TSO Games
On the decidability of safety games under the total store order semantics

Stephan Spengler
Uppsala University
Uppsala, Sweden
stephan.spengler@it.uu.se

Sanchari Sil
Chennai Mathematical Institute
Chennai, India
sanchari@cmi.ac.in

We consider an extension of the classical Total Store Order (TSO) semantics by expanding it to turn-based 2-player safety games. During her turn, a player can select any of the communicating processes and perform its next transition. We consider different formulations of the safety game problem depending on whether one player or both of them transfer messages from the process buffers to the shared memory. We give the complete decidability picture for all the possible alternatives.

1 Introduction

Most modern architectures, such as Intel x86 [21], SPARC [27], IBM's POWER [20], and ARM [11], implement several relaxations and optimisations that reduce the latency of memory accesses. This has the effect of breaking the Sequential Consistency (SC) assumption [22]. SC is the classical strong semantics for concurrent programs that interleaves the parallel executions of processes while maintaining the order in which instructions were issued. Programmers usually assume that the execution of programs follows the SC model. However, this is not true when we consider concurrent programs running on modern architectures. In fact, even simple programs such as mutual exclusion and producer-consumer protocols, that are correct under SC, may exhibit erroneous behaviors. This is mainly due to the relaxation of the execution order of the instructions. For instance, a standard relaxation is to allow the reordering of reads and writes of the same process if the reads have been issued after the writes and they concern different memory locations. This relaxation can be implemented using an unbounded perfect FIFO queue/buffer between each process and the memory. These buffers are used to store delayed writes. The corresponding model is called Total Store Ordering (TSO) and corresponds to the formalisation of SPARC and Intel x86 [24, 26].

In TSO, an unbounded buffer is associated with each process. When a process executes a write operation, this write is appended to the end of the buffer of that process. A pending write operation on the variable x at the head of a buffer can be deleted in a non-deterministic manner. This updates the value of the shared variable x in the memory. To perform a read operation on a variable x, the process first checks its buffer for a pending write operation on the variable x. If such a write exists, then the process reads the value written by the newest pending write operation on x. Otherwise, the process fetches the value of the variable x from the memory. The verification of programs running under TSO is challenging due to the unboundedness of the buffers. In fact, the induced state space of a program under TSO maybe infinite even if the program itself is a finite-state system.

The reachability problem for programs under TSO checks whether a given program state is reachable during program execution. It is also called safety problem, in case the target state is considered to be a bad state. It has been shown decidable using different alternative semantics for TSO (e.g., [13, 4, 3]). Furthermore, it has been shown in [13] that lossy channel systems (see e.g., [10, 18, 9, 25]) can

A. Achilleos and D. Della Monica (Eds.): Fourteenth International Symposium on Games, Automata, Logics, and Formal Verification (GandALF 2023).
EPTCS 390, 2023, pp. 82–98, doi:10.4204/EPTCS.390.6

be simulated by programs running under TSO. This entails that the reachability problem for programs under TSO is non-primitive recursive and that the repeated reachability problem is undecidable. This is an immediate consequence of the fact that the reachability problem for lossy channel is non-primitive recursive [25] and that the repeated reachability problem is undecidable [9]. The termination problem for programs running under TSO has been shown to be decidable in [12] using the framework of well-structured transition systems [18, 10].

The authors of [14, 15] consider the robustness problem for programs running under TSO. This problem consists in checking whether, for any given TSO execution, there is an equivalent SC execution of the same program. Two executions are declared equivalent by the robustness criterion if they agree on (1) the order in which instructions are executed within the same process (i.e., program order), (2) the write instruction from which each read instruction fetches its value (i.e., read-from relation), and (3) the order in which write instruction on the same variable are committed to memory (i.e., store ordering). The problem of checking whether a program is robust has been shown to be PSPACE-complete in [14]. A variant of the robustness problem which is called persistence, declares that two runs are equivalent if (1) they have the same program order and (2) all write instructions reach the memory in the same order. Checking the persistency of a program under TSO has been shown to be PSPACE-complete in [6]. Observe that the persistency and robustness problems are stronger than the safety problem (i.e., if a program is safe under SC and robust/persistent, then it is also safe under TSO).

Due to the non-determinism of the buffer updates, the buffers associated with each process under TSO appear to exhibit a lossy behaviour. Previously, games on lossy channel systems (and more general on monotonic systems) were studied in [8]. Unfortunately these results are not applicable / transferable to programs under TSO whose induced transition systems are not monotone [13].

In this paper, we consider a natural continuation of the works on both the study of the decidability/complexity of the formal verification of programs under TSO and the study of games on concurrent systems. This is further motivated by the fact that formal games provide a framework to reason about a system's behaviour, which can be leveraged in control model checking, for example in controller synthesis problems.

In more detail, we consider (safety) games played on the transition systems induced by programs running under TSO. Given a program under TSO, we construct a game in which two players A and B take turns in executing instructions of the program. The goal of player B is to reach a given set of final configurations, while player A tries to avoid this. Thus, it can also be seen as a reachability game with respect to player B. In this game, the turn determines which player will execute the next program instruction. However, this definition leaves the control of updates undefined. To address this, we give the player the possibility to update memory by removing the pending writes from the buffer between the execution of two instructions.

The control over the buffer updates is shared between the two players in varying ways. We differentiate between multiple scenarios based on when exactly each player is allowed to update. In particular, for each player A or B we have the following cases: (1) she can never update, (2) she can update after her own turn, (3) she can update before her own turn, and (4) she can always update, i.e. before and after her own turn. In total, we obtain an exhaustive collection of 16 different TSO games. We divide these 16 games into four different groups, depending on their decidability results.

- Group I (7 games) can be reduced to TSO games with 2-bounded buffers.
- Group II (1 game) can be reduced to TSO games with bounded buffers.
- Group III (7 games) can simulate perfect channel systems.
- Group IV (1 game) can be reduced to a finite game without buffers.

		Player A: always	before	after	never
Player B:	always	I (d)			
	before		II (d)		
	after			III (u)	
	never				IV (d)

Figure 1: Groups of TSO games, where players A and B are allowed to update the buffer: always, before their own move, after their own move, or never. The games in group I, II and IV are decidable (d), the games in group III are undecidable (u).

This classification is shown in Figure 1. Of these four groups, only Group III is undecidable, the others each reduce to a finite game and are thus decidable.

Finally, we establish the exact computational complexity for the decidable games. In fact, we show that the problem is EXPTIME-complete. We prove EXPTIME-hardness by a reduction from the problem of acceptance of a word by a linearly bounded alternating Turing machine [17]. To prove EXPTIME-membership, we show that it is possible to compute the set of winning region for player B in exponential time. These results are surprising given the non-primitive recursive complexity of the reachability problem for programs under TSO and the undecidability of the repeated reachability problem.

Related Works. In addition to the related work mentioned in the introduction on the decidability / complexity of the verification problems of programs running under TSO, there have been some works on parameterized verification of programs running under TSO. The problem consists in verifying a concurrent program regardless of the number of involved processes (which are identical finite-state systems). The parameterised reachability problem of programs running under TSO has been shown to be decidable in [2, 3]. While this problem for concurrent programs performing only read and writing operations (no atomic read-write instructions) is PSPACE-complete [7]. This result has been recently extended to processes manipulating abstract data types over infinite domains [5]. Checking the robustness of a parameterised concurrent system is decidable and EXPSPACE-hard [14].

As far as we know this is the first work that considers the game problem for programs running under TSO. The proofs and techniques used in this paper are different from the ones used to prove decidability / complexity results for the verification of programs under TSO except the undecidability result which uses some ideas from the reduction from the reachability problem for lossy channel systems to its corresponding problem for programs under TSO [13]. However, our undecidability proof requires us to implement a protocol that detects lossiness of messages in order to turn the lossy channel system into a perfect one (which is the most intricate part of the proof).

2 Preliminaries

2.1 Transition Systems

A *(labeled) transition system* is a triple $\langle \mathsf{C}, \mathsf{L}, \rightarrow \rangle$, where C is a set of *configurations*, L is a set of *labels*, and $\rightarrow \subseteq \mathsf{C} \times \mathsf{L} \times \mathsf{C}$ is a *transition relation*. We usually write $\mathsf{c}_1 \xrightarrow{\mathsf{label}} \mathsf{c}_2$ if $\langle \mathsf{c}_1, \mathsf{label}, \mathsf{c}_2 \rangle \in \rightarrow$. Furthermore, we write $\mathsf{c}_1 \rightarrow \mathsf{c}_2$ if there exists some label such that $\mathsf{c}_1 \xrightarrow{\mathsf{label}} \mathsf{c}_2$. A *run* π of $\mathcal{T}$ is a sequence of transitions $\mathsf{c}_0 \xrightarrow{\mathsf{label}_1} \mathsf{c}_1 \xrightarrow{\mathsf{label}_2} \mathsf{c}_2 \ldots \xrightarrow{\mathsf{label}_n} \mathsf{c}_n$. It is also written as $\mathsf{c}_0 \xrightarrow{\pi} \mathsf{c}_n$. A configuration c' is

reachable from a configuration c, if there exists a run from c to c'.

