+1,p,0) = r+1$, and this leads to a contradiction since $t_{i_{r+1}} \in \Delta_{p^u}$. Thus, we have $\texttt{states}\left(d_{r,p_{max},\sharp(r,p_{max})}\right)(p) = \texttt{states}\left(d_{r+1,p_{min},0}\right)(p)$.

In a similar manner to the case of states, we can show the property (2). Now we show the property (3). Using a similar reasoning as for the process $p$, we know that $\texttt{states}\left(c_{\alpha(r+1,p^u,0)-1}\right)(p^u) = \texttt{states}\left(c_{\alpha(r,p^u,\sharp(r,p^u))}\right)(p^u)$. From the definition of $\pi_{\mathsf{DTSO}}$, it follows that $c_{\alpha(r+1,p^u,0)-1} \xrightarrow{t_{\alpha(r+1,p^u,0)}}_{\mathsf{DTSO}} c_{\alpha(r+1,p^u,0)}$. Furthermore, from the fact that $\mathsf{view}_p(c_{\alpha(r+1,p^u,0)}) = r+1$ and $\mathsf{view}_p(c_{\alpha(r+1,p^u,0)-1}) < r+1$, we have two cases to consider:

- $\texttt{buffers}\left(c_{\alpha(r+1,p^u,0)-1}\right)(p^u) = \epsilon$: It follows from the conditions for $\mathsf{view}_p(c_{\alpha(r+1,p^u,0)})$ and $\mathsf{view}_p(c_{\alpha(r+1,p^u,0)-1})$ that $t_{\alpha(r+1,p^u,0)} = t_{i_{r+1}}$, $\texttt{buffers}\left(c_{\alpha(r+1,p^u,0)}\right)(p^u) = (x, v, \mathrm{OWN})$, and that $\texttt{states}\left(c_{\alpha(r+1,p^u,0)-1}\right)(p^u) = q$ and $\texttt{states}\left(c_{\alpha(r+1,p^u,0)}\right)(p^u) = q'$. From
$$\texttt{DTSO2TSO}_+\left(\texttt{buffers}\left(c_{\alpha(r,p^u,\sharp(r,p^u))}\right)(p^u)\right) = \texttt{DTSO2TSO}_+\left(\texttt{buffers}\left(c_{\alpha(r+1,p^u,0)-1}\right)(p^u)\right)$$
we have $\texttt{buffers}\left(d_{r,p_{max},\sharp(r,p_{max})}\right)(p^u) = \texttt{buffers}\left(d_{r+1,p_{min},0}\right)(p^u) = \epsilon$. Moreover, we have $\texttt{states}\left(d_{r,p_{max},\sharp(r,p_{max})}\right)(p^u) = q$, and $\texttt{states}\left(d_{r+1,p_{min},0}\right)(p^u) = q'$. Then, it is easy to see that $\texttt{mem}\left(c_{i_{r+1}}\right) = \texttt{mem}\left(c_{i_r}\right)[x \hookleftarrow v]$. Hence, we have $d_{r,p_{max},\sharp(r,p_{max})} \xrightarrow{t_{i_{r+1}}}_{\mathsf{TSO}} d' \xrightarrow{\mathsf{update}_{p^u}}_{\mathsf{TSO}} d_{r+1,p_{min},0}$ for some configuration $d'$.
- $\texttt{buffers}\left(c_{\alpha(r+1,p^u,0)-1}\right)(p^u) \neq \epsilon$: It follows from the conditions for $\mathsf{view}_p(c_{\alpha(r+1,p^u,0)})$ and $\mathsf{view}_p(c_{\alpha(r+1,p^u,0)-1})$ that $t_{\alpha(r+1,p^u,0)}$ is a delete transition of the process $p^u$. As a consequence, $\texttt{buffers}\left(c_{\alpha(r+1,p^u,0)-1}\right)(p^u) = w \cdot (x, v, \mathrm{OWN}) \cdot m$ and $\texttt{buffers}\left(c_{\alpha(r+1,p^u,0)}\right)(p^u) = w \cdot (x, v, \mathrm{OWN})$. Hence, we see that
$$\texttt{DTSO2TSO}_+\left(\texttt{buffers}\left(c_{\alpha(r+1,p^u,0)}\right)(p^u)\right) = \texttt{DTSO2TSO}_+\left(\texttt{buffers}\left(c_{\alpha(r+1,p^u,0)-1}\right)(p^u)\right) \cdot (x, v)$$
and therefore $\texttt{buffers}\left(d_{r+1,p_{min},0}\right)(p^u) = \texttt{buffers}\left(d_{r,p_{max},\sharp(r,p_{max})}\right)(p^u)\,(x, v)$. Furthermore, we have $\texttt{states}\left(c_{\alpha(r+1,p^u,0)}\right)(p^u) = \texttt{states}\left(c_{\alpha(r+1,p^u,0)-1}\right)(p^u)$ and this implies that $\texttt{states}\left(d_{r,p_{max},\sharp(r,p_{max})}\right)(p^u) = \texttt{states}\left(d_{r+1,p_{min},0}\right)(p^u)$. Then, it is easy to see that $\texttt{mem}\left(c_{i_{r+1}}\right) = \texttt{mem}\left(c_{i_r}\right)[x \hookleftarrow v]$. Hence, we have $d_{r,p_{max},\sharp(r,p_{max})} \xrightarrow{\mathsf{update}_{p^u}}_{\mathsf{TSO}} d_{r+1,p_{min},0}$.

This concludes the proof of Lemma A.11. □

The following lemma shows that the TSO-computation $\pi_{\mathsf{TSO}}$ starts from the initial TSO-configuration.

**Lemma A.12.** $d_{0,p_{min},0}$ *is the initial TSO-configuration.*

*Proof.* Let us take any $p \in \mathbb{P}$. By the definitions of $d_{0,p_{min},0}$, $d_{0,p,0}$, and $\alpha(0,p,0)$, it follows that $\mathtt{states}\left(d_{0,p_{min},0}\right)(p) = \mathtt{states}\left(d_{0,p,0}\right)(p) = \mathtt{states}\left(c_{\alpha(0,p,0)}\right)(p) = \mathtt{states}\left(c_0\right)(p) = \mathbf{q}_{init}$. Also, $\mathtt{buffers}\left(d_{0,p_{min},0}\right)(p) = \mathtt{buffers}\left(d_{0,p,0}\right)(p) = \mathtt{DTSO2TSO}_{+}\left(\mathtt{buffers}\left(c_0\right)(p)\right) = \epsilon$. Finally, we have $\mathtt{mem}\left(d_{0,p_{min},0}\right) = \mathtt{mem}\left(c_{i_0}\right) = \mathtt{mem}\left(c_0\right)$. The result follows immediately for the definition of the initial TSO-configuration. This concludes the proof of Lemma A.12. □

The following lemma shows that the target of the TSO-computation $\pi_{\mathsf{TSO}}$ has the same local process states as the target $c_n$ of the DTSO-computation $\pi_{\mathsf{DTSO}}$.

**Lemma A.13.** $\mathtt{states}\left(d_{k,p_{max},\sharp(k,p_{max})}\right) = \mathtt{states}\left(c_n\right)$.

*Proof.* Let us take any $p \in \mathbb{P}$. By the definitions of $d_{k,p_{max},\sharp(k,p_{max})}$ and $d_{k,p,\sharp(k,p)}$, it follows that $\mathtt{states}\left(d_{k,p_{max},\sharp(k,p_{max})}\right)(p) = \mathtt{states}\left(d_{k,p,\sharp(k,p)}\right)(p) = \mathtt{states}\left(c_{\alpha(k,p,\sharp(k,p))}\right)(p)$. By definition of $\alpha(k,p,\sharp(k,p))$, we know that $t_j \notin \Delta_p$ for all $j : \alpha(k,p,\sharp(k,p)) < j \leq n$. Therefore, we have $\mathtt{states}\left(c_j\right)(p) = \mathtt{states}\left(c_n\right)(p)$ for all $j : \alpha(k,p,\sharp(k,p)) \leq j < n$. In particular, we have $\mathtt{states}\left(c_{\alpha(k,p,\sharp(k,p))}\right)(p) = \mathtt{states}\left(c_n\right)(p)$. Hence, we have $\mathtt{states}\left(d_{k,p_{max},\sharp(r,p_{max})}\right)(p) = \mathtt{states}\left(c_n\right)(p)$. This concludes the proof of Lemma A.13. □

A.2. **From TSO to Dual TSO.** We show the *only if direction* of Theorem 3.4. Consider a TSO-computation

$$\pi_{\mathsf{TSO}} = c_0 \xrightarrow{t_1}_{\mathsf{TSO}} c_1 \xrightarrow{t_2}_{\mathsf{TSO}} c_2 \cdots \xrightarrow{t_{n-1}}_{\mathsf{TSO}} c_{n-1} \xrightarrow{t_n}_{\mathsf{TSO}} c_n.$$

where $c_0 = c_{init}$ and $c_i$ is of the form $(\mathbf{q}_i, \mathbf{b}_i, \mathbf{mem}_i)$ for all $i : 1 \leq i \leq n$ with $\mathbf{q}_n = \mathbf{q}_{target}$. In the following, we will derive a DTSO-computation $\pi_{\mathsf{DTSO}}$ such that $\mathtt{states}\left(\mathit{target}\left(\pi_{\mathsf{DTSO}}\right)\right) = \mathtt{states}\left(c_n\right)$, i.e. the runs $\pi_{\mathsf{TSO}}$ and $\pi_{\mathsf{DTSO}}$ reach the same set of local states at the end of the runs.

Similar to the previous case, we will first define some functions that we will use in the construction of the computation $\pi_{\mathsf{DTSO}}$. Then, we define a sequence of DTSO-configurations that appear in $\pi_{\mathsf{DTSO}}$. Finally, we show that the DTSO-computation $\pi_{\mathsf{DTSO}}$ exists. In particular, the target configuration $\mathit{target}\left(\pi_{\mathsf{DTSO}}\right)$ has the same local states as the target $c_n$ of the TSO-computation $\pi_{\mathsf{TSO}}$.

For every $p \in \mathbb{P}$, let $\Delta_p^{\mathsf{w,arw}} \subseteq \Delta_p$ (resp. $\Delta_p^{\mathsf{u,arw}} \subseteq \Delta_p \cup \{\mathsf{update}_p\}$) be the set of write (resp. update) and atomic read-write transitions that can be performed by process $p$. Let $\Delta_p^{\mathsf{r}}$ be the set of read transitions that can be performed by $p$.

Let $I = i_1 \ldots i_m$ be the maximal sequence of indices such that $1 \leq i_1 < i_2 < \cdots < i_m \leq n$ and for every $j : 1 \leq j \leq m$, we have $t_{i_j}$ is an *update* transition or an atomic read-write transition (i.e., $t_{i_j} \in \bigcup_{p\in\mathbb{P}} \Delta_p^{\mathsf{u,arw}}$). In the following, we assume that $i_0 = 0$. Let $I_p$ be the maximal subsequence of $I$ such that all transitions with indices in $I_p$ belong to process $p$.

Let $I' = i'_1 \ldots i'_m$ be the maximal sequence of indices such that $1 \leq i'_1 < i'_2 < \cdots < i'_m \leq n$ and for every $j : 1 \leq j \leq m$, we have $t_{i'_j}$ is a *write* transition or an atomic read-write transition (i.e., $t_{i'_j} \in \bigcup_{p\in\mathbb{P}} \Delta_p^{\mathsf{w,arw}}$). Let $I'_p$ be the maximal subsequence of $I'$ such that all transitions with indices in $I'_p$ belong to process $p$. Observe that $|I_p| = |I'_p|$.