For a configuration c, we defined $\mathrm{Pre}(c) = \{c' \mid c' \to c\}$ and $\mathrm{Post}(c) = \{c' \mid c \to c'\}$. We extend these notions to sets of configurations C' with $\mathrm{Pre}(C') = \bigcup_{c \in C'} \mathrm{Pre}(c)$ and $\mathrm{Post}(C') = \bigcup_{c \in C'} \mathrm{Post}(c)$.

An *unlabeled transition system* is a transition system without labels. Formally, it is defined as a TS with a singleton label set. In this case, we omit the labels.

2.2 Perfect Channel Systems

Given a set of messages M, define the set of channel operations $\mathsf{Op} = \{!m, ?m \mid m \in M\} \cup \{\mathtt{skip}\}$. A *perfect channel system* (PCS) is a triple $\mathcal{L} = \langle S, M, \delta \rangle$, where S is a set of states, M is a set of messages, and $\delta \subseteq S \times \mathsf{Op} \times S$ is a transition relation. We write $s_1 \xrightarrow{\mathsf{op}} s_2$ if $\langle s_1, \mathsf{op}, s_2 \rangle \in \delta$.

Intuitively, a PCS models a finite state automaton that is augmented by a *perfect* (i.e. non-lossy) FIFO buffer, called *channel*. During a *send operation* $!m$, the channel system appends m to the tail of the channel. A transition $?m$ is called *receive operation*. It is only enabled if the channel is not empty and m is its oldest message. When the channel system performs this operation, it removes m from the head of the channel. Lastly, a `skip` operation just changes the state, but does not modify the buffer.

The formal semantics of $\mathcal{L}$ are defined by a transition system $\mathcal{T}_\mathcal{L} = \langle C_\mathcal{L}, L_\mathcal{L}, \to_\mathcal{L} \rangle$, where $C_\mathcal{L} = S \times M^*$, $L_\mathcal{L} = \mathsf{Op}$ and the transition relation $\to_\mathcal{L}$ is the smallest relation given by:

- If $s_1 \xrightarrow{!m} s_2$ and $w \in M^*$, then $\langle s_1, w \rangle \xrightarrow{!m}_\mathcal{L} \langle s_2, m \cdot w \rangle$.
- If $s_1 \xrightarrow{?m} s_2$ and $w \in M^*$, then $\langle s_1, w \cdot m \rangle \xrightarrow{?m}_\mathcal{L} \langle s_2, w \rangle$.
- If $s_1 \xrightarrow{\mathtt{skip}} s_2$ and $w \in M^*$, then $\langle s_1, w \rangle \xrightarrow{\mathtt{skip}}_\mathcal{L} \langle s_2, w \rangle$.

A state $s_F \in S$ is *reachable* from a configuration $c_0 \in C_\mathcal{L}$, if there exists a configuration $c_F = \langle s_F, w_F \rangle$ such that c_F is reachable from c_0 in $\mathcal{T}_\mathcal{L}$. The **state reachability problem** of PCS is, given a perfect channel system $\mathcal{L}$, an initial configuration $c_0 \in C_\mathcal{L}$ and a final state $s_F \in S$, to decide whether s_F is reachable from c_0 in $\mathcal{T}_\mathcal{L}$. It is undecidable [16].

3 Concurrent Programs

3.1 Syntax

Let Dom be a finite data domain and Vars be a finite set of shared variables over Dom. We define the *instruction set* $\mathsf{Instrs} = \{\mathtt{rd}(x,d), \mathtt{wr}(x,d) \mid x \in \mathsf{Vars}, d \in \mathsf{Dom}\} \cup \{\mathtt{skip}, \mathtt{mf}\}$, which are called *read*, *write*, *skip* and *memory fence*, respectively. A process is represented by a finite state labeled transition system. It is given as the triple $\mathsf{Proc} = \langle Q, \mathsf{Instrs}, \delta \rangle$, where Q is a finite set of *local states* and $\delta \subseteq Q \times \mathsf{Instrs} \times Q$ is the transition relation. As with transition systems, we write $q_1 \xrightarrow{\mathsf{instr}} q_2$ if $\langle q_1, \mathsf{instr}, q_2 \rangle \in \delta$ and $q_1 \to q_2$ if there exists some instr such that $q_1 \xrightarrow{\mathsf{instr}} q_2$.

A *concurrent program* is a tuple of processes $\mathcal{P} = \langle \mathsf{Proc}^\iota \rangle_{\iota \in \mathcal{I}}$, where $\mathcal{I}$ is a finite set of process identifiers. For each $\iota \in \mathcal{I}$ we have $\mathsf{Proc}^\iota = \langle Q^\iota, \mathsf{Instrs}, \delta^\iota \rangle$. A *global* state of $\mathcal{P}$ is a function $\mathcal{S} : \mathcal{I} \to \bigcup_{\iota \in \mathcal{I}} Q^\iota$ that maps each process to its local state, i.e $\mathcal{S}(\iota) \in Q^\iota$.

3.2 TSO Semantics

Under TSO semantics, the processes of a concurrent program do not interact with the shared memory directly, but indirectly through a FIFO *store buffer* instead. When performing a *write* instruction $\mathtt{wr}(x,d)$,

read-own-write
$$\frac{\mathsf{q} \xrightarrow{\mathtt{rd(x,d)}} \mathsf{q}' \qquad \mathcal{S}(\iota)=\mathsf{q} \qquad \mathcal{B}(\iota)|_{\{\mathsf{x}\}\times\mathsf{Dom}}=\langle \mathsf{x},\mathsf{d}\rangle\cdot\mathsf{w}}{\langle \mathcal{S},\mathcal{B},\mathcal{M}\rangle \xrightarrow{\mathtt{rd(x,d)}_\iota}_{\mathcal{P}} \langle \mathcal{S}[\iota\leftarrow\mathsf{q}'],\mathcal{B},\mathcal{M}\rangle}$$

read-from-memory
$$\frac{\mathsf{q} \xrightarrow{\mathtt{rd(x,d)}} \mathsf{q}' \qquad \mathcal{S}(\iota)=\mathsf{q} \qquad \mathcal{B}(\iota)|_{\{\mathsf{x}\}\times\mathsf{Dom}}=\varepsilon \qquad \mathcal{M}(\mathsf{x})=\mathsf{d}}{\langle \mathcal{S},\mathcal{B},\mathcal{M}\rangle \xrightarrow{\mathtt{rd(x,d)}_\iota}_{\mathcal{P}} \langle \mathcal{S}[\iota\leftarrow\mathsf{q}'],\mathcal{B},\mathcal{M}\rangle}$$

write
$$\frac{\mathsf{q} \xrightarrow{\mathtt{wr(x,d)}} \mathsf{q}' \qquad \mathcal{S}(\iota)=\mathsf{q}}{\langle \mathcal{S},\mathcal{B},\mathcal{M}\rangle \xrightarrow{\mathtt{wr(x,d)}_\iota}_{\mathcal{P}} \langle \mathcal{S}[\iota\leftarrow\mathsf{q}'],\mathcal{B}[\iota\leftarrow\langle \mathsf{x},\mathsf{d}\rangle\cdot\mathcal{B}(\iota)],\mathcal{M}\rangle}$$

skip
$$\frac{\mathsf{q} \xrightarrow{\mathtt{skip}} \mathsf{q}' \qquad \mathcal{S}(\iota)=\mathsf{q}}{\langle \mathcal{S},\mathcal{B},\mathcal{M}\rangle \xrightarrow{\mathtt{skip}_\iota}_{\mathcal{P}} \langle \mathcal{S}[\iota\leftarrow\mathsf{q}'],\mathcal{B},\mathcal{M}\rangle}$$

memory-fence
$$\frac{\mathsf{q} \xrightarrow{\mathtt{mf}} \mathsf{q}' \qquad \mathcal{S}(\iota)=\mathsf{q} \qquad \mathcal{B}(\iota)=\varepsilon}{\langle \mathcal{S},\mathcal{B},\mathcal{M}\rangle \xrightarrow{\mathtt{mf}_\iota}_{\mathcal{P}} \langle \mathcal{S}[\iota\leftarrow\mathsf{q}'],\mathcal{B},\mathcal{M}\rangle}$$

update
$$\frac{\mathcal{B}(\iota)=\mathsf{w}\cdot\langle \mathsf{x},\mathsf{d}\rangle}{\langle \mathcal{S},\mathcal{B},\mathcal{M}\rangle \xrightarrow{\mathtt{up}_\iota}_{\mathcal{P}} \langle \mathcal{S},\mathcal{B}[\iota\leftarrow\mathsf{w}],\mathcal{M}[\mathsf{x}\leftarrow\mathsf{d}]\rangle}$$

Figure 2: TSO semantics

the process adds a new message $\langle \mathsf{x},\mathsf{d}\rangle$ to the tail of its store buffer. A *read* instruction $\mathtt{rd}(\mathsf{x},\mathsf{d})$ works differently depending on the current buffer content of the process. If the buffer contains a write message on variable x, the value d must correspond to the value of the most recent such message. Otherwise, the value is read directly from memory. A *skip* instruction only changes the local state of the process. The *memory fence* instruction is disabled, i.e. it cannot be executed, unless the buffer of the process is empty. Additionally, at any point during the execution, the process can *update* the write message at the head of its buffer to the memory. For example, if the oldest message in the buffer is $\langle \mathsf{x},\mathsf{d}\rangle$, it will be removed from the buffer and the memory value of variable x will be updated to contain the value d. This happens in a non-deterministic manner.