For every $j : 1 \leq j \leq m$, let $\mathtt{proc}\,(j)$ be the process that has the update or atomic read-write transition $t_{i_j}$ where $i_j \in I$. We define $\mathtt{match}\,(i_j)$ to be the index of the write (resp. atomic read-write) transition $t_{\mathtt{match}(i_j)}$ that corresponds to the update (resp. atomic read-write) transition $t_{i_j}$. Formally, $\mathtt{match}\,(i_j) := l$ where $\exists k : 1 \leq k \leq |I_p|, I_p(k) = i_j$, $I'_p(k) = l$ and $1 \leq l \leq n$. Observe that if $t_{i_j}$ is an atomic read-write operation, then $\mathtt{match}\,(i_j) = i_j$.

**Example A.14.** We give an example of how to calculate the function $\mathtt{match}$ for a TSO-computation. Let us consider the following TSO-computation

$$\pi_{\mathsf{TSO}} = c_0 \xrightarrow{t_1}_{\mathsf{TSO}} c_1 \xrightarrow{t_2}_{\mathsf{TSO}} c_2 \xrightarrow{t_3}_{\mathsf{TSO}} c_3$$

containing only transitions of a process $p$ with two variables $x$ and $y$ where $c_i = (\mathbf{q}_i, \mathbf{b}_i, \mathbf{mem}_i)$ for all $i : 0 \leq i \leq n = 3$ such that:

$$\begin{array}{llll}
\mathbf{q}_0(p) = q_0, & \mathbf{b}_0(p) = \epsilon, & \mathbf{mem}_0(x) = 0, \mathbf{mem}_0(y) = 0, & t_1 = (q_0, \mathsf{w}(x,1), q_1)\,, \\
\mathbf{q}_1(p) = q_1, & \mathbf{b}_1(p) = (x,1), & \mathbf{mem}_1(x) = 0, \mathbf{mem}_1(y) = 0, & t_2 = \mathsf{update}_p, \\
\mathbf{q}_2(p) = q_1, & \mathbf{b}_2(p) = \epsilon, & \mathbf{mem}_2(x) = 1, \mathbf{mem}_2(y) = 0, & t_3 = (q_1, \mathsf{r}(y,0), q_2)\,, \\
\mathbf{q}_3(p) = q_2, & \mathbf{b}_3(p) = \epsilon, & \mathbf{mem}_3(x) = 1, \mathbf{mem}_3(y) = 0. &
\end{array}$$

Following the above definitions of $I$ and $I'$, $I = i_1 = 2$ (hence, $m = 1$) is the maximal sequence of indices of all update or atomic read-write transitions in $\pi_{\mathsf{TSO}}$. In a similar way, $I' = i'_1 = 1$ is the maximal sequence of indices of all write or atomic read-write transitions in $\pi_{\mathsf{TSO}}$. We note that $t_{i_1} = t_2$ is an update transition, and $t'_{i'_1} = t_1$ is a write transition. Since the TSO-computation contains only transition of the process $p$, it follows that $I = I_p$ and $I' = I'_p$. Following the above definition of $\mathtt{match}$, with $m = 1$ and $n = 3$, we have $\mathtt{match}\,(i_1) = \mathtt{match}\,(2) = 1$. △

For every $j : 1 \leq j \leq n$ such that $t_j \in \Delta_p^{\mathsf{r}}$ is a read transition of process $p$, we define $\mathsf{fromMem}(t_j)$ as a predicate such that $\mathsf{fromMem}(t_j)$ holds if and only if $(x, v') \notin \mathtt{buffers}\,(c_{j-1})$ for all $v' \in \mathbb{V}$.

For every $j : 1 \leq j \leq n$ and $p \in \mathbb{P}$, we define the function $\mathsf{label}_p(j)$ as follows:

(1) $\mathsf{label}_p(j) := (x, v)$ if $t_j \in \Delta_p^r$ is of the form $(q, \mathsf{r}(x,v), q')$ and $\mathsf{fromMem}(t_j)$ holds.
(2) $\mathsf{label}_p(j) := (x, v, \textsc{own})$ if $t_j = \mathsf{update}_p$ and $\mathtt{match}\,(j) = l$ with $t_l$ of the form $(q, \mathsf{w}(x,v), q')$.
(3) $\mathsf{label}_p(j) := \epsilon$ otherwise.

Given a sequence $\ell_1 \cdots \ell_k$ with $k \geq 1$ and $1 \leq \ell_i \leq n$ for all $i : 1 \leq i \leq k$, we define $\mathsf{label}_p(\ell_1 \cdots \ell_k) := \mathsf{label}_p(\ell_1) \cdots \mathsf{label}_p(\ell_{k-1}) \cdot \mathsf{label}_p(\ell_k)$. Let $\mathsf{label}_p^{\mathsf{rev}}(\ell_1 \cdots \ell_k)$ with $k \geq 1$ and $1 \leq \ell_i \leq n$ for all $i : 1 \leq i \leq k$ be the reversed string of $\mathsf{label}_p(\ell_1 \cdots \ell_k)$, i.e. $\mathsf{label}_p^{\mathsf{rev}}(\ell_1 \cdots \ell_k) := \mathsf{label}_p(\ell_k) \cdot \mathsf{label}_p(\ell_{k-1}) \cdots \mathsf{label}_p(\ell_1)$.

**Example A.15.** In the following, we give an example of how to calculate the functions $\mathtt{fromMem}$ and $\mathtt{label}$ for the TSO-computation $\pi_{\mathsf{TSO}}$ given in Example A.14. We recall that $n = 3$ and the function $\mathtt{match}$ is given in Example A.14. We also note that $t_3$ is the only read transition in $\pi_{\mathsf{TSO}}$. Following the above definition of $\mathtt{fromMem}$, we have that $\mathtt{fromMem}(t_3)$ holds. Then following the definition of $\mathtt{match}$, for every $j : 1 \leq j \leq n = 3$, we define the function $\mathtt{label}_p(j)$ as follows:

$$\mathtt{label}_p(1) = \epsilon, \qquad \mathtt{label}_p(2) = (x, 1, own), \qquad \mathtt{label}_p(3) = (y, 0). \qquad \triangle$$

Below we show how to simulate all transitions of the TSO-computation $\pi_{\mathsf{TSO}}$ by a set of corresponding transitions in the DTSO-computation $\pi_{\mathsf{DTSO}}$. The idea is to divide the DTSO-computation to $m + 1$ phases. For $0 \leq r < m$, each phase $r$ will end at the configuration $d_{r+1}$ by the simulation of the transition $t_{\mathtt{match}(i_{r+1})}$ in $\pi_{\mathsf{TSO}}$. Moreover, in phase $r : 0 \leq r < m$, we call the process $\mathtt{proc}\,(r+1)$ as the *active* process, and other processes as the *inactive* ones. We execute only the DTSO-transitions of the active process $p = \mathtt{proc}\,(r+1)$ in its active phases. For other processes $p' \neq p$, we only change the content of their buffers in the active phases of $p$. In the final phase $r = m$, all processes will be considered to be active because the index $i_{m+1}$ is not defined in the definition of the sequence $I$. The DTSO-computation $\pi_{\mathsf{DTSO}}$ will end at the configuration $d_{m+1}$.

For every $r : -1 \leq r < m$ and $p \in \mathbb{P}$, we define the function $\mathtt{pos}\,(r, p)$ in an inductive way on $r$:

(1) $\mathtt{pos}\,(-1, p) := 0$ for all $p \in \mathbb{P}$.
(2) $\mathtt{pos}\,(r, p) := \mathtt{pos}\,(r - 1, p)$ for all $p \neq \mathtt{proc}\,(r + 1)$ and $0 \leq r < m$.
(3) $\mathtt{pos}\,(r, p) := \mathtt{match}\,(i_{r+1})$ for $p = \mathtt{proc}\,(r + 1)$ and $0 \leq r < m$.

In other words, the function $\mathtt{pos}\,(r, p)$ is the index of the last simulated transition by process $p$ at the end of phase $r$ in the computation $\pi_{\mathsf{TSO}}$. Moreover, we use $\mathtt{pos}\,(-1, p)$ to be the index of the starting transition of process $p$ before phase 0.

**Example A.16.** In the following, we give an example of how to calculate the function $\mathtt{pos}$ for the TSO-computation $\pi_{\mathsf{TSO}}$ given in Example A.14. We recall that $m = 1$ and $\pi_{\mathsf{TSO}}$ contains only transitions of the process $p$. We also recall that the function $\mathtt{match}$ is given in Example A.15. Following the above definition of $\mathtt{pos}$, for every $r : -1 \leq r < m = 1$, we define the function $\mathtt{pos}\,(r, p)$ as follows:

$$\mathtt{pos}\,(-1, p) = 0, \qquad\qquad \mathtt{pos}\,(0, p) = 1. \qquad \triangle$$

Let $d_0 = c^D_{init} = (\mathbf{q}_{init}, \mathbf{b}_{init}, \mathbf{mem}_{init})$. We define the sequence of DTSO-configurations $d_1, \ldots, d_m, d_{m+1}$ by defining their local states, buffer contents, and memory states as follows:

(1) For every configuration $d_{r+1}$ where $0 \leq r < m$:
- $\mathtt{states}\,(d_{r+1})\,(p) := \mathtt{states}\,\left(c_{\mathtt{pos}(r,p)}\right)(p)$,
- $\mathtt{mem}\,(d_{r+1}) := \mathtt{mem}\,\left(c_{i_{r+1}}\right)$,
- $\mathtt{buffers}\,(d_{r+1})\,(p) := \mathsf{label}^{\mathsf{rev}}_p(\mathtt{pos}\,(r, p) + 1 \cdots i_{r+1})$.

(2) For the final configuration $d_{m+1}$:
- $\mathtt{states}\,(d_{m+1})\,(p) := \mathtt{states}\,(c_n)\,(p)$,
- $\mathtt{mem}\,(d_{m+1}) := \mathtt{mem}\,(c_n)$,
- $\mathtt{buffers}\,(d_{m+1})\,(p) := \epsilon$.

**Example A.17.** In the following, we give an example of how to calculate the sequence of configurations $d_1, \ldots, d_m, d_{m+1}$ that will appear in the constructed DTSO-computation $\pi_{\mathsf{DTSO}}$ from the TSO-computation $\pi_{\mathsf{TSO}}$ given in Figure A.14. We recall that $m = 1$, $n = 3$, and the TSO-computation $\pi_{\mathsf{TSO}}$ contains only transitions of the process $p$. We also recall that the functions $\mathtt{label}$ and $\mathtt{pos}$ are given in Example A.15 and Example A.16, respectively.

The DTSO-computation $\pi_{\mathsf{DTSO}}$ will consist of $m + 1 = 2$ phases, referred as the phase 0 and the phase 1. For each $r : 0 \leq r \leq m + 1 = 2$, we define the DTSO-configuration $d_r = (\mathbf{q}'_r, \mathbf{b}'_r, \mathbf{mem}'_r)$ based on the TSO-configurations that are appearing in $\pi_{\mathsf{TSO}}$ as follows:

$$\begin{array}{llll}
d_0 : & \mathbf{q}'_0(p) = q_0, & \mathbf{b}'_0(p) = \epsilon, & \mathbf{mem}'_0(x) = 0, \mathbf{mem}'_0(y) = 0, \\
d_1 : & \mathbf{q}'_1(p) = q_1, & \mathbf{b}'_1(p) = (x, 1, own), & \mathbf{mem}'_1(x) = 1, \mathbf{mem}'_1(y) = 0, \\
d_2 : & \mathbf{q}'_2(p) = q_2, & \mathbf{b}'_1(p) = \epsilon, & \mathbf{mem}'_2(x) = 1, \mathbf{mem}'_2(y) = 0.
\end{array}$$

Finally, we construct the DTSO-computation as follows:

$$\pi_{\mathsf{DTSO}} = d_0 \xrightarrow{t'_1}_{\mathsf{DTSO}} d_1 \xrightarrow{t'_2}_{\mathsf{DTSO}} d_{12} \xrightarrow{t'_3}_{\mathsf{DTSO}} d_{13} \xrightarrow{t'_4}_{\mathsf{DTSO}} d_{14} \xrightarrow{t'_5}_{\mathsf{DTSO}} d_2$$

where $d_{12} = (\mathbf{q}'_{12}, \mathbf{b}'_{12}, \mathbf{mem}_{12})$, $d_{13} = (\mathbf{q}_{13}, \mathbf{b}'_{13}, \mathbf{mem}_{13})$, $d_{14} = (\mathbf{q}'_{14}, \mathbf{b}'_{14}, \mathbf{mem}_{14})$, $t'_1 = (q_0, \mathsf{w}(x,1), q_1)$, $t'_2 = \mathsf{propagate}_p^y$, $t'_3 = \mathsf{delete}_p$ $t'_4 = (q_1, \mathsf{r}(y,0), q_2)$, $t'_5 = \mathsf{delete}_p$, and:

$$
\begin{array}{llll}
d_{12}: & \mathbf{q}'_{12}(p) = q_1, & \mathbf{b}'_{12}(p) = (y,0)\cdot(x,1,own), & \mathbf{mem}'_{12}(x) = 1, \mathbf{mem}'_{12}(y) = 0,\\
d_{13}: & \mathbf{q}'_{13}(p) = q_1, & \mathbf{b}'_{13}(p) = (y,0), & \mathbf{mem}'_{13}(x) = 1, \mathbf{mem}'_{13}(y) = 0,\\
d_{14}: & \mathbf{q}'_{14}(p) = q_2, & \mathbf{b}'_{14}(p) = (y,0), & \mathbf{mem}'_{14}(x) = 1, \mathbf{mem}'_{14}(y) = 0.
\end{array}
$$

Since there is only one update transition in $\pi_{\mathsf{DTSO}}$ and $\pi_{\mathsf{TSO}}$, it is easy to see that $\pi_{\mathsf{TSO}}$ has the same sequence of memory updates as $\pi_{\mathsf{DTSO}}$. It is also easy to see that $d_0 = c^D_{init}$ and $d_3 = (\mathtt{states}\,(c_3)\,, \mathbf{b}, \mathtt{mem}\,(c_3))$ where $\mathbf{b}(p) := \epsilon$. Therefore $\pi_{\mathsf{DTSO}}$ is a witness of the construction. △

Lemma A.18 shows the existence of a DTSO-computation $\pi_{\mathsf{DTSO}}$ that starts from the initial TSO-configuration and whose target has the same local state definitions as the target $c_n$ of the TSO-computation $\pi_{\mathsf{TSO}}$. The only if direction of Theorem 3.4 will follow directly from Lemma A.18. This concludes the proof of the only if direction of Theorem 3.4.

**Lemma A.18.** *The following properties hold for the constructed sequence $d_1, \ldots, d_m, d_{m+1}$:*

- *For every $r : 0 \le r < m$, $d_r \xrightarrow{*}_{\mathsf{DTSO}} d_{r+1}$,*
- *$d_m \xrightarrow{*}_{\mathsf{DTSO}} d_{m+1}$.*

*Proof.* We show the proof of the lemma follows directly Lemma A.19 and Lemma A.23. To make the proof understandable, below we consider a $\mathsf{fence}$ transition $t = (q, \mathsf{fence}, q')$ such that $c \xrightarrow{t}_{\mathsf{TSO}} c'$ for some $c, c'$ as an atomic read-write transition of the form $(q, \mathsf{arw}(x,v,v), q')$ where $v \in \mathbb{V}$ is the memory value of variable $x \in \mathbb{X}$ in $c$. For a given TSO-computation $\pi_{\mathsf{TSO}}$, we can calculate such value $v$ for each $\mathsf{fence}$ transition $\pi_{\mathsf{TSO}}$. □

**Lemma A.19.** *If $0 \le r < m$, then $d_r \xrightarrow{*}_{\mathsf{DTSO}} d_{r+1}$.*

*Proof.* We are in phase $r$. Because from the configuration $d_r$, the memory has not been changed until the transition $t_{i_{r+1}}$, we observe that all memory-read transitions of the process $p$ between transitions $t_{i_r}$ and $t_{i_{r+1}}$ will get values from $\mathtt{mem}\,(d_r)$ where $p \in \mathbb{P}$. Therefore, we can execute a sequence of propagation transitions to propagate from the memory to the buffer of process $p$ to full fill it by all messages that will satisfy all memory-read transitions of $p$ between $t_{i_r}$ and $t_{i_{r+1}}$. We propagate to processes according to the order $\prec$: first to the process $p_{min}$ and last to the process $p_{max}$. We have the following sequence: $d_r \xrightarrow{(\Delta^{\mathsf{propagate}})^*}_{\mathsf{DTSO}} d_r^{p_{min}} \cdots \xrightarrow{(\Delta^{\mathsf{propagate}})^*}_{\mathsf{DTSO}} d_r^{p_{max}}$. The shape of the configuration $d_r^{p_{max}}$ is:

- $\mathtt{states}\,(d_r^{p_{max}})\,(p) = \mathtt{states}\,\left(c_{\mathtt{pos}(r-1,p)}\right)(p)$,
- $\mathtt{mem}\,(d_r^{p_{max}}) = \mathtt{mem}\,(c_{i_r})$,
- $\mathtt{buffers}\,(d_r^{p_{max}})\,(p) = \mathsf{label}_p^{\mathsf{rev}}(\mathtt{pos}\,(r-1,p) + 1 \cdots i_{r+1} - 1)$.

Below let $p = \mathtt{proc}\,(r+1)$ be the active process in phase $r$ of the DTSO-computation. For each transition $t$ in the sequence of transitions (including updates) of the active process, $seq{=}(t_{\mathtt{pos}(r-1,p)+1} \cdots t_{\mathtt{match}(i_{r+1})})|_{\Delta_{\mathsf{proc}(r+1)} \cup \left\{\mathsf{update}_{\mathsf{proc}(r+1)}\right\}}$, we execute a set of transitions in the DTSO-computation as follows:

- To simulate a memory-read transition, we execute the same read transition. And then we execute a delete transition to delete the oldest message in the buffer of $\mathtt{proc}\,(r+1)$.
- To simulate a read-own-write transition, we execute the same read transition.

- To simulate a write transition, we execute the same write transition. This transition must be the transition $t_{\mathtt{match}(i_{r+1})}$. According to Dual TSO semantics, we add an own-message to the buffer of $\mathtt{proc}\,(r+1)$.
- To simulate an $\mathsf{arw}$ transition, we execute the same atomic read-write transition. This transition must be the transition $t_{\mathtt{match}(i_{r+1})}$ and $\mathtt{match}\,(i_{r+1}) = i_{r+1}$.
- To simulate an update transition, we execute a delete transition to delete the oldest message in the buffer of $\mathtt{proc}\,(r+1)$.
- To simulate a $\mathsf{nop}$ transition, we execute the same transitions in the DTSO-computation.

Let $\beta(r,l)$ indicate the index in the TSO-computation of the $l^{th}$ transition in the sequence $seq$ where $1 \leq l \leq |seq|$. Formally, we define $\beta(r,l) := j$ where $1 \leq j \leq n$, $t_j \in (\Delta_p \cup \{\mathsf{update}_p\})$ and $seq(l) = t_j$. Let configuration $d_{r,l}$ where $0 \leq r < m$ be the DTSO-configuration *before* simulating the transition with the index $\beta(r,l)$. We define $d_{r,l}$ by defining its local states, buffer contents, and memory state:

- $\mathtt{states}\,(d_{r,l})\,(p) = \mathtt{states}\,(c_{\mathtt{pos}(r-1,p)})\,(p)$ for all inactive process $p$ and all $l : 1 \leq l \leq |seq|$,
- $\mathtt{states}\,(d_{r,l})\,(p) = \mathtt{states}\,(c_{\beta(r,l)-1})\,(p)$ for the active process $p$ and all $l : 1 \leq l \leq |seq|$,
- $\mathtt{mem}\,(d_{r,l}) = \mathtt{mem}\,(c_{i_r})$ for the active process $p$ and all $l : 1 \leq l \leq |seq|$,
- $\mathtt{buffers}\,(d_{r,l})\,(p) = \mathsf{label}_p^{\mathsf{rev}}(\mathtt{pos}\,(r-1,p)+1 \cdots i_{r+1}-1)$ for all inactive process $p$ and all $l : 1 \leq l \leq |seq|$,
- $\mathtt{buffers}\,(d_{r,l})\,(p) = \mathsf{label}_p^{\mathsf{rev}}(\beta(r,l) \cdots i_{r+1}-1)$ for the active process $p$ and all $l : 1 \leq l \leq |seq|$.

The Lemma A.20, Lemma A.22, and Lemma A.21 imply the result. More precisely, it shows the existence of a DTSO-computation that starts from the DTSO-configuration $d_r^{pmax}$ and whose target is the configuration $d_{r+1}$. This concludes the proof of Lemma A.19. □

**Lemma A.20.** *$d_{r,1} = d_r^{p_{max}}$ for $0 \leq r < m$.*

*Proof.* We show that $d_{r,1}$ and $d_r^{p_{max}}$ have the same local states, memory, and buffer contents. We consider two cases for the active and inactive processes.

- For inactive process $p \neq \mathtt{proc}\,(r+1)$, it is easy to see that:
  - $\mathtt{states}\,(d_{r,1})\,(p) = \mathtt{states}\,(c_{\mathtt{pos}(r-1,p)})\,(p) = \mathtt{states}\,(d_r^{pmax})\,(p)$ by the definitions of configurations $d_{r,1}$ and $d_r^{pmax}$.
  - $\mathtt{buffers}\,(d_{r,1})\,(p) = \mathsf{label}_p^{\mathsf{rev}}(\mathtt{pos}\,(r-1,p)+1 \cdots i_{r+1}-1) = \mathtt{buffers}\,(d_r^{pmax})\,(p)$ by the definitions of configurations $d_{r,1}$ and $d_r^{pmax}$.
- For the active process $p = \mathtt{proc}\,(r+1)$:
  - $\mathtt{states}\,(d_{r,1})\,(p) = \mathtt{states}\,(c_{\beta(r,1)-1})\,(p) = \mathtt{states}\,(c_{\mathtt{pos}(r-1,p)})\,(p)$ by the definition of $\beta(r,1)$. Therefore $\mathtt{states}\,(d_{r,1})\,(p) = \mathtt{states}\,(d_r^{pmax})\,(p)$.
  - $\mathtt{buffers}\,(d_{r,1})\,(p) = \mathsf{label}_p^{\mathsf{rev}}(\beta(r,1) \cdots i_{r+1}-1) = \mathsf{label}_p^{\mathsf{rev}}(\mathtt{pos}\,(r-1,p)+1 \cdots i_{r+1}-1)$ by the definition of $\beta(r,1)$. Therefore $\mathtt{buffers}\,(d_{r,1})\,(p) = \mathtt{buffers}\,(d_r^{pmax})\,(p)$.