Formally, we introduce a TSO *configuration* as a tuple $\mathsf{c}=\langle \mathcal{S},\mathcal{B},\mathcal{M}\rangle$, where:

- $\mathcal{S}:\mathcal{I}\to\bigcup_{\iota\in\mathcal{I}}\mathsf{Q}^\iota$ is a global state of $\mathcal{P}$.
- $\mathcal{B}:\mathcal{I}\to(\mathsf{Vars}\times\mathsf{Dom})^*$ represents the buffer state of each process.
- $\mathcal{M}:\mathsf{Vars}\to\mathsf{Dom}$ represents the memory state of each shared variable.

Given a configuration c, we write $\mathcal{S}(\mathsf{c})$, $\mathcal{B}(\mathsf{c})$ and $\mathcal{M}(\mathsf{c})$ for the global program state, buffer state and memory state of c. The semantics of a concurrent program running under TSO is defined by a transition system $\mathcal{T}_\mathcal{P}=\langle \mathsf{C}_\mathcal{P},\mathsf{L}_\mathcal{P},\to_\mathcal{P}\rangle$, where $\mathsf{C}_\mathcal{P}$ is the set of all possible TSO configurations, $\mathsf{L}_\mathcal{P}=\{\mathsf{instr}_\iota \mid \mathsf{instr}\in\mathsf{Instrs},\iota\in\mathcal{I}\}\cup\{\mathtt{up}_\iota \mid \iota\in\mathcal{I}\}$ is the set of labels. The transition relation $\to_\mathcal{P}$ is given by the rules in Figure 2, where we use $\mathcal{B}(\iota)|_{\{\mathsf{x}\}\times\mathsf{Dom}}$ to denote the restriction of $\mathcal{B}(\iota)$ to write messages on the variable x.

A global state $\mathcal{S}_F$ is *reachable* from an initial configuration c_0, if there is a configuration c_F with $\mathcal{S}(\mathsf{c}_F)=\mathcal{S}_F$ such that c_F is reachable from c_0 in $\mathcal{T}_\mathcal{P}$. The **state reachability problem** of TSO is, given a program $\mathcal{P}$, an initial configuration c_0 and a final global state $\mathcal{S}_F$, to decide whether $\mathcal{S}_F$ is reachable from c_0 in $\mathcal{T}_\mathcal{P}$.

We define $\mathtt{up}^*$ to be the transitive closure of $\{\mathtt{up}_\iota \mid \iota \in \mathcal{I}\}$, i.e. $\mathsf{c}_1 \xrightarrow{\mathtt{up}^*}_{\mathcal{P}} \mathsf{c}_2$ if and only if c_2 can be obtained from c_1 by some amount of buffer updates.

4 Games

4.1 Definitions

A *(safety) game* is an unlabeled transition sytem, in which two players A and B take turns making a *move* in the transition system, i.e. changing the state of the game from one configuration to an adjacent one. The goal of player B is to reach a given set of final configurations, while player A tries to avoid this. Thus, it can also be seen as a *reachability* game with respect to player B.

Formally, a game is defined as a tuple $\mathcal{G} = \langle \mathsf{C}, \mathsf{C}_A, \mathsf{C}_B, \to, \mathsf{C}_F \rangle$, where C is the set of configurations, C_A and C_B form a partition of C, the transition relation is restricted to $\to \subseteq (\mathsf{C}_A \times \mathsf{C}_B) \cup (\mathsf{C}_B \times \mathsf{C}_A)$, and $\mathsf{C}_F \subseteq \mathsf{C}_A$ is a set of *final states*. Furthermore, we assume without loss of generality that $\mathcal{G}$ is deadlock-free, i.e. $\mathrm{Post}(\mathsf{c}) \neq \emptyset$ for all $\mathsf{c} \in \mathsf{C}$.

A *play* P of $\mathcal{G}$ is an infinite sequence $\mathsf{c}_0, \mathsf{c}_1, \ldots$ such that $\mathsf{c}_i \to \mathsf{c}_{i+1}$ for all $i \in \mathbb{N}$. In the context of safety games, P is *winning* for player B if there is $i \in \mathbb{N}$ such that $\mathsf{c}_i \in \mathsf{C}_F$. Otherwise, it is *winning* for player A. This means that player B tries to force the play into C_F, while player A tries to avoid this.

A *strategy* of player A is a partial function $\sigma_A : \mathsf{C}^* \rightharpoonup \mathsf{C}_B$, such that $\sigma_A(\mathsf{c}_0, \ldots, \mathsf{c}_n)$ is defined if and only if $\mathsf{c}_0, \ldots, \mathsf{c}_n$ is a prefix of a play, $\mathsf{c}_n \in \mathsf{C}_A$ and $\sigma_A(\mathsf{c}_0, \ldots, \mathsf{c}_n) \in \mathrm{Post}(\mathsf{c}_n)$. A strategy σ_A is called *positional*, if it only depends on c_n, i.e. if $\sigma_A(\mathsf{c}_0, \ldots, \mathsf{c}_n) = \sigma_A(\mathsf{c}_n)$ for all $(\mathsf{c}_0, \ldots, \mathsf{c}_n)$ on which σ_A is defined. Thus, a positional strategy is usually given as a total function $\sigma_A : \mathsf{C}_A \to \mathsf{C}_B$. Given two games $\mathcal{G}$ and $\mathcal{G}'$ and a strategy σ_A for $\mathcal{G}$, an *extension* of σ_A to $\mathcal{G}'$ is a strategy σ'_A of $\mathcal{G}'$ that is also an extension of σ_A to the configuration set of $\mathcal{G}'$ in the mathematical sense, i.e. $\sigma_A(\mathsf{c}_0, \ldots, \mathsf{c}_n) = \sigma'_A(\mathsf{c}_0, \ldots, \mathsf{c}_n)$ for all $(\mathsf{c}_0, \ldots, \mathsf{c}_n)$ on which σ_A is defined. Conversely, σ_A is called the *restriction* of σ'_A to $\mathcal{G}$. For player B, strategies are defined accordingly.

Two strategies σ_A and σ_B together with an initial configuration c_0 induce a play $\mathsf{P}(\mathsf{c}_0, \sigma_A, \sigma_B) = \mathsf{c}_0, \mathsf{c}_1, \ldots$ such that $\mathsf{c}_{i+1} = \sigma_A(\mathsf{c}_0, \ldots, \mathsf{c}_i)$ for all $\mathsf{c}_i \in \mathsf{C}_A$ and $\mathsf{c}_{i+1} = \sigma_B(\mathsf{c}_0, \ldots, \mathsf{c}_i)$ for all $\mathsf{c}_i \in \mathsf{C}_B$. A strategy σ_A is *winning* from a configuration c, if for *all* strategies σ_B it holds that $\mathsf{P}(\sigma_A, \sigma_B, \mathsf{c})$ is a winning play for player A. A configuration c is *winning* for player A if she has a strategy that is winning from c. Equivalent notions exist for player B. The **safety problem** for a game $\mathcal{G}$ and a configuration c is to decide whether c is winning for player A.

Lemma 1 (Proposition 2.21 in [23])**.** *In safety games, every configuration is winning for exactly one player. A player with a winning strategy also has a positional winning strategy.*

Since we only consider safety games in this paper, strategies will be considered to be positional unless explicitly stated otherwise. Furthermore, Lemma 1 implies the following:

- $\mathsf{c}_A \in \mathsf{C}_A$ is winning for player A $\iff$ there is $\mathsf{c}_B \in \mathrm{Post}(\mathsf{c}_A)$ that is winning for player A.
- $\mathsf{c}_B \in \mathsf{C}_B$ is winning for player A $\iff$ all $\mathsf{c}_A \in \mathrm{Post}(\mathsf{c}_B)$ are winning for player A.