In both cases, for the memory, $\mathtt{mem}\,(d_{r,1}) = \mathtt{mem}\,(c_{i_r}) = \mathtt{mem}\,(d_r^{pmax})$ by the definitions of configurations $d_{r,1}$ and $d_r^{pmax}$.

This concludes the proof of Lemma A.20. □

**Lemma A.21.** *$d_{r,|seq|} \xrightarrow{t_{\mathtt{match}(i_{r+1})}}_{\mathsf{DTSO}} d_{r+1}$ for $0 \leq r < m$.*

*Proof.* To prove the lemma, we will show the following properties:

(1) $\exists d'_{r+1} : d_{r,|seq|} \xrightarrow{t_{\mathtt{match}(i_{r+1})}}_{\mathsf{DTSO}} d'_{r+1}$, i.e. the transition $t_{\mathtt{match}(i_{r+1})}$ is feasible from the configuration $d_{r,|seq|}$.

(2) Moreover, $d'_{r+1}$=$d_{r+1}$.

Let $p = \mathtt{proc}\,(r+1)$ be the active process. We show the property (1) by considering two cases:

- $\mathtt{match}\,(i_{r+1})$ is a write transition: By simulation, we execute the same transition in the DTSO-computation. It is feasible since $\mathtt{states}\left(d_{r,|seq|}\right)(p) = \mathtt{states}\left(c_{\beta(r,|seq|)-1}\right)(p) = \mathtt{states}\left(c_{\mathtt{match}(i_{r+1})-1}\right)(p)$ by the definitions of $\beta(r, |seq|)$ and $d_{r,|seq|}$. This concludes the property (1).
- $\mathtt{match}\,(i_{r+1})$ is an atomic read-write transition: We notice that $\mathtt{match}\,(i_{r+1}) = i_{r+1}$. It is feasible since $\mathtt{states}\left(d_{r,|seq|}\right)(p) = \mathtt{states}\left(c_{\beta(r,l)-1}\right)(p) = \mathtt{states}\left(c_{\mathtt{match}(i_{r+1})-1}\right)(p)$, $\mathtt{mem}\left(d_{r,|seq|}\right) = \mathtt{mem}\,(c_{i_r})$, and
$$\mathtt{buffers}\left(d_{r,|seq|}\right)(p) = \mathtt{buffers}\left(c_{\beta(r,l)-1}\right)(p) = \mathtt{buffers}\left(c_{\mathtt{match}(i_{r+1})-1}\right)(p) = \epsilon$$
by the definitions of $\beta(r, |seq|)$ and $d_{r,|seq|}$. This concludes the property (1).

We show the property (2) by showing that $d'_{r+1}$ and $d_{r+1}$ have the same local states, memory, and buffer contents. Recall that the $t_{\mathtt{match}(i_{r+1})}$ can be a write transition or an atomic read-write transition.

We consider inactive processes. For an inactive process $p \neq \mathtt{proc}\,(r+1)$, we have:

- Since the transition $t_{\mathtt{match}(i_{r+1})}$ is of the active process, we have $\mathtt{states}\left(d'_{r+1}\right)(p) = \mathtt{states}\,(d_r^{pmax})\,(p)$. Moreover, by the definition of $d_r^{pmax}$, we see that $\mathtt{states}\,(d_r^{pmax})\,(p) = \mathtt{states}\left(c_{\mathtt{pos}(r-1,p)}\right)(p)$ . Hence, by the definition of $d_{r+1}$,
$$\mathtt{states}\left(d'_{r+1}\right)(p) = \mathtt{states}\,(d_{r+1})\,(p)\,.$$
- Since the transition $t_{\mathtt{match}(i_{r+1})}$ is of the active process, we have $\mathtt{buffers}\left(d'_{r+1}\right)(p) = \mathtt{buffers}\,(d_r^{pmax})\,(p)$. Moreover, by the definition of $d_r^{pmax}$, we have $\mathtt{buffers}\,(d_r^{pmax})\,(p) = \mathsf{label}_p^{\mathsf{rev}}(\mathtt{pos}\,(r-1,p)+1\cdots i_{r+1}-1)$. Hence, by the definition of $d_{r+1}$,
$$\mathtt{buffers}\left(d'_{r+1}\right)(p)) = \mathtt{buffers}\,(d_{r+1})\,(p)\,.$$

We consider the active process $p = \mathtt{proc}\,(r+1)$ for the case that the transition $t_{\mathtt{match}(i_{r+1})}$ is a write one. By executing the same transition, we add an owing message to the buffer of process $p$ and change the memory.

- Since the transition $t_{\mathtt{match}(i_{r+1})}$ is of the active process, we have
$$\mathtt{states}\left(d'_{r+1}\right)(p) = \mathtt{states}\left(c_{\beta(r,l)}\right)(p)\,.$$
Moreover, it follows from the fact $\beta(r,l) = \mathtt{match}\,(i_{r+1})$ and the definition of $\mathtt{pos}\,(r,p)$ that $\mathtt{states}\left(c_{\beta(r,l)}\right)(p) = \mathtt{states}\left(c_{\mathtt{match}(i_{r+1})}\right)(p) = \mathtt{states}\left(c_{\mathtt{pos}(r,p)}\right)(p)$. Hence, it follows by the definition of $d_{r+1}$ that $\mathtt{states}\left(d'_{r+1}\right)(p) = \mathtt{states}\,(d_{r+1})\,(p)$.
- Since the transition $t_{\mathtt{match}(i_{r+1})}$ is of the active process, we have $\mathtt{buffers}\left(d'_{r+1}\right)(p) = \mathsf{label}_p^{\mathsf{rev}}(i_{r+1}) \cdot \mathtt{buffers}\left(d_{r,|seq|}\right)(p)$. Then, it follows from the definition of $d_{r,|seq|}$ that $\mathtt{buffers}\left(d'_{r+1}\right)(p) = \mathsf{label}_p^{\mathsf{rev}}(i_{r+1}) \cdot \mathsf{label}_p^{\mathsf{rev}}(\beta(r,|seq|)\cdots i_{r+1}-1) = \mathsf{label}_p^{\mathsf{rev}}(i_{r+1}) \cdot \mathsf{label}_p^{\mathsf{rev}}(\mathtt{match}\,(i_{r+1})\cdots i_{r+1}-1) = \mathsf{label}_p^{\mathsf{rev}}(\mathtt{match}\,(i_{r+1})\cdots i_{r+1}) = \mathsf{label}_p^{\mathsf{rev}}(\mathtt{pos}\,(r,p)\cdots i_{r+1}) = \mathsf{label}_p^{\mathsf{rev}}(\mathtt{pos}\,(r,p)+1\cdots i_{r+1})$. Hence, it follows by the definition of $d_{r+1}$ that $\mathtt{buffers}\left(d'_{r+1}\right)(p) = \mathtt{buffers}\,(d_{r+1})\,(p)$.

We consider the active process $p = \mathtt{proc}\,(r+1)$ for the case that the transition $t_{\mathtt{match}(i_{r+1})}$ is an atomic read-write one. By simulation, we execute the same transition and change the memory.

- Since the transition $t_{\mathtt{match}(i_{r+1})}$ is of the active process, we have $\mathtt{states}\left(d'_{r+1}\right)(p) = \mathtt{states}\left(c_{\beta(r,l)}\right)(p)$. Moreover, it follows from the fact $\beta(r,l) = \mathtt{match}\,(i_{r+1})$ and the definition of $\mathtt{pos}\,(r,p)$ that $\mathtt{states}\left(c_{\beta(r,l)}\right)(p) = \mathtt{states}\left(c_{\mathtt{match}(i_{r+1})}\right)(p) = \mathtt{states}\left(c_{\mathtt{pos}(r,p)}\right)(p)$. Hence, it follows by the definition of $d_{r+1}$ that $\mathtt{states}\left(d'_{r+1}\right)(p) = \mathtt{states}\,(d_{r+1})\,(p)$.
- Since the transition $t_{\mathtt{match}(i_{r+1})}$ is of the active process, we have $\mathtt{buffers}\left(d'_{r+1}\right)(p) = \epsilon$. From the definitions of $d_{r+1}$ and $\mathtt{pos}\,(r,p)$ and the fact $\mathtt{match}\,(i_{r+1}) = i_{r+1}$, we have $\mathtt{buffers}\,(d_{r+1})\,(p) = \mathsf{label}_p^{\mathsf{rev}}(\mathtt{pos}\,(r,p) + 1 \cdots i_{r+1}) = \mathsf{label}_p^{\mathsf{rev}}(\mathtt{match}\,(i_{r+1}) + 1 \cdots i_{r+1}) = \mathsf{label}_p^{\mathsf{rev}}(i_{r+1} + 1 \cdots i_{r+1}) = \epsilon$. Hene, it follows that $\mathtt{buffers}\left(d'_{r+1}\right)(p) = \mathtt{buffers}\,(d_{r+1})\,(p)$.

For both cases, for the memory, we have $\mathtt{mem}\left(d'_{r+1}\right)(p) = \mathtt{mem}\left(c_{i_{r+1}}\right) = \mathtt{mem}\,(d_{r+1})$ from the fact that we change the memory by transition $t_{\mathtt{match}(i_{r+1})}$ and by the definition of $d_{r+1}$. Finally, we have $d'_{r+1} = d_{r+1}$.

This concludes the proof of Lemma A.21. □

**Lemma A.22.** $d_{r,l} \xrightarrow{*}_{\mathsf{DTSO}} d_{r,l+1}$ *for* $0 \leq r < m$, $1 \leq l < |seq|$.

*Proof.* The transition $t_{\beta(r,l)}$ can be a read-from-memory, read-own-write, nop, update one. First, we give our simulation of the transition $t_{\beta(r,l)}$ from the configuration $d_{r,l}$ and show that this simulation is feasible. We consider different types of the transition $t_{\beta(r,l)}$. Let process $p = \mathtt{proc}\,(r+1)$ is the active process.