A *finite game* is a game with a finite set of configurations. It is rather intuitive that the safety problem is decidable for finite games, e.g. by applying a backward induction algorithm. In particular, the winning configurations for each player are computable in linear time:

Lemma 2 (Chapter 2 in [19])**.** *Computing the set of winning configurations for a finite game with n configurations and m transitions is in $\mathcal{O}(n+m)$.*

4.2 TSO games

A TSO program $\mathcal{P} = \langle \mathsf{Proc}^\iota \rangle_{\iota \in \mathcal{I}}$ and a set of final local states $\mathsf{Q}_F^{\mathcal{P}} \subseteq \mathsf{Q}^{\mathcal{P}}$ induce a safety game $\mathcal{G}(\mathcal{P}, \mathsf{Q}_F^{\mathcal{P}}) = \langle \mathsf{C}, \mathsf{C}_A, \mathsf{C}_B, \rightarrow, \mathsf{C}_F \rangle$ as follows. The sets C_A and C_B are copies of the set $\mathsf{C}^{\mathcal{P}}$ of TSO configurations, annotated by A and B, respectively: $\mathsf{C}_A := \{\mathsf{c}_A \mid \mathsf{c} \in \mathsf{C}^{\mathcal{P}}\}$ and $\mathsf{C}_B := \{\mathsf{c}_B \mid \mathsf{c} \in \mathsf{C}^{\mathcal{P}}\}$. The set of final configurations is defined as $\mathsf{C}_F := \{\langle \mathcal{S}, \mathcal{B}, \mathcal{M} \rangle_A \in \mathsf{C}_A \mid \exists\, \iota \in \mathcal{I} : \mathcal{S}(\iota) \in \mathsf{Q}_F^{\mathcal{P}}\}$, i.e. the set of all configurations where at least one process is in a final state. The transition relation $\rightarrow$ is defined by the following rules:

- For each transition $\mathsf{c} \xrightarrow{\mathsf{instr}_\iota}_{\mathcal{P}} \mathsf{c}'$ where $\mathsf{c}, \mathsf{c}' \in \mathsf{C}^{\mathcal{P}}$, $\iota \in \mathcal{I}$ and $\mathsf{instr} \in \mathsf{Instrs}$, it holds that $\mathsf{c}_A \rightarrow \mathsf{c}'_B$ and $\mathsf{c}_B \rightarrow \mathsf{c}'_A$. This means that each player can execute any TSO instruction, but they take turns alternatingly.
- *If player A can update before her own turn:* For each transition $\mathsf{c}_A \rightarrow \mathsf{c}'_B$ introduced by any of the previous rules, it holds that $\tilde{\mathsf{c}}_A \rightarrow \mathsf{c}'_B$ for all $\tilde{\mathsf{c}}$ with $\tilde{\mathsf{c}} \xrightarrow{\mathtt{up}^*}_{\mathcal{P}} \mathsf{c}$.
- *If player A can update after her own turn:* For each transition $\mathsf{c}_A \rightarrow \mathsf{c}'_B$ introduced by any of the previous rules, it holds that $\mathsf{c}_A \rightarrow \tilde{\mathsf{c}}'_B$ for all $\tilde{\mathsf{c}}'$ with $\mathsf{c}' \xrightarrow{\mathtt{up}^*}_{\mathcal{P}} \tilde{\mathsf{c}}'$.
- The update rules for player B are defined in a similar manner.

From this definition, we obtain 16 different variants of TSO games, which differ in whether each of the players can update *never*, *before* her turn, *after* her turn, or *always* (before and after her turn). We group games with similar decidability and complexity results together. An overview of these four groups is presented in Figure 1. Each group is described in detail in the following sections.

But first, we present a general result that gives a lower complexity bound for all groups of TSO games. Unexpectedly, even a single process is enough to show EXPTIME-hardness. We prove this by reducing the *word acceptance problem* of *linearly bounded alternating Turing machines* (ATM) to the safety problem of a single-process TSO game. The idea is to store the state and head position of the ATM in the local state of the process, and use a set of variables to save the word on the working tape. Based on the alternations of the Turing machine, either player A or player B decides which transition the program will simulate. Interestingly, we can argue that the exact type of TSO game is irrelevant. Moreover, the construction does not make use of the memory buffers, which implies that the result would even hold if the program followed SC semantics. The formal proof can be found in Appendix A of the extended version of this paper [28].

Theorem 3. *The safety problem for TSO games is* EXPTIME*-hard.*

5 Group I

All TSO games in this group have the following in common: There is one player that can update messages *after* her turn, and the other player can update messages *before* her turn. Both players might be allowed to do more than that, but fortunately we do not need to differentiate between those cases. In the following, we call the player that updates after her turn *player X*, and the other one *player Y*. Although the definition of safety games seems to be of asymmetric nature (player B tries to *reach* a final configuration, while player A tries to *avoid* them), the proof does not rely on the exact identity of player X and Y.

In this section, given a configuration c, we write $\bar{\mathsf{c}}$ to denote the unique configuration obtained from c after updating all messages to the memory. More formally, $\mathsf{c} \xrightarrow{\mathtt{up}^*} \bar{\mathsf{c}}$ and all buffers of $\bar{\mathsf{c}}$ are empty.

Let $\mathcal{G} = \langle \mathsf{C}, \mathsf{C}_A, \mathsf{C}_B, \rightarrow, \mathsf{C}_F \rangle$ be a TSO game as described above, currently in some configuration $\mathsf{c}_0 \in \mathsf{C}$. We first consider the situation where player X has a winning strategy σ_X from c_0. Let σ_Y be an

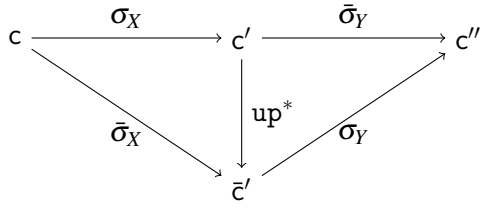

Figure 3: Commutative diagram of strategies in games of group I.

arbitrary strategy for player Y and define two more strategies $\bar{\sigma}_X : \mathsf{c} \mapsto \overline{\sigma_X(\mathsf{c})}$ and $\bar{\sigma}_Y : \mathsf{c} \mapsto \sigma_Y(\bar{\mathsf{c}})$. That is, they act like σ_X and σ_Y, respectively, with the addition that $\bar{\sigma}_X$ empties the buffer *after* each turn and $\bar{\sigma}_Y$ empties the buffer *before* each turn. From the definitions it follows directly that $\bar{\sigma}_Y(\sigma_X(\mathsf{c})) = \sigma_Y(\bar{\sigma}_X(\mathsf{c}))$ for all $\mathsf{c} \in \mathsf{C}_X$. An example can be seen in Figure 3.

We argue that $\bar{\sigma}_X$ is a winning strategy for player X. The intuition behind this is as follows: Using the notation of Figure 3, if a configuration c'' is reachable from $\bar{\mathsf{c}}'$, then it is also reachable from c', since player Y can empty all buffers at the start of her turn and then proceed as if she started in $\bar{\mathsf{c}}'$. On the other hand, there might be configurations reachable from c' but not $\bar{\mathsf{c}}'$, for example a read transition might get disabled by one of the buffer updates. Thus, player X never gets a disadvantage by emptying the buffers.

Claim 4. *$\bar{\sigma}_X$ is a winning strategy from c_0.*

Proof. **Case $\mathsf{c}_0 \in \mathsf{C}_X$:** Since $\bar{\sigma}_Y(\sigma_X(\mathsf{c})) = \sigma_Y(\bar{\sigma}_X(\mathsf{c}))$ for all $\mathsf{c} \in \mathsf{C}_X$, the plays $\mathsf{P}_1 = \mathsf{P}(\mathsf{c}_0, \sigma_X, \bar{\sigma}_Y)$ and $\mathsf{P}_2 = \mathsf{P}(\mathsf{c}_0, \bar{\sigma}_X, \sigma_Y)$ agree on every second configuration, i.e. the configurations in C_X. Moreover, the configurations in between (after an odd number of steps) at least share the same global state, i.e. $\mathcal{S}(\sigma_X(\mathsf{c})) = \mathcal{S}(\bar{\sigma}_X(\mathsf{c}))$. In particular, the sequence of visited global TSO states is the same in both plays. Since σ_X is a winning strategy from c_0, it means that P_1 is winning for player X. This means that P_2 is also winning, because for both players, a winning play is clearly determined by the sequence of visited global TSO states. Because we chose σ_Y arbitrarily, it follows that $\bar{\sigma}_X$ is a winning strategy.