- $t_{\beta(r,l)}$ is a read-from-memory transition: By simulation, we execute the same transition in the DTSO-computation. Note that under the DTSO semantics, this transition will read an element in the buffers. Then we delete the oldest element in the buffer of the active process. The transition $t_{\beta(r,l)}$ is feasible because by the definition of $d_{r,l}$, we have $\mathtt{states}\,(d_{r,l})\,(p) = \mathtt{states}\left(c_{\beta(r,l)-1}\right)(p)$ and $\mathtt{buffers}\,(d_{r,l})\,(p) = \mathsf{label}_p^{\mathsf{rev}}(\beta(r,l) \cdots i_{r+1} - 1)$.
- $t_{\beta(r,l)}$ is a nop transition: By simulation, we execute the same transition in the DTSO-computation. The nop transition is feasible because by the definition of $d_{r,l}$, we have $\mathtt{states}\,(d_{r,l})\,(p) = \mathtt{states}\left(c_{\beta(r,l)-1}\right)(p)$.
- $t_{\beta(r,l)}$ is a read-own-write read transition: By simulation, we execute the same transition in the DTSO-computation. Observe that $\mathtt{states}\,(d_{r,l})\,(p) = \mathtt{states}\left(c_{\beta(r,l)-1}\right)(p)$. We show the read-own-write transition is feasible in the DTSO-computation. In the TSO-computation, this read must get its value from a write transition $t'_1 \in \Delta_p^{\mathsf{w}}$ that has the corresponding update transition $t'_2 \in \Delta_p^{\mathsf{update}}$. According to the TSO semantics, the write comes and goes out the buffer in FIFO order. We have the order of these transitions in the TSO-computation: (i) transition $t_{\beta(r,l)}$ is between transitions $t'_1$ and $t_{\mathtt{match}(i_{r+1})}$, and (ii) transition $t'_2$ is between transitions $t_{\beta(r,l)}$ and $t_{\mathtt{match}(i_{r+1})}$. Moreover, (iii) there is no other write transition of the same process and the same variable between transitions $t'_1$ and $t_{\beta(r,l)}$. In the simulation of the DTSO-computation, when we meet the transition $t'_1$ we put an own-message $m$ to the buffer of the active process. From that we do not meet any write transition to the same variable of the active process until the simulation of transition $t_{\beta(r,l)}$. Moreover, the message $m$ exists in the buffer until the simulation of transition $t_{\beta(r,l)}$ because the update transition $t'_2$ is after the transition $t_{\beta(r,l)}$. Therefore the message $m$ is the newest own-message in the buffer that can match to the read $t_{\beta(r,l)}$. In other words, the read transition $t_{\beta(r,l)}$ is feasible.
- $t_{\beta(r,l)}$ is an update transition: By simulation, we delete the oldest own-message in the buffer of the active process in the DTSO-computation. This transition is feasible because by the

definition of $d_{r,l}$, we have $\texttt{states}\left(d_{r,l}\right)(p) = \texttt{states}\left(c_{\beta(r,l)-1}\right)(p)$ and $\texttt{buffers}\left(d_{r,l}\right)(p) = \mathsf{label}^{\mathsf{rev}}_p(\beta(r,l)\cdots i_{r+1}-1)$.

We have show our simulation of the transition $t_{\beta(r,l)}$ in the DTSO-computation is feasible. Let $d'_{r,l+1}$ be the configuration in the DTSO-computation after the simulation. We proceed the proof of the lemma by proving that $d'_{r,l+1}{=}d_{r,l+1}$. To do this, we will show that $d'_{r,l+1}$ and $d_{r,l+1}$ have the same local states, memory, and buffer contents.

We consider inactive processes. For an inactive process $p \neq \texttt{proc}\,(r+1)$, we have:

- Since in the simulation, we only execute the transition of the active process, we have $\texttt{states}\left(d'_{r,l+1}\right)(p) = \texttt{states}\left(d^{pmax}_r\right)(p)$. Moreover, by the definition of $d^{pmax}_r$, we see that $\texttt{states}\left(d^{pmax}_r\right)(p) = \texttt{states}\left(c_{\texttt{pos}(r-1,p)}\right)(p)$ . Hence, it follows by the definition of $d_{r,l+1}$ that $\texttt{states}\left(d'_{r,l+1}\right)(p) = \texttt{states}\left(d_{r,l+1}\right)(p)$.
- Since in the simulation, we only execute the transition of the active process, we have $\texttt{buffers}\left(d'_{r,l+1}\right)(p) = \texttt{buffers}\left(d^{pmax}_r\right)(p)$. Moreover, by the definition of $d^{pmax}_r$, we see that $\texttt{buffers}\left(d^{pmax}_r\right)(p) = \mathsf{label}^{\mathsf{rev}}_p(\texttt{pos}\,(r-1,p)+1\cdots i_{r+1}-1)$. Hence, it follows by the definition of $d_{r,l+1}$ that $\texttt{buffers}\left(d'_{r+1}\right)(p) = \texttt{buffers}\left(d_{r,l+1}\right)(p)$.

We consider the active process $p \neq \texttt{proc}\,(r+1)$ for the case that the transition $t_{\beta(r,l)}$ is a read-from-memory one. From the simulation of $t_{\beta(r,l)}$, $\texttt{states}\left(d'_{r,l+1}\right)(p) = \texttt{states}\left(c_{\beta(r,l+1)-1}\right)(p)$. We have $\texttt{states}\left(d_{r,l+1}\right)(p) = \texttt{states}\left(c_{\beta(r,l+1)-1}\right)(p)$ from the definition of $d_{r,l+1}$. Furthermore, because we delete the oldest message in the buffer of the process $p$ after we execute the read transition, it follows by the definition of $d_{r,l+1}$ that $\texttt{buffers}\left(d'_{r,l+1}\right)(p) = \mathsf{label}^{\mathsf{rev}}_p(\beta(r,l+1)\cdots i_{r+1}-1) = \texttt{buffers}\left(d_{r,l+1}\right)(p)$. Finally, by the definitions of $d_{r,l}$ and $d_{r,l+1}$, we have $\texttt{mem}\left(d'_{r,l+1}\right) = \texttt{mem}\left(d_{r,l}\right) = \texttt{mem}\left(c_{i_r}\right) = \texttt{mem}\left(d_{r,l+1}\right)$. Hence, it follows that $d'_{r,l+1}{=}d_{r,l+1}$.

We consider the active process $p \neq \texttt{proc}\,(r+1)$ for the case that the transition $t_{\beta(r,l)}$ is a nop one. From the simulation of $t_{\beta(r,l)}$, $\texttt{states}\left(d'_{r,l+1}\right)(p) = \texttt{states}\left(c_{\beta(r,l+1)-1}\right)(p)$. From the definition of $d_{r,l+1}$, $\texttt{states}\left(d_{r,l+1}\right)(p) = \texttt{states}\left(c_{\beta(r,l+1)-1}\right)(p)$. From the definitions of $d_{r,l}$ and $d_{r,l+1}$, $\texttt{buffers}\left(d'_{r,l+1}\right)(p) = \texttt{buffers}\left(d_{r,l}\right)(p) = \mathsf{label}^{\mathsf{rev}}_p(\beta(r,l)\cdots i_{r+1}-1) = \mathsf{label}^{\mathsf{rev}}_p(\beta(r,l+1)\cdots i_{r+1}-1) = \texttt{buffers}\left(d_{r,l+1}\right)(p)$. Finally, by the definitions of $d_{r,l}$ and $d_{r,l+1}$, we have $\texttt{mem}\left(d'_{r,l+1}\right) = \texttt{mem}\left(d_{r,l}\right) = \texttt{mem}\left(c_{i_r}\right) = \texttt{mem}\left(d_{r,l+1}\right)$. Hence, it follows that $d'_{r,l+1}{=}d_{r,l+1}$.

We consider the active process $p \neq \texttt{proc}\,(r+1)$ for the case that the transition $t_{\beta(r,l)}$ is a read-own-write one. From the simulation of the transition, we have $\texttt{states}\left(d'_{r,l+1}\right)(p) = \texttt{states}\left(c_{\beta(r,l+1)-1}\right)(p)$. Then from the definition of $d_{r,l+1}$, we have $\texttt{states}\left(d_{r,l+1}\right)(p) = \texttt{states}\left(c_{\beta(r,l+1)-1}\right)(p)$. It follows from the definitions of $d_{r,l}$ and $d_{r,l+1}$ that $\texttt{buffers}\left(d'_{r,l+1}\right)(p) = \texttt{buffers}\left(d_{r,l}\right)(p) = \mathsf{label}^{\mathsf{rev}}_p(\beta(r,l)\cdots i_{r+1}-1) = \mathsf{label}^{\mathsf{rev}}_p(\beta(r,l+1)\cdots i_{r+1}-1) = \texttt{buffers}\left(d_{r,l+1}\right)(p)$. Finally, by the definitions of $d_{r,l}$ and $d_{r,l+1}$, we have $\texttt{mem}\left(d'_{r,l+1}\right) = \texttt{mem}\left(d_{r,l}\right) = \texttt{mem}\left(c_{i_r}\right) = \texttt{mem}\left(d_{r,l+1}\right)$ . Hence, it follows that $d'_{r,l+1}{=}d_{r,l+1}$.

We consider the active process $p \neq \texttt{proc}(r+1)$ for the case that the transition $t_{\beta(r,l)}$ is an update one. From the simulation of the transition, $\texttt{states}\left(d'_{r,l+1}\right)(p) = \texttt{states}\left(c_{\beta(r,l+1)-1}\right)(p)$. We have $\texttt{states}\left(c_{\beta(r,l+1)-1}\right)(p) = \texttt{states}\left(d_{r,l+1}\right)(p)$ from the definition of $d_{r,l+1}$. Moreover, we have $\texttt{buffers}\left(d'_{r,l+1}\right)(p) = \mathsf{label}^{\mathsf{rev}}_p(\beta(r,l+1)\cdots i_{r+1}-1) = \texttt{buffers}\left(d_{r,l+1}\right)(p)$. Futhermore, by the definitions of $d_{r,l}$ and $d_{r,l+1}$, we have $\texttt{mem}\left(d'_{r,l+1}\right) = \texttt{mem}\left(d_{r,l}\right) = \texttt{mem}\left(c_{i_r}\right) = \texttt{mem}\left(d_{r,l+1}\right)$ . Hence, it follows that $d'_{r,l+1}{=}d_{r,l+1}$.

This concludes the proof of Lemma A.22. □

**Lemma A.23.** $d_m \xrightarrow{*}_{\mathsf{DTSO}} d_{m+1}$.

*Proof.* We are in the final phase $r = m$. Observe that in this phase we do not have any write and atomic read-write transitions. Because from the configuration $d_m$ until the end of the TSO-computation the memory has not been changed, we observe that all memory-read transitions of a process $p \in \mathbb{P}$ after transitions $t_{i_m}$ get their values from $\texttt{mem}\left(d_m\right)$. Therefore, we can execute a sequence of propagation transitions to propagate from the memory to buffer of the process $p$ to full fill it by all messages that will satisfy all memory-read transitions of $p$ after $t_{i_m}$. We propagate to processes according to the order $\prec$: first to process $p_{min}$ and last to process $p_{max}$. We have the following sequence: $d_m \xrightarrow{(\Delta^{\mathsf{propagate}})^*}_{\mathsf{DTSO}} d_m^{p_{min}} \cdots \xrightarrow{(\Delta^{\mathsf{propagate}})^*}_{\mathsf{DTSO}} d_m^{p_{max}}$.

Next we simulate the remaining transitions $(t_{\mathsf{pos}(m-1,p)+1}\cdots t_n)|_{\Delta_p \cup \left\{\mathsf{update}_p\right\}}$ for each process $p$ of the TSO-computation $\pi_{\mathsf{TSO}}$ according to the order $\prec$: first process $p_{min}$ and last process $p_{max}$.

- To simulate a memory-read transition, we execute the same read transition. And then we execute a delete transition to delete the oldest message in the buffer of the process $p$.
- To simulate a read-own-write transition, we execute the same read transition.
- To simulate an update transition, we execute a delete transition to delete the oldest message in the buffer of the process $p$.
- To simulate a $\mathsf{nop}$ transition, we execute the same transitions in the DTSO-computation.

Following the same argument as in Lemma A.20, Lemma A.21, and Lemma A.22 we show that all simulations of transitions are feasible. As a consequence, from the configuration $d_m$ we reach the configuration $d_{m+1}$ where for all $p \in \mathbb{P}$: $\texttt{states}\left(d_{m+1}\right)(p) = \texttt{states}\left(c_n\right)(p)$, $\texttt{buffers}\left(d_{m+1}\right)(p){=}\epsilon$, and $\texttt{mem}\left(d_{m+1}\right) = \texttt{mem}\left(c_n\right)$.