Case $\mathsf{c}_0 \in \mathsf{C}_Y$: For the other case, we consider the configurations in $\mathrm{Post}(\mathsf{c}_0)$ instead. We observe that σ_X must be a winning strategy for all $\mathsf{c} \in \mathrm{Post}(\mathsf{c}_0)$. We apply the first case of this proof to each of these configurations and obtain that $\bar{\sigma}_X$ is a winning strategy for all of them. It follows that $\bar{\sigma}_X$ is a winning strategy for c_0. □

Suppose that player X plays her modified strategy as described above. We observe that after at most two steps, every play induced by her strategy and an arbitrary strategy of the opposing player only visits configurations with at most one message in the buffers: Player X will empty all buffers at the end of each of her turns and player Y can only add at most one message to the buffers in between. Hence, they can play on a finite set of configurations instead.

To show this, we construct a finite game $\mathcal{G}' = \langle \mathsf{C}', \mathsf{C}'_A, \mathsf{C}'_B, \to', \mathsf{C}'_F \rangle$ as follows. C'_Y contains all configurations of C_Y that have at most one buffer message, i.e. $\{\langle \mathcal{S}, \mathcal{B}, \mathcal{M} \rangle_Y \in \mathsf{C}_Y \mid \sum_{\iota \in \mathcal{I}} |\mathcal{B}(\iota)| \leq 1\}$. If $\mathsf{c}_0 \in \mathsf{C}_Y$, we also add it to C'_Y, otherwise to C'_X. Lastly, we add $\mathrm{Post}(\mathsf{C}'_Y)$ to C'_X, where Post is with respect to $\mathcal{G}$. $\to'$ is defined as the restriction of $\to$ to configurations of $\mathcal{G}'$, and $\mathsf{C}'_F = \mathsf{C}_F \cap \mathsf{C}'_A$. Note that C'_X also contains configurations with two messages. This is needed to account for the case that player Y has a winning strategy, which is handled later in this proof. Now, let $\bar{\sigma}'_X$ be the restriction of $\bar{\sigma}_X$ to C'_X (in the mathematical sense, i.e $\bar{\sigma}'_X : \mathsf{C}'_X \to \mathsf{C}_Y$ and $\bar{\sigma}_X(\mathsf{c}) = \bar{\sigma}'_X(\mathsf{c})$ for all $\mathsf{c} \in \mathsf{C}'_X$).

Claim 5. *$\bar{\sigma}'_X$ is a winning strategy for c_0 in $\mathcal{G}'$.*

Proof. Looking at the definitions, we confirm that $\bar{\sigma}'_X$ actually is a valid strategy for $\mathcal{G}'$, i.e. $\bar{\sigma}'_X(\mathsf{c}) \in \mathsf{C}'_Y$, for all $\mathsf{c} \in \mathsf{C}'_X$, since $\bar{\sigma}'_X(\mathsf{c})$ has empty buffers. (This makes $\bar{\sigma}'_X$ the restriction of $\bar{\sigma}_X$ to $\mathcal{G}'$.) Consider a strategy σ'_Y for player Y in $\mathcal{G}'$ and an arbitrary extension σ_Y to $\mathcal{G}$. Because $\bar{\sigma}'_X$ and $\bar{\sigma}_X$ agree on C'_X and $\bar{\sigma}'_Y$ and $\bar{\sigma}_Y$ agree on C'_Y, $\mathsf{P} = \mathsf{P}(\mathsf{c}_0, \bar{\sigma}'_X, \bar{\sigma}_Y)$ and $\mathsf{P}' = \mathsf{P}(\mathsf{c}_0, \bar{\sigma}'_X, \bar{\sigma}_Y)$ are in fact the exact same play. Since $\bar{\sigma}_X$ is a winning strategy, P is a winning play, and thus also P'. Here, note that $\mathcal{G}$ and $\mathcal{G}'$ agree on the final configurations within C'. Since σ'_Y was arbitrary, it follows that $\bar{\sigma}'_X$ is a winning strategy from c_0 in $\mathcal{G}'$. □

What is left to show is that a winning strategy for $\mathcal{G}'$ induces a winning strategy for $\mathcal{G}$. Suppose σ'_X is a winning strategy for player X in game $\mathcal{G}'$ for the configuration c_0. Let σ_X be an arbitrary extension of σ'_X to $\mathcal{G}$.

Claim 6. *σ_X is a winning strategy for c_0 in $\mathcal{G}$.*

Proof. Let σ_Y be a strategy of player Y in $\mathcal{G}$ and σ'_Y the restriction of σ_Y to C'_Y (again, in the mathematical sense). Since the outgoing transitions of every $\mathsf{c} \in \mathsf{C}'_Y$ are the same in both $\mathcal{G}$ and $\mathcal{G}'$, σ'_Y is a strategy for $\mathcal{G}'$ (and the restriction of σ_Y to $\mathcal{G}'$). Furthermore, starting from c_0, we see that σ_X and σ_Y induce the exact same play in $\mathcal{G}$ as σ'_X and σ'_Y in $\mathcal{G}'$. Since the former play is winning, so must be the latter one. □

Now, we quickly cover the situation where it is player Y that has a winning strategy. We follow the same arguments as previously, with minor changes. This time, assume σ_Y to be a winning strategy and let σ_X be arbitrary. Define $\bar{\sigma}_X$ and $\bar{\sigma}_Y$ as above. Following the beginning of the proof of Claim 4, we can conclude that the sequence of visited global TSO states is the same in both play P_1 and P_2. For the remainder of the proof, we swap the roles of X and Y and obtain that $\bar{\sigma}_Y$ is a winning strategy.

Let $\bar{\sigma}'_Y$ be the restriction of $\bar{\sigma}_Y$ to C'_Y. Since $\bar{\sigma}'_Y(\mathsf{C}'_Y) = \bar{\sigma}_Y(\mathsf{C}'_Y) \subseteq \mathrm{Post}(\mathsf{C}'_Y) \subseteq \mathsf{C}'_X$, it follows that $\bar{\sigma}'_Y$ is a strategy of $\mathcal{G}'$ (Post is again with respect to $\mathcal{G}$). Consider a strategy σ'_X for player X in $\mathcal{G}'$ and an arbitrary extension σ_X to $\mathcal{G}$. Similar as in Claim 5, we see that $\mathsf{P}(\mathsf{c}_0, \bar{\sigma}'_X, \bar{\sigma}_Y) = \mathsf{P}(\mathsf{c}_0, \bar{\sigma}'_X, \bar{\sigma}_Y)$ and conclude that $\bar{\sigma}'_Y$ is a winning strategy.

The other direction follows from the proof of Claim 6, with the roles of X and Y swapped.

Theorem 7. *The safety problem for games of group I is* EXPTIME*-complete.*

Proof. By Claim 4 and Claim 5, if a configuration c_0 is winning for player X in $\mathcal{G}$, then it is also winning in $\mathcal{G}'$. The reverse holds by Claim 6. The equivalent statement for player Y follows from results outlined above. Thus, the safety problem for $\mathcal{G}$ is equivalent to the safety problem for $\mathcal{G}'$. $\mathcal{G}'$ is finite and has exponentially many configurations. EXPTIME-completeness follows immediately from Lemma 2 (membership) and Theorem 3 (hardness). □

Remark 8. In the game where both players are allowed to update the buffer at any time, we can show an interesting conclusion. By Claim 4 and the equivalent statement for the second player, we can restrict both players to strategies that empty the buffer after each turn. Thus, the game is played only on configurations with empty buffer, except for the initial configuration which might contain some buffer messages. This implies that the TSO program that is described by the game implicitly follows SC semantics.

6 Group II

This group contains TSO games where both players are allowed to update the buffer *only* before their own move. Let player X be the player that has a winning strategy and player Y her opponent. Note that

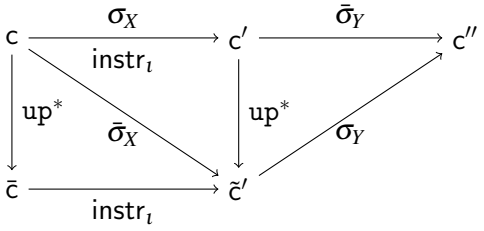

Figure 4: Commutative diagram of strategies in games of group II, in the case where $\mathsf{instr}_\iota \neq \mathtt{rd}(\mathtt{x},\mathtt{d})$.

this differs from the previous section, in which the players X and Y were defined based on their updating capabilties.

Similar to the argumentation for Group I, we want to show that player X also has a winning strategy where she empties the buffer in each move. But, in contrast to before, this time there is an exception: Since the player has to update the buffer *before* her move, by updating a memory variable she might disable a read transition that she intended to execute. Thus, we do not require her to empty the buffer in that case.