This concludes the proof of Lemma A.23. □

## Appendix B. Proof of Lemma 4.2

Let $c_i = (\mathbf{q}_i, \mathbf{b}_i, \mathbf{mem}_i)$ be DTSO-configurations for $i : 1 \leq i \leq 3$. Let us assume that $c_1 \xrightarrow{t}_{\mathsf{DTSO}} c_2$ for some $t \in \Delta_p \cup \left\{\mathsf{propagate}^x_p, \mathsf{delete}_p\right\}$ and $p \in \mathbb{P}$. We will define $c_4 = (\mathbf{q}_4, \mathbf{b}_4, \mathbf{mem}_4)$ such that $c_3 \xrightarrow{*}_{\mathsf{DTSO}} c_4$ and $c_2 \sqsubseteq c_4$. We consider the following cases depending on $t$:

(1) Nop: $t = (q_1, \mathsf{nop}, q_2)$. Define $\mathbf{q}_4 := \mathbf{q}_2$, $\mathbf{b}_4 := \mathbf{b}_3$, and $\mathbf{mem}_4 := \mathbf{mem}_2 = \mathbf{mem}_3 = \mathbf{mem}_1$. We have $c_3 \xrightarrow{t}_{\mathsf{DTSO}} c_4$.

(2) Write to memory: $t = (q, \mathsf{w}(x,v), q')$. Define $\mathbf{q}_4 := \mathbf{q}_2$, $\mathbf{b}_4 := \mathbf{b}_3\left[p \hookleftarrow (x, v, \textsc{own}) \cdot \mathbf{b}_3(p)\right]$, and $\mathbf{mem}_4 := \mathbf{mem}_2$. We have $c_3 \xrightarrow{t}_{\mathsf{DTSO}} c_4$.

(3) Propagate: $t = \mathsf{propagate}_p^x$. Define $\mathbf{q}_4 := \mathbf{q}_2$, $\mathbf{mem}_4 := \mathbf{mem}_2 = \mathbf{mem}_3 = \mathbf{mem}_1$, and $\mathbf{b}_4 := \mathbf{b}_3\,[p \hookleftarrow (x, v) \cdot \mathbf{b}_3(p)]$ where $v = \mathbf{mem}_4(x)$. We have $c_3 \xrightarrow{t}_{\mathsf{DTSO}} c_4$.

(4) Delete: $t = \mathsf{delete}_p$. Define $\mathbf{q}_4 := \mathbf{q}_2$ and $\mathbf{mem}_4 := \mathbf{mem}_2 = \mathbf{mem}_3 = \mathbf{mem}_1$. Define $\mathbf{b}_4$ according to one of the following cases:
- If $\mathbf{b}_1 = \mathbf{b}_2\,[p \hookleftarrow \mathbf{b}_2(p) \cdot (x, v)]$, then define $\mathbf{b}_4 := \mathbf{b}_3$. In other words, we define $c_4 := c_3$.
- If $\mathbf{b}_1 = \mathbf{b}_2\,[p \hookleftarrow \mathbf{b}_2(p) \cdot (x, v, \textsc{own})]$ and $(x, v', \textsc{own}) \in \mathbf{b}_2(p)$ for some $v' \in \mathbb{V}$, then define $\mathbf{b}_4 := \mathbf{b}_3$. In other words, we define $c_4 := c_3$.
- If $\mathbf{b}_1 = \mathbf{b}_2\,[p \hookleftarrow \mathbf{b}_2(p) \cdot (x, v, \textsc{own})]$ and there is no $v' \in \mathbb{V}$ such that $(x, v', \textsc{own}) \in \mathbf{b}_2(p)$. Since $\mathbf{b}_1(p) \sqsubseteq \mathbf{b}_3(p)$, we know that there is an $i$ and therefore a smallest $i$ such that $\mathbf{b}_3(p)(i) = (x, v, \textsc{own})$. Define $\mathbf{b}_4 := \mathbf{b}_3\,[p \hookleftarrow \mathbf{b}_3(p)(1) \cdot \mathbf{b}_3(p)(2) \cdots \mathbf{b}_3(p)(i-1)]$. We can perform the following sequence of transitions $c_3 \xrightarrow{\mathsf{delete}_p}_{\mathsf{DTSO}} c'_1 \xrightarrow{\mathsf{delete}_p}_{\mathsf{DTSO}} c'_2 \cdots \xrightarrow{\mathsf{delete}_p}_{\mathsf{DTSO}} c'_{|\mathbf{b}_3(p)|-i} \xrightarrow{\mathsf{delete}_p}_{\mathsf{DTSO}} c_4$. In other words, we reach the configuration $c_4$ from $c_3$ by first deleting $|\mathbf{b}_3(p)| - i$ messages from the head of $\mathbf{b}_3(p)$.

(5) Read: $t = (q, \mathsf{r}(x, v), q')$. Define $\mathbf{q}_4 := \mathbf{q}_2$, and $\mathbf{mem}_4 := \mathbf{mem}_2 = \mathbf{mem}_3 = \mathbf{mem}_1$. We define $\mathbf{b}_4$ according to one of the following cases:
- Read-own-write: If there is an $i : 1 \le i \le |\mathbf{b}_1(p)|$ such that $\mathbf{b}_1(p)(i) = (x, v, \textsc{own})$, and there are no $j : 1 \le j < i$ and $v' \in \mathbb{V}$ such that $\mathbf{b}_1(p)(j) = (x, v', \textsc{own})$. Since $\mathbf{b}_1(p) \sqsubseteq \mathbf{b}_3(h(p))$, there is an $i' : 1 \le i' \le |\mathbf{b}_3(p)|$ such that $\mathbf{b}_3(p)(i') = (x, v, \textsc{own})$, and there are no $j : 1 \le j < i'$ and $v' \in \mathbb{V}$ such that $\mathbf{b}_3(p)(j) = (x, v', \textsc{own})$. Define $\mathbf{b}_4 := \mathbf{b}_3$. In other words, we define $c_4 := c_3$.
- Read from buffer: If $(x, v', \textsc{own}) \notin \mathbf{b}_1(p)$ for all $v' \in \mathbb{V}$ and $\mathbf{b}_1(p) = w \cdot (x, v)$. Let $i$ be the largest $i : 1 \le i \le |\mathbf{b}_3(p)|$ such that $\mathbf{b}_3(p)(i) = (x, v)$. Since $\mathbf{b}_1(p) \sqsubseteq \mathbf{b}_3(p)$, we know that such index $i$ exists. Define $\mathbf{b}_4 := \mathbf{b}_3\,[p \hookleftarrow \mathbf{b}_3(p)(1) \cdot \mathbf{b}_3(p)(2) \cdots \mathbf{b}_3(p)(i-1)]$. We can perform the following sequence of transitions $c_3 \xrightarrow{\mathsf{delete}_p}_{\mathsf{DTSO}} c'_1 \xrightarrow{\mathsf{delete}_p}_{\mathsf{DTSO}} c'_2 \cdots \xrightarrow{\mathsf{delete}_p}_{\mathsf{DTSO}} c'_{|\mathbf{b}_3(p)|-i} \xrightarrow{\mathsf{delete}_p}_{\mathsf{DTSO}} c_4$. In other words, we reach the configuration $c_4$ from $c_3$ by first deleting $|\mathbf{b}_3(p)| - i$ messages from the head of $\mathbf{b}_3(p)$.

(6) Fence: $t = (q, \mathsf{fence}, q')$. Define $\mathbf{q}_4 := \mathbf{q}_2$, $\mathbf{b}_4 := \epsilon$, and $\mathbf{mem}_4 := \mathbf{mem}_2$. We can perform the following sequence of transitions

$$\alpha_3 \xrightarrow{\mathsf{delete}_p}_{\mathsf{DTSO}} \alpha'_1 \xrightarrow{\mathsf{delete}_p}_{\mathsf{DTSO}} \alpha'_2 \cdots \xrightarrow{\mathsf{delete}_p}_{\mathsf{DTSO}} \alpha'_{|\mathbf{b}_3(p)|} \xrightarrow{t}_{\mathsf{DTSO}} \alpha_4.$$

In other words, we reach the configuration $c_4$ from $c_3$ by first emptying the content of $\mathbf{b}_3(p)$ and then performing $t$.

(7) ARW: $t = (q, \mathsf{arw}(x, v, v'), q')$. Define $\mathbf{q}_4 := \mathbf{q}_2$, $\mathbf{b}_4 := \epsilon$, and $\mathbf{mem}_4 := \mathbf{mem}_2$. We can reach the configuration $c_4$ from $c_3$ in a similar manner to the case of the fence transition.

This concludes the proof of Lemma 4.2. □

## Appendix C. Proof of Lemma 4.3

First we show that the ordering $w \sqsubseteq w'$ is a well-quasi-ordering. It is an immediate consequence of the fact that (i) the sub-word relation is a well-quasi-ordering on finite words [Hig52], and that (ii) the number of own-messages in the form $(x, v, own)$ that should be equal, is finite.

Given two DTSO-configurations $c = (\mathbf{q}, \mathbf{b}, \mathbf{mem})$ and $c' = (\mathbf{q}', \mathbf{b}', \mathbf{mem}')$. We define three orders $\sqsubseteq^{state}$, $\sqsubseteq^{mem}$, and $\sqsubseteq^{buffer}$ over configurations of $\mathsf{C}_{\mathsf{DTSO}}$: $c \sqsubseteq^{state} c'$ iff $\mathbf{q} = \mathbf{q}'$, $c \sqsubseteq^{mem} c'$ iff $\mathbf{mem'} = \mathbf{mem}$, and $c \sqsubseteq^{bufer} c'$ iff $\mathbf{b}(p) \sqsubseteq \mathbf{b}'(p)$ for all process $p \in \mathbb{P}$.

It is easy to see that each one of three orderings is a well-quasi-ordering. It follows that the ordering $\sqsubseteq$ on DTSO-configurations based on $\sqsubseteq^{state}$, $\sqsubseteq^{mem}$, and $\sqsubseteq^{buffer}$ is a well-quasi-ordering.

Since the number of processes, the number of local states, memory content, and the number of own-messages that should be equal are finite, it is decidable whether $c_1 \sqsubseteq c_2$.