Formally, let $\mathcal{G} = \langle \mathsf{C}, \mathsf{C}_X, \mathsf{C}_Y, \rightarrow, \mathsf{C}_F \rangle$ be a TSO game where both players are allowed to perform buffer updates exactly before their own moves. Suppose σ_X is a winning strategy for player X and some configuration c_0. We construct another strategy $\bar{\sigma}_X$ for player X. Let $\mathsf{c} \in \mathsf{C}_X$, $\mathsf{c}' = \sigma_X(\mathsf{c})$ and $\bar{\mathsf{c}}$ as in the previous section, i.e. the unique configuration such that $\mathsf{c} \xrightarrow{\mathtt{up}^*}_{\mathcal{P}} \bar{\mathsf{c}}$ and the buffers of c are empty. Suppose that $\mathsf{c} \xrightarrow{\mathsf{instr}_\iota}_{\mathcal{P}} \mathsf{c}'$, where instr_ι is not a read instruction. Then, starting from c, updating all buffer messages does not change that the transition from $\mathcal{S}(\mathsf{c})(\iota)$ to $\mathcal{S}(\mathsf{c}')(\iota)$ is enabled. Thus, instr_ι can also be executed from $\bar{\mathsf{c}}$. We call the resulting configuration $\tilde{\mathsf{c}}'$ and observe that $\bar{\mathsf{c}} \rightarrow_{\mathcal{P}} \tilde{\mathsf{c}}'$ and $\mathsf{c}' \xrightarrow{\mathtt{up}^*} \tilde{\mathsf{c}}'$. We define $\bar{\sigma}_X(\mathsf{c}) = \tilde{\mathsf{c}}'$. This can be seen in Figure 4. Note that $\tilde{\mathsf{c}}'$ may have at most one message in its buffers. In the other case, where there is no transition from c to c' other than read instructions, we define $\bar{\sigma}_X(\mathsf{c}) = \sigma_X(\mathsf{c}) = \mathsf{c}'$.

Claim 9. *$\bar{\sigma}_X$ is a winning strategy for c_0.*

Proof. First, suppose that $\mathsf{c}_0 \in \mathsf{C}_X$ and let σ_Y be an arbitrary strategy of player Y. We define another (non-positional) strategy $\bar{\sigma}_Y$, that depends on the last two configurations, by $\bar{\sigma}_Y(\mathsf{c},\mathsf{c}') = \sigma_Y(\bar{\sigma}_X(\mathsf{c}))$. We observe that for all $\mathsf{c} \in \mathsf{C}_X$, it holds that $\bar{\sigma}_Y(\mathsf{c}, \sigma_X(\mathsf{c})) = \sigma_Y(\bar{\sigma}_X(\mathsf{c}))$. It follows that the play P_1 induced by σ_X and $\bar{\sigma}_Y$ and the play P_2 induced by $\bar{\sigma}_X$ and σ_Y agree on every second configuration, i.e. the configurations in C_X. In particular, the sequence of visited global TSO configurations is the same in both plays. Since σ_X is winning, it means that P_1 is winning for player X and thus also P_2 is winning. Because we chose σ_Y arbitrarily, it follows that $\bar{\sigma}_X$ is a winning strategy.

Otherwise, if $\mathsf{c}_0 \in \mathsf{C}_Y$, we consider the successors of c_0 instead. We note that $\bar{\sigma}_X$ must also be a winning strategy for each $\mathsf{c} \in \mathrm{Post}(\mathsf{c}_0)$. But then, we can apply the previous arguments to each of those configurations and conclude that $\bar{\sigma}_X$ is a winning strategy for all of them. Thus, it is also a winning strategy for c_0. □

We conclude that if player X has a winning strategy σ_X, then she also has a winning strategy $\bar{\sigma}_X$ where she empties the buffers before every turn in which she does not perform a read operation. By symmetry, the same holds true for player Y. Thus, we can limit our analysis to this type of strategies. We see that the number of messages in the buffers is bounded: Suppose that the game is in configuration

$c \in C_X$. Then, $\bar{\sigma}_X$ either empties the buffer and adds at most one new message, or it performs a transition due to a read instruction, which does not increase the size of the buffers. The analogous argumentation holds for player Y. Hence, we can reduce the game to a game on bounded buffers, which is finite state and thus decidable.

Given the configuration c_0 as above, we construct a finite game $\mathcal{G}' = \langle C', C'_X, C'_Y, \rightarrow', C'_F \rangle$ as follows. The set C'_X contains all configurations from C_X which have at most as many buffer messages than c_0 (or at most one message, if c_0 has empty buffers): $C'_X = \{c \in C_X \mid |\mathcal{B}(c)| \leq \max\{1, |\mathcal{B}(c_0)|\}\}$, where $|\mathcal{B}| = \sum_{\iota \in \mathcal{I}} |\mathcal{B}(\iota)|$. The set C'_Y is defined accordingly. Note that both sets are finite. Lastly, $\rightarrow'$ is defined as the restriction of $\rightarrow$ to configurations of $\mathcal{G}'$, and $C'_F = C_F \cap C'_A$. We define $\bar{\sigma}'_X$ to be the restriction of $\bar{\sigma}_X$ to C'_X. Since $\bar{\sigma}'_X(c) \in C'_Y$ for all $c \in C'_X$, $\bar{\sigma}'_X$ is indeed a valid strategy for $\mathcal{G}'$. In particular, it is the restriction of $\bar{\sigma}_X$ to $\mathcal{G}'$.

Claim 10. *$\bar{\sigma}'_X$ is a winning strategy for c_0 in $\mathcal{G}'$.*

Proof. First, consider the case where $c_0 \in C_X$. Let σ'_Y be a strategy for player Y in $\mathcal{G}'$ and let σ_Y be an arbitrary extension of σ'_Y to $\mathcal{G}$. The play P induced by $\bar{\sigma}_X$ and σ_Y in $\mathcal{G}$ is the same as the play P' induced by $\bar{\sigma}'_X$ and σ'_Y in $\mathcal{G}'$. Since $\bar{\sigma}_X$ is a winning strategy, P is a winning play. It follows that P' must also be a winning strategy. Since σ'_Y was arbitrary, it follows that $\bar{\sigma}'_X$ is a winning strategy and c_0 is winning in $\mathcal{G}'$. □

Theorem 11. *The safety problem for games of group II is* EXPTIME*-complete.*

Proof. By Claim 9 and Claim 10, if a configuration c_0 is winning for player A in game $\mathcal{G}$, then it is also winning in $\mathcal{G}'$. The same holds true for player B. Thus, the safety problem for $\mathcal{G}$ is equivalent to the safety problem for $\mathcal{G}'$. Similar to the games of group I, $\mathcal{G}'$ is finite and has exponentially many configurations. By Lemma 2 and Theorem 3, we can again conclude that the safety problem is EXPTIME-complete. □

7 Group III

This group consists of all games where exactly one player has control over the buffer updates, and additionally the game where both players are allowed to update buffer messages *after* their own move. Intuitively, all of them have in common that the TSO program can attribute a buffer update to one specific player. If only one player can update messages, this is clear. In the other game, the first player who observes that a buffer message has reached the memory is not the one who has performed the buffer update. Thus, the program is able to punish misbehaviour, i.e. not following protocols or losing messages.

We will show that the safety problem is undecidable for this group of games. To accomplish that, we reduce the state reachability problem of PCS to the safety problem of each game. Since the former problem is undecidable, so is the latter.

The case where player A is allowed to perform buffer updates at any time is called the *A-TSO game*. It is explained in detail in the following. The other cases work similar, but require slightly different program constructions. They are presented in the appendix [28].

Consider the A-TSO game, i.e. the case where player A can update messages at any time, but player B can never do so. Given a PCS $\mathcal{L} = \langle S, M, \rightarrow_{\mathcal{L}} \rangle$ and a final state $s_F \in S$, we construct a TSO program $\mathcal{P}$ that simulates $\mathcal{L}$. We design the program such that s_F is reachable in $\mathcal{L}$ if and only if player B wins the safety game induced by $\mathcal{P}$. Thus, the construction gives her the initiative to decide which transitions of $\mathcal{L}$ will be simulated. Meanwhile, the task of player A is to take care of the buffer updates.

$\mathcal{P}$ consists of three processes Proc^1, Proc^2 and Proc^3, that operate on the variables $\{\mathtt{x}_{\mathtt{wr}}, \mathtt{x}_{\mathtt{rd}}, \mathtt{y}\}$ over the domain $\mathsf{M} \uplus \{0, 1, \bot\}$. The first process simulates the control flow and the message channel of the PCS $\mathcal{L}$. The second process provides a mean to read from the channel. The only task of the third process is to prevent deadlocks, or rather to make any deadlocked player lose. Proc^3 achieves this with four states: the initial state, an intermediate state, and one winning state for each player, respectively. If one of the players cannot move in both Proc^1 and Proc^2, they have to take a transition in Proc^3. From the initial state of this process, there exists only one outgoing transition, which is to the intermediate state. From there, the other player can move to her respective winning state and the process will only self-loop from then on. For player A, her state is winning because she can refuse to update any messages, which will ensure that player B keeps being deadlocked in Proc^1 and Proc^2. For player B, her state simply is contained in $\mathsf{Q}_F^{\mathcal{P}}$. In the following, we will mostly omit Proc^3 from the analysis and just assume that both players avoid reaching a configuration where they cannot take any transition in either Proc^1 or Proc^2.

As mentioned above, we will construct Proc^1 and Proc^2 to simulate the perfect channel system in a way that gives player B the control about which channel operation will be simulated. To achieve this, each channel operation will need an even number of transitions to be simulated in $\mathcal{P}$. Since player B starts the game, this means that after every fully completed simulation step, it is again her turn and she can initiate another simulation step as she pleases. Furthermore, during the simulation of a skip or send operation, we want to prevent player A from executing Proc^2, since this process is only needed for the receive operation. Suppose that we want to block player A from taking a transition $\mathsf{q} \xrightarrow{\mathsf{instr}}_{\mathcal{P}} \mathsf{q}'$. We add a new transition $\mathsf{q}' \xrightarrow{\mathtt{skip}}_{\mathcal{P}} \mathsf{q}_F$, where $\mathsf{q}_F \in \mathcal{S}_F^{\mathcal{P}}$. Hence, reaching q' is immediately losing for player A, since player B can respond by moving to q_F.

Next, we will describe how Proc^1 and Proc^2 simulate the perfect channel system $\mathcal{L}$. For each transition in $\mathcal{L}$, we construct a sequence of transitions in Proc^1 that simulates both the state change and the channel behaviour of the $\mathcal{L}$-transition. To achieve this, Proc^1 uses its buffer to store the messages of the PCS's channel. In particular, to simulate a send operation $!\mathsf{m}$, Proc^1 adds the message $\langle \mathtt{x}_{\mathtt{wr}}, \mathsf{m} \rangle$ to its buffer. For receive operations, Proc^1 cannot read its own oldest buffer message, since it is overshadowed by the more recent messages. Thus, the program uses Proc^2 to read the message from memory and copies it to the variable $\mathtt{x}_{\mathtt{rd}}$, where it can be read by Proc^1. We call the combination of reading a message m from $\mathtt{x}_{\mathtt{wr}}$ and writing it to $\mathtt{x}_{\mathtt{rd}}$ the *rotation* of m.

While this is sufficient to simulate all behaviours of the PCS, it also allows for additional behaviour that is not captured by $\mathcal{L}$. More precisely, we need to ensure that each channel message is received *once and only once*. Equivalently, we need to prevent the *loss* and *duplication* of messages. This can happen due to multiple reasons.

The first phenomenon that allows the loss of messages is the seeming lossiness of the TSO buffer. Although it is not strictly lossy, it can appear so: Consider an execution of $\mathcal{P}$ that simulates two send operations $!\mathsf{m}_1$ and $!\mathsf{m}_2$, i.e. Proc^1 adds $\langle \mathtt{x}_{\mathtt{wr}}, \mathsf{m}_1 \rangle$ and $\langle \mathtt{x}_{\mathtt{wr}}, \mathsf{m}_2 \rangle$ to its buffer. Assume that player A decides to update both messages to the memory, without Proc^2 performing a message rotation in between. The first message m_1 is overwritten by the second message m_2 and is lost beyond recovery.

To prevent this, we extend the construction of Proc^1 such that it inserts an auxiliary message $\langle \mathtt{y}, 1 \rangle$ into its buffer after the simulation of each send operation. After a message rotation, that is, after Proc^2 copied a message from $\mathtt{x}_{\mathtt{wr}}$ to $\mathtt{x}_{\mathtt{rd}}$, the process then resets the value of $\mathtt{x}_{\mathtt{wr}}$ to its initial value $\bot$. Next, the process checks that $\mathtt{y}$ contains the value 0, which indicates that only one message was updated to the memory. Now, player A is allowed to update exactly one $\langle \mathtt{y}, 1 \rangle$ buffer message, after which Proc^2 resets $\mathtt{y}$ to 0. To ensure that player A has actually updated only one message in this step, Proc^2 then checks that $\mathtt{x}_{\mathtt{wr}}$ is still empty. Since player A is exclusively responsible for buffer updates, Proc^2 deadlocks her

whenever one of these checks fails.

In the next scenario, we discover a different way of message loss. Consider again an execution of $\mathcal{P}$ that simulates two send operations $!m_1$ and $!m_2$. Assume Player A updates m_1 to the memory and Proc^2 performs a message rotation. Immediately afterwards, the same happens to m_2, without Proc^1 simulating a receive operation in between. Again, m_1 is overwritten by m_2 before being received, thus it is lost.

Player A is prevented from losing a message in this way by disallowing her to perform a complete message rotation (including the update of one $\langle y,1\rangle$-message and the reset of the variables) entirely on her own. More precisely, we add a winning transition for player B to Proc^2 that she can take if and only if player A is the one initiating the update of $\langle y,1\rangle$. On the other hand, player A can prevent player B from performing two rotations right after each other by refusing to update the next buffer message until Proc^1 initiates the simulation of a receive operation.

Lastly, we investigate message duplication. This occurs if Proc^1 simulates two receive operations without Proc^2 performing a message rotation in between. In this case, the most recently rotated message is received twice.

The program prevents this by blocking Proc^1 from progressing after a receive operation until Proc^2 has finished a full rotation. In detail, at the very end of the message rotation and $\langle y,1\rangle$-update, Proc^2 reset the value of $x_{\mathtt{rd}}$ to its initial value $\perp$. After simulating a receive operation, Proc^1 is blocked until it can read this value from memory.

This concludes the mechanisms implemented to ensure that each channel message is received *once and only once*. Thus, we have constructed an A-TSO game that simulates a perfect channel system. We summarise our results in the following theorem. The formal proof can be found in Appendix B [28].

Theorem 12. *The safety problem for the A-TSO game is undecidable.*

8 Group IV

In TSO games where no player is allowed to perform any buffer updates, there is no communication between the processes at all. A read operation of a process Proc^ι on a variable x either reads the initial value from the shared memory, or the value of the last write of Proc^ι on x from the buffer, if such a write operation has happened.

Thus, we are only interested in the transitions that are enabled for each process, but we do not need to care about the actual buffer content. In particular, the information that we need to capture from the buffers and the memory is the values that each process can read from the variables, and whether a process can execute a memory fence instruction or not. Together with the global state of the current configuration, this completely determines the enabled transitions in the system.

We call this concept the *view* of the processes on the concurrent system and define it formally as a tuple $v = \langle \mathcal{S}, \mathcal{V}, \mathcal{F}\rangle$, where:

- $\mathcal{S} : \mathcal{I} \to \bigcup_{\iota\in\mathcal{I}} Q^\iota$ is a global state of $\mathcal{P}$.
- $\mathcal{V} : \mathcal{I} \times \mathsf{Vars} \to \mathsf{Dom}$ defines which value each process reads from a variable.
- $\mathcal{F} : \mathcal{I} \to \{\mathsf{true}, \mathsf{false}\}$ represents the possibility to perform a memory fence instruction.

Given a view $v = \langle \mathcal{S}, \mathcal{V}, \mathcal{F}\rangle$, we write $\mathcal{S}(v)$, $\mathcal{V}(v)$ and $\mathcal{F}(v)$ for the global program state $\mathcal{S}$, the value state $\mathcal{V}$ and the fence state $\mathcal{F}$ of v.

The view of a configuration c is denoted by $v(c)$ and defined in the following way. First, $\mathcal{S}(v(c)) = \mathcal{S}(c)$. For all $\iota \in \mathcal{I}$ and $x \in \mathsf{Vars}$, if $\mathcal{B}(c)(\iota)|_{\{x\}\times\mathsf{Dom}} = \langle x, d\rangle \cdot w$, then $\mathcal{V}(v(c))(\iota, x) = d$. Otherwise,

$\mathcal{V}(\mathsf{v}(\mathsf{c}))(\iota,\mathsf{x}) = \mathcal{M}(\mathsf{c})(\mathsf{x})$. Lastly, $\mathcal{F}(\mathsf{v}(\mathsf{c}))(\iota) = \mathsf{true}$ if and only if $\mathcal{B}(\mathsf{c})(\iota) = \varepsilon$. We extend the notation to sets of configurations in the usual way, i.e. $\mathsf{v}(\mathsf{C}') = \{\mathsf{v}(\mathsf{c}) \mid \mathsf{c} \in \mathsf{C}'\}$.

For $\mathsf{c},\mathsf{c}' \in \mathsf{C}_{\mathcal{P}}$, if $\mathsf{v}(\mathsf{c}) = \mathsf{v}(\mathsf{c}')$, then we write $\mathsf{c} \equiv \mathsf{c}'$ and say that c and c' are *view-equivalent*. In such a case, a local process of $\mathcal{P}$ cannot differentiate between c and c' in the sense that the enabled transitions in both configurations are the same. Lemma 13 captures this idea formally.