This concludes the proof of Lemma 4.3. □

## Appendix D. Proof of Lemma 4.4

Consider a DTSO-configuration $c = (\mathbf{q}, \mathbf{b}, \mathbf{mem})$. Let we recall the definition of $\mathsf{minpre}(\{c\})$: $\mathsf{minpre}(\{c\}) := \mathtt{min}\,(\mathtt{Pre}_{\mathcal{T}}\,(\{c\}\uparrow) \cup \{c\}\uparrow)$. We observe that

$$\mathsf{minpre}(\{c\}) = \mathtt{min}\left(\cup_{t \in \Delta \cup \Delta''} \mathtt{min}\left\{c' |\; c' \xrightarrow{t} c\right\} \cup \{c\}\right).$$

For $t \in \Delta \cup \Delta''$, we select $\mathtt{min}\left\{c' |\; c' \xrightarrow{t} c\right\}$ to be the minimal set of all *finite* DTSO-configurations of the form $c' = (\mathbf{q}', \mathbf{b}', \mathbf{mem}')$ such that one of the following properties is satisfied:

(1) Nop: $t = (q_1, \mathsf{nop}, q_2)$, $\mathbf{q}(p) = q_2$ for some $p \in \mathbb{P}$, $\mathbf{q}' = \mathbf{q}\,[p \hookleftarrow q_1]$, $\mathbf{b}' = \mathbf{b}$, and $\mathbf{mem}' = \mathbf{mem}$.
(2) Write: $t = (q_1, \mathsf{w}(x, v), q_2)$, $\mathbf{q}(p) = q_2$ for some $p \in \mathbb{P}$, $\mathbf{b}(p) = (x, v, \text{OWN}) \cdot w$ for some $w$, $\mathbf{mem}(x) = v$, $\mathbf{mem}'(y) = \mathbf{mem}(y)$ if $y \neq x$, $\mathbf{q}' = \mathbf{q}\,[p \hookleftarrow q_1]$, and one of the following properties is satisfied:
   - $\mathbf{b}' = \mathbf{b}\,[p \hookleftarrow w]$.
   - $\mathbf{b}' = \mathbf{b}\,[p \hookleftarrow w_1 \cdot (x, v', own) \cdot w_2]$ for some $v' \in \mathbb{V}$ where $w_1 \cdot w_2 = w$ and $(x, v'', own) \notin w_1$ for all $v'' \in \mathbb{V}$.
(3) Propagate: $t = \mathsf{propagate}_p^x$ for some $p \in \mathbb{P}$, $\mathbf{mem}(x) = v$, $\mathbf{q}' = \mathbf{q}$, $\mathbf{mem}' = \mathbf{mem}$, $\mathbf{b}(p) = (x, v) \cdot w$ for some $w$, and $\mathbf{b}' = \mathbf{b}\,[p \hookleftarrow w]$.
(4) Read: $t = (q_1, \mathsf{r}(x, v), q_2)$, $\mathbf{q}(p) = q_2$ for some $p \in \mathbb{P}$, $\mathbf{q}' = \mathbf{q}\,[p \hookleftarrow q_1]$, and $\mathbf{mem}' = \mathbf{mem}$, and one of the following conditions is satisfied:
   - Read-own-write: there is an $i : 1 \leq i \leq |\mathbf{b}(p)|$ such that $\mathbf{b}(p)(i) = (x, v, own)$, and there are no $j : 1 \leq j < i$ and $v' \in \mathbb{V}$ such that $\mathbf{b}(p)(j) = (x, v', own)$, and $\mathbf{b}' = \mathbf{b}$.
   - Read from buffer: $(x, v', \text{OWN}) \notin \mathbf{b}(p)$ for all $v' \in \mathbb{V}$, $\mathbf{b}(p) = w \cdot (x, v)$ for some $w$, and $\mathbf{b}' = \mathbf{b}$.
   - Read from buffer: $(x, v', own) \notin \mathbf{b}(p)$ for all $v' \in \mathbb{V}$, $\mathbf{b}(p) \neq w \cdot (x, v)$ for all $w$, and $\mathbf{b}' = \mathbf{b}\,[p \hookleftarrow \mathbf{b}(p) \cdot (x, v)]$.
(5) Fence: $t = (q_1, \mathsf{fence}, q_2)$, $\mathbf{q}(p) = q_2$ for some $p \in \mathbb{P}$, $\mathbf{b}(p) = \epsilon$, $\mathbf{q}' = \mathbf{q}\,[p \hookleftarrow q_1]$, $\mathbf{b}' = \mathbf{b}$, and $\mathbf{mem}' = \mathbf{mem}$.
(6) ARW: $t = (q_1, \mathsf{arw}(x, v, v'), q_2)$, $\mathbf{mem}(x) = v'$, $\mathbf{mem}' = \mathbf{mem}\,[x \hookleftarrow v]$, $\mathbf{q}(p) = q_2$ for some $p \in \mathbb{P}$, $\mathbf{b}(p) = \epsilon$, $\mathbf{q}' = \mathbf{q}\,[p \hookleftarrow q_1]$, $\mathbf{b}' = \mathbf{b}$.
(7) Delete: $t = \mathsf{delete}_p$ for some $p \in \mathbb{P}$, $\mathbf{q}' = \mathbf{q}$, $\mathbf{mem}' = \mathbf{mem}$. Moreover, $(x, v, own) \notin \mathbf{b}(p)$ for some $x \in \mathbb{X}$ and all $v \in \mathbb{V}$, $\mathbf{b}' = \mathbf{b}\,[p \hookleftarrow \mathbf{b}(p) \cdot (x, v', own)]$ for some $v' \in \mathbb{V}$.

This concludes the proof of Lemma 4.4. □

## Appendix E. Proof of Lemma 5.2

Let $\alpha_i = (\mathbb{P}_i, c_i)$ and $c_i = (\mathbf{q}_i, \mathbf{b}_i, \mathbf{mem}_i)$ for $i : 1 \leq i \leq 4$. We show that if $\alpha_1 \xrightarrow{t} \alpha_2$ and $\alpha_1 \trianglelefteq \alpha_3$ for some $t \in \Delta_p \cup \left\{\mathsf{propagate}_p^x, \mathsf{delete}_p\right\}$ and $p \in \mathbb{P}_1$ (note that $\mathbb{P}_1 = \mathbb{P}_2$) then the configuration $\alpha_4$ exists such that $\alpha_3 \rightarrow^* \alpha_4$ and $\alpha_2 \trianglelefteq \alpha_4$. First we define $\mathbb{P}_4{:=}\mathbb{P}_3$. Because of $\alpha_1 \trianglelefteq \alpha_3$, there exists an injection $h : \mathbb{P}_1 \mapsto \mathbb{P}_3$ in the ordering $\alpha_1 \trianglelefteq \alpha_3$. We define an injection $h' : \mathbb{P}_2 \mapsto \mathbb{P}_4$ in the ordering $\alpha_2 \trianglelefteq \alpha_4$ such that $h = h'$. Moreover, for $p \in \mathbb{P}_4$, let $\mathbf{q}_4(p) := \mathbf{q}_2(h'(p))$ if the process $p \in \mathbb{P}_2$, otherwise $\mathbf{q}_4(p) := \mathbf{q}_3(p)$. We define $c_4$ depending on different cases of $t$:

(1) Nop: $t = (q_1, \mathsf{nop}, q_2)$. Define $\mathbf{b}_4 := \mathbf{b}_3$ and $\mathbf{mem}_4 := \mathbf{mem}_2 = \mathbf{mem}_3 = \mathbf{mem}_1$. We have $\alpha_3 \xrightarrow{t}_{\mathsf{DTSO}} \alpha_4$.

(2) Write: $t = (q, \mathsf{w}(x,v), q')$. Define $\mathbf{b}_4 := \mathbf{b}_3\left[h(p) \hookleftarrow (x, v, \textsc{own}) \cdot \mathbf{b}_3(h(p))\right]$ and $\mathbf{mem}_4 := \mathbf{mem}_2$. We have $\alpha_3 \xrightarrow{t}_{\mathsf{DTSO}} \alpha_4$.

(3) Propagate: $t = \mathsf{propagate}_p^x$. Define $\mathbf{mem}_4 := \mathbf{mem}_2 = \mathbf{mem}_3 = \mathbf{mem}_1$ and $\mathbf{b}_4 := \mathbf{b}_3\left[h(p) \hookleftarrow (x, v) \cdot \mathbf{b}_3(h(p))\right]$ where $v = \mathbf{mem}_4(x)$. We have $\alpha_3 \xrightarrow{t}_{\mathsf{DTSO}} \alpha_4$.

(4) Delete: $t = \mathsf{delete}_p$. Define $\mathbf{mem}_4 := \mathbf{mem}_2 = \mathbf{mem}_3 = \mathbf{mem}_1$. Define $\mathbf{b}_4$ according to one of the following cases:
   - If $\mathbf{b}_1 = \mathbf{b}_2\left[p \hookleftarrow \mathbf{b}_2(p) \cdot (x, v)\right]$, then define $\mathbf{b}_4 := \mathbf{b}_3$. In other words, we have $\alpha_4 = \alpha_3$.
   - If $\mathbf{b}_1 = \mathbf{b}_2\left[p \hookleftarrow \mathbf{b}_2(p) \cdot (x, v, \textsc{own})\right]$ and $(x, v', \textsc{own}) \in \mathbf{b}_2(p)$ for some $v' \in \mathbb{V}$, then define $\mathbf{b}_4 := \mathbf{b}_3$. In other words, we have $\alpha_4 = \alpha_3$.
   - If $\mathbf{b}_1 = \mathbf{b}_2\left[p \hookleftarrow \mathbf{b}_2(p) \cdot (x, v, \textsc{own})\right]$ and there is no $v' \in \mathbb{V}$ with $(x, v', \textsc{own}) \in \mathbf{b}_2(p)$, then since $\mathbf{b}_1(p) \sqsubseteq \mathbf{b}_3(h(p))$ we know that there is an $i$ and therefore a smallest $i$ such that $\mathbf{b}_3(h(p))(i) = (x, v, \textsc{own})$. Define
   $$\mathbf{b}_4 := \mathbf{b}_3\left[h(p) \hookleftarrow \mathbf{b}_3(h(p))(1) \cdot \mathbf{b}_3(h(p))(2) \cdots \mathbf{b}_3(h(p))(i-1)\right]$$
   We can perform the following sequence of transitions $\alpha_3 \xrightarrow{\mathsf{delete}_p}_{\mathsf{DTSO}} \alpha'_1 \xrightarrow{\mathsf{delete}_p}_{\mathsf{DTSO}} \alpha'_2 \cdots \xrightarrow{\mathsf{delete}_p}_{\mathsf{DTSO}} \alpha'_{|\mathbf{b}_3(h(p))|-i} \xrightarrow{\mathsf{delete}_p}_{\mathsf{DTSO}} \alpha_4$. In other words, we reach the configuration $\alpha_4$ from $\alpha_3$ by first deleting $|\mathbf{b}_3(h(p))| - i$ messages from the head of $\mathbf{b}_3(h(p))$.

(5) Read: $t = (q, \mathsf{r}(x,v), q')$. Define $\mathbf{mem}_4 := \mathbf{mem}_2$. We define $\mathbf{b}_4$ according to one of the following cases:
   - Read-own-write: If there is an $i : 1 \leq i \leq |\mathbf{b}_1(p)|$ such that $\mathbf{b}_1(p)(i) = (x, v, \textsc{own})$, and there are no $1 \leq j < i$ and $v' \in \mathbb{V}$ such that $\mathbf{b}_1(p)(j) = (x, v', \textsc{own})$. Since $\mathbf{b}_1(p) \sqsubseteq \mathbf{b}_3(h(p))$, there is an $i' : 1 \leq i' \leq |\mathbf{b}_1(p)|$ such that $\mathbf{b}_1(p)(i') = (x, v, \textsc{own})$, and there are no $1 \leq j < i'$ and $v' \in \mathbb{V}$ such that $\mathbf{b}_1(p)(j) = (x, v', \textsc{own})$. Define $\mathbf{b}_4 := \mathbf{b}_3$. In other words, we have that $\alpha_4 = \alpha_3$.
   - Read from buffer: If $(x, v', \textsc{own}) \notin \mathbf{b}_1(p)$ for all $v' \in \mathbb{V}$ and $\mathbf{b}_1 = \mathbf{b}_2\left[p \hookleftarrow \mathbf{b}_2(p) \cdot (x, v)\right]$, then let $i$ be the largest $i : 1 \leq i \leq |\mathbf{b}_3(h(p))|$ such that $\mathbf{b}_3(h(p))(i) = (x, v)$. Since $\mathbf{b}_1(p) \sqsubseteq \mathbf{b}_3(h(p))$, we know that such an $i$ exists. Define
   $$\mathbf{b}_4 := \mathbf{b}_3\left[h(p) \hookleftarrow \mathbf{b}_3(h(p))(1) \cdot \mathbf{b}_3(h(p))(2) \cdots \mathbf{b}_3(h(p))(i-1)\right]$$
   We can reach the configuration $\alpha_4$ from $\alpha_3$ in a similar manner to the last case of the delete transition.

(6) Fence: $t = (q, \mathsf{fence}, q')$. Define $\mathbf{b}_4 := \epsilon$ and $\mathbf{mem}_4 := \mathbf{mem}_2$. We can perform the following sequence of transitions $\alpha_3 \xrightarrow{\mathsf{delete}_p}_{\mathsf{DTSO}} \alpha'_1 \xrightarrow{\mathsf{delete}_p}_{\mathsf{DTSO}} \alpha'_2 \cdots \xrightarrow{\mathsf{delete}_p}_{\mathsf{DTSO}}$

$\alpha'_{|\mathbf{b}_3(h(p))|} \xrightarrow{t}_{\mathsf{DTSO}} \alpha_4$. In other words, we can reach the configuration $\alpha_4$ from $\alpha_3$ by first emptying the contents of $\mathbf{b}_3(h(p))$ and then performing $t$.

(7) ARW: $t = (q, \mathsf{arw}(x, v, v'), q')$. Define $\mathbf{b}_4 := \epsilon$ and $\mathbf{mem}_4 := \mathbf{mem}_2$. We can reach the configuration $\alpha_4$ from $\alpha_3$ in a similar manner to the case of the fence transition.

This concludes the proof of Lemma 5.2. □

## Appendix F. Proof of Lemma 5.4

Consider a parameterized configuration $\alpha = (\mathbb{P}, c)$ with $c = (\mathbf{q}, \mathbf{b}, \mathbf{mem})$. We recall the definition of $\mathsf{minpre}(\{\alpha\})$: $\mathsf{minpre}(\{\alpha\}) {:=} \mathtt{min}\,(\mathtt{Pre}_{\mathcal{T}}\,(\{\alpha\}\uparrow) \cup \{\alpha\}\uparrow)$. We observe that

$$\mathsf{minpre}(\{\alpha\}) = \mathtt{min}\left(\cup_{t\in\Delta\cup\Delta''}\mathtt{min}\left\{\alpha' |\ \alpha' \xrightarrow{t} \alpha\right\} \cup \{\alpha\}\right).$$

For $t \in \Delta \cup \Delta''$, we select $\mathtt{min}\left\{\alpha' |\ \alpha' \xrightarrow{t} \alpha\right\}$ to be the minimal set of all *finite* parameterized configurations of the form $\alpha' = (\mathbb{P}', c')$ with $c' = (\mathbf{q}', \mathbf{b}', \mathbf{mem}')$ such that one of the following properties is satisfied:

(1) Nop: $t = (q_1, \mathsf{nop}, q_2)$, $\mathbf{q}(p) = q_2$ for some $p \in \mathbb{P}$, $\mathbb{P}' = \mathbb{P}$, $\mathbf{q}' = \mathbf{q}\,[p \hookleftarrow q_1]$, $\mathbf{b}' = \mathbf{b}$, and $\mathbf{mem}' = \mathbf{mem}$.

(2) Write: $t = (q_1, \mathsf{w}(x, v), q_2)$, $\mathbf{mem}(x) = v$ for some $v \in \mathbb{V}$, $\mathbf{mem}'(y) = \mathbf{mem}(y)$ if $y \neq x$, and one of the following conditions is satisfied:
- $\mathbf{q}(p) = q_2$ for some $p \in \mathbb{P}$, $\mathbb{P}' = \mathbb{P}$, $\mathbf{q}' = \mathbf{q}\,[p \hookleftarrow q_1]$, $\mathbf{b}' = \mathbf{b}\,[p \hookleftarrow w]$, $\mathbf{b}(p) = (x, v, \text{OWN}) \cdot w$ for some $w \in ((\mathbb{X} \times \mathbb{V}) \cup (\mathbb{X} \times \mathbb{V} \times \{\text{OWN}\}))^*$.
- $\mathbf{q}(p) = q_2$ for some $p \in \mathbb{P}$, $\mathbb{P}' = \mathbb{P}$, $\mathbf{q}' = \mathbf{q}\,[p \hookleftarrow q_1]$, $\mathbf{b}(p) = (x, v, \text{OWN}) \cdot w$ for some $w \in ((\mathbb{X} \times \mathbb{V}) \cup (\mathbb{X} \times \mathbb{V} \times \{\text{OWN}\}))^*$, $\mathbf{b}' = \mathbf{b}\,[p \hookleftarrow w_1 \cdot (x, v', own) \cdot w_2]$ for some $v' \in \mathbb{V}$ where $w_1, w_2 \in ((\mathbb{X} \times \mathbb{V}) \cup (\mathbb{X} \times \mathbb{V} \times \{\text{OWN}\}))^*$, $w_1 \cdot w_2 = w$ and $(x, v'', own) \notin w_1$ for all $v'' \in \mathbb{V}$. In other words, $(x, v', own)$ is the most recent message to variable $x$ belonging to $p$ in the buffer $\mathbf{b}'(p)$. This condition corresponds to the case when we have some messages $(x, v', own)$ that are hidden by the message $(x, v, own)$ in the buffer $\mathbf{b}(p)$.
- $\mathbf{q}(p) \neq q_2$ or $\mathbf{b}(p) \neq (x, v, \text{OWN}) \cdot w$ for any $p \in \mathbb{P}$, $w \in ((\mathbb{X} \times \mathbb{V}) \cup (\mathbb{X} \times \mathbb{V} \times \{\text{OWN}\}))^*$, $\mathbb{P}' = \mathbb{P} \cup \{p'\}$ for some $p' \notin \mathbb{P}$, $\mathbf{q}'(p') = q_1$, $\mathbf{q}'(p'') = \mathbf{q}(p'')$ if $p'' \neq p'$, $\mathbf{b}'(p') = \langle (x_1, v_1, \text{OWN}) | \epsilon \rangle \langle (x_2, v_2, \text{OWN}) | \epsilon \rangle \cdots \langle (x_m, v_m, \text{OWN}) | \epsilon \rangle$ where $x_i \neq x_j$, $v_i \in \mathbb{V}$, $1 \leq i, j \leq |X|$ and $\mathbf{b}'(p'') = \mathbf{b}(p'')$ if $p'' \neq p'$. In other words, we add one more process $p'$ to the configuration $\alpha'$.

(3) Propagate: $t = \mathsf{propagate}_p^x$ for some $p \in \mathbb{P}$, $\mathbf{mem}(x) = v$, $\mathbb{P}' = \mathbb{P}$, $\mathbf{q}' = \mathbf{q}$, $\mathbf{mem}' = \mathbf{mem}$, $\mathbf{b}(p) = (x, v) \cdot w$ for some $w \in ((\mathbb{X} \times \mathbb{V}) \cup (\mathbb{X} \times \mathbb{V} \times \{\text{OWN}\}))^*$, and $\mathbf{b}' = \mathbf{b}\,[p \hookleftarrow w]$.

(4) Read: $t = (q_1, \mathsf{r}(x, v), q_2)$, $\mathbf{q}(p) = q_2$ for some $p \in \mathbb{P}$, $\mathbb{P}' = \mathbb{P}$, $\mathbf{q}' = \mathbf{q}\,[p \hookleftarrow q_1]$, and $\mathbf{mem}' = \mathbf{mem}$, and one of the following conditions is satisfied:
- Read-own-write: there is an $i : 1 \leq i \leq |\mathbf{b}(p)|$ such that $\mathbf{b}(p)(i) = (x, v, own)$, and there are no $j : 1 \leq j < i$ and $v' \in \mathbb{V}$ such that $\mathbf{b}(p)(j) = (x, v', own)$, and $\mathbf{b}' = \mathbf{b}$.
- Read from buffer: $(x, v', \text{OWN}) \notin \mathbf{b}(p)$ for all $v' \in \mathbb{V}$, $\mathbf{b}(p) = w \cdot (x, v)$ for some $w \in ((\mathbb{X} \times \mathbb{V}) \cup (\mathbb{X} \times \mathbb{V} \times \{\text{OWN}\}))^*$, and $\mathbf{b}' = \mathbf{b}$.
- Read from buffer: $(x, v', own) \notin \mathbf{b}(p)$ for all $v' \in \mathbb{V}$, $\mathbf{b}(p) \neq w \cdot (x, v)$ for any $w \in ((\mathbb{X} \times \mathbb{V}) \cup (\mathbb{X} \times \mathbb{V} \times \{\text{OWN}\}))^*$, and $\mathbf{b}' = \mathbf{b}\,[p \hookleftarrow \mathbf{b}(p) \cdot (x, v)]$. This condition corresponds to the case when we have some messages $(x, v)$ that are not explicitly presented at the head of the buffer $\mathbf{b}(p)$.

(5) Fence: $t = (q_1, \mathsf{fence}, q_2)$, $\mathbf{q}(p) = q_2$ for some $p \in \mathbb{P}$, $\mathbf{b}(p) = \epsilon$, $\mathbb{P}' = \mathbb{P}$, $\mathbf{q}' = \mathbf{q}\,[p \hookleftarrow q_1]$, $\mathbf{b}' = \mathbf{b}$, and $\mathbf{mem}' = \mathbf{mem}$.

(6) ARW: $t = (q_1, \mathsf{arw}(x, v, v'), q_2)$, $\mathbf{mem}(x) = v'$, $\mathbf{mem}' = \mathbf{mem}\,[x \hookleftarrow v]$, and one of the following conditions is satisfied:
- $\mathbf{q}(p) = q_2$ for some $p \in \mathbb{P}$, $\mathbf{b}(p) = \epsilon$, $\mathbb{P}' = \mathbb{P}$, $\mathbf{q}' = \mathbf{q}\,[p \hookleftarrow q_1]$, $\mathbf{b}' = \mathbf{b}$.
- $\mathbf{q}(p) \neq q_2$ or $\mathbf{b}(p) \neq \epsilon$ for any $p \in \mathbb{P}$, $\mathbb{P}' = \mathbb{P} \cup \{p'\}$ for some $p' \notin \mathbb{P}$, $\mathbf{q}'(p') = q_1$, $\mathbf{q}'(p'') = \mathbf{q}(p'')$ if $p'' \neq p'$, $\mathbf{b}'(p') = \epsilon$, and $\mathbf{b}'(p'') = \mathbf{b}(p'')$ if $p'' \neq p'$. In other words, we add one more process $p'$ to the configuration $\alpha'$.

(7) Delete: $t = \mathsf{delete}_p$ for some $p \in \mathbb{P}$, $\mathbb{P}' = \mathbb{P}$, $\mathbf{q}' = \mathbf{q}$, $\mathbf{mem}' = \mathbf{mem}$, $(x, v, own) \notin \mathbf{b}(p)$ for some $x \in \mathbb{X}$ and all $v \in \mathbb{V}$, $\mathbf{b}' = \mathbf{b}\,[p \hookleftarrow \mathbf{b}(p) \cdot (x, v', own)]$ for some $v' \in \mathbb{V}$.

This concludes the proof of Lemma 5.4. $\square$