Lemma 13. *For all* $\mathsf{c}_1,\mathsf{c}_2,\mathsf{c}_3 \in \mathsf{C}_{\mathcal{P}}$, $\iota \in \mathcal{I}$ *and* $\mathsf{instr} \in \mathsf{Instrs}$ *with* $\mathsf{c}_1 \xrightarrow{\mathsf{instr}_\iota} \mathsf{c}_2$ *and* $\mathsf{c}_1 \equiv \mathsf{c}_3$, *there exists a* $\mathsf{c}_4 \in \mathsf{C}_{\mathcal{P}}$ *such that* $\mathsf{c}_3 \xrightarrow{\mathsf{instr}_\iota} \mathsf{c}_4$ *and* $\mathsf{c}_2 \equiv \mathsf{c}_4$.

Proof. We first show that instr_ι is enabled at c_3. Since $\mathsf{c}_1 \equiv \mathsf{c}_3$, it holds that $\mathcal{S}(\mathsf{c}_1) = \mathcal{S}(\mathsf{c}_3)$. Furthermore, if $\mathsf{instr}_\iota = \mathtt{rd}(\mathsf{x},\mathsf{d})_\iota$, then $\mathcal{V}(\mathsf{v}(\mathsf{c}_1))(\iota,\mathsf{x}) = \mathcal{V}(\mathsf{v}(\mathsf{c}_3))(\iota,\mathsf{x}) = \mathsf{d}$. Also, if $\mathsf{instr}_\iota = \mathtt{mf}_\iota$, then $\mathcal{F}(\mathsf{v}(\mathsf{c}_1))(\iota) = \mathcal{F}(\mathsf{v}(\mathsf{c}_3))(\iota) = \varepsilon$. From these considerations and the definition of the TSO semantics (see Figure 2), it follows that instr_ι is indeed enabled at c_3.

Let c_4 be the configuration obtained after performing instr_ι, i.e. $\mathsf{c}_3 \xrightarrow{\mathtt{rd}(\mathsf{x},\mathsf{d})_\iota} \mathsf{c}_4$. It holds that $\mathcal{S}(\mathsf{c}_4) = \mathcal{S}(\mathsf{c}_2) = \mathcal{S}(\mathsf{c}_1)[\iota \leftarrow \mathcal{S}(\mathsf{c}_2)(\iota)]$. If $\mathsf{instr}_\iota = \mathtt{wr}(\mathsf{x},\mathsf{d})_\iota$, then $\mathcal{V}(\mathsf{v}(\mathsf{c}_4)) = \mathcal{V}(\mathsf{v}(\mathsf{c}_2)) = \mathcal{V}(\mathsf{v}(\mathsf{c}_1))[(\iota,\mathsf{x}) \leftarrow \mathsf{d}]$ and $\mathcal{F}(\mathsf{v}(\mathsf{c}_4)) = \mathcal{F}(\mathsf{v}(\mathsf{c}_2)) = \mathcal{F}(\mathsf{v}(\mathsf{c}_1))[\iota \leftarrow \mathsf{false}]$. Otherwise, $\mathcal{V}(\mathsf{v}(\mathsf{c}_4)) = \mathcal{V}(\mathsf{v}(\mathsf{c}_2)) = \mathcal{V}(\mathsf{v}(\mathsf{c}_1))$ and $\mathcal{F}(\mathsf{v}(\mathsf{c}_4)) = \mathcal{F}(\mathsf{v}(\mathsf{c}_2)) = \mathcal{F}(\mathsf{v}(\mathsf{c}_1))$. In all cases it follows that $\mathsf{c}_2 \equiv \mathsf{c}_4$. □

We define a finite safety game played on TSO views and show that we can restrict our analysis to this game. Let $\mathcal{G} = \langle \mathsf{C}, \mathsf{C}_A, \mathsf{C}_B, \rightarrow, \mathsf{C}_F \rangle$ be a TSO game where neither player can perform any updates. We define a new game $\mathcal{G}' = \langle \mathsf{V}, \mathsf{V}_A, \mathsf{V}_B, \rightarrow', \mathsf{V}_F \rangle$ that is played on the views of $\mathcal{G}$. We define $\mathsf{V}_A = \{\mathsf{v}(\mathsf{c})_A \mid \mathsf{c}_A \in \mathsf{C}_A\}$, $\mathsf{V}_B = \{\mathsf{v}(\mathsf{c})_B \mid \mathsf{c}_B \in \mathsf{C}_B\}$, $\mathsf{V} = \mathsf{V}_A \cup \mathsf{V}_B$ and $\mathsf{V}_F = \{\mathsf{v}(\mathsf{c})_A \mid \mathsf{c}_A \in \mathsf{C}_F\}$. Lastly, $\mathsf{v}(\mathsf{c}) \rightarrow' \mathsf{v}(\mathsf{c}')$ if and only if $\mathsf{c} \rightarrow \mathsf{c}'$. This is well-defined by Lemma 13.

Lemma 14. *A configuration* $\mathsf{c}_0 \in \mathsf{C}$ *is winning (for player A / B) in* $\mathcal{G}$ *if and only if the view* $\mathsf{v}_0 = \mathsf{v}(\mathsf{c}_0) \in \mathsf{V}$ *is winning (for player A / B) in* $\mathcal{G}'$.

Proof. To simplify notation, we extend $\mathsf{v}(\mathsf{c})$ to configurations of TSO games by $\mathsf{v}(\mathsf{c}_A) = \mathsf{v}(\mathsf{c})_A$ and $\mathsf{v}(\mathsf{c}_B) = \mathsf{v}(\mathsf{c})_B$ for $\mathsf{c}_A \in \mathsf{C}_A$ and $\mathsf{c}_B \in \mathsf{C}_B$. Hence, we can write $\mathsf{V}_A = \mathsf{v}(\mathsf{C}_A)$ and similar.

Suppose c_0 is winning for some player X with (positional) strategy σ_X and consider the case $\mathsf{c}_0 \in \mathsf{C}_X$. In the following, we will define a (non-positional) strategy σ'_X for $\mathcal{G}'$.

First, we need an auxiliary function $f : \mathsf{C} \times \mathsf{V} \rightarrow \mathsf{C}$ that fulfills the condition: For all $\mathsf{c} \in \mathsf{C}$ and $\mathsf{v} \in \mathsf{V}$ such that $\mathsf{v}(\mathsf{c}) \rightarrow' \mathsf{v}$, it holds that $\mathsf{c} \rightarrow f(\mathsf{c},\mathsf{v})$ and $\mathsf{v} = \mathsf{v}(f(\mathsf{c},\mathsf{v}))$. Intuitively, f selects a successor of c with view v. Such a function exists by Lemma 13.

For n even and a sequence $\mathsf{v}_0,\ldots,\mathsf{v}_n$, iteratively define $\mathsf{c}_{2i-1} = \sigma(\mathsf{c}_{2i-2})$ and $\mathsf{c}_{2i} = f(\mathsf{c}_{2i-1},\mathsf{v}_{2i})$ for $i = 1,\ldots,n/2$. Then, $\sigma'_X(\mathsf{v}_0,\ldots,\mathsf{v}_n) = \mathsf{v}(\sigma_X(\mathsf{c}_n))$. We will show that σ'_X is a winning strategy for v_0. Consider a positional strategy σ'_Y for player Y in $\mathcal{G}'$. We define a positional strategy σ_Y for player Y in $\mathcal{G}$ by $\sigma_Y(\mathsf{c}) = f(\mathsf{c},\sigma'_Y(\mathsf{v}(\mathsf{c})))$. Consider the play $\mathsf{P} = \mathsf{c}_0,\mathsf{c}_1,\ldots$ induced by σ_X and σ_Y, and the play $\mathsf{P}' = \mathsf{v}_0,\mathsf{v}_1,\ldots$ induced by σ'_X and σ'_Y.

We proof by induction over k, that (i) $\mathsf{v}_k = \mathsf{v}(\mathsf{c}_k)$ and (ii) c_k of P coincides with c_k as in the definition of σ'_X. In this context, we refer to the latter with $\bar{\mathsf{c}}_k$. For $k = 0$, $\mathsf{v}_0 = \mathsf{v}(\mathsf{c}_0)$ and $\mathsf{c}_0 = \bar{\mathsf{c}}_0$ by definition. For k odd, $\mathsf{c}_k = \sigma_X(\mathsf{c}_{k-1}) = \bar{\mathsf{c}}_k$ by the induction hypothesis. Also,

$$\mathsf{v}_k = \sigma'_X(\mathsf{v}_0,\ldots,\mathsf{v}_{k-1}) = \mathsf{v}(\sigma_X(\bar{\mathsf{c}}_{k-1})) = \mathsf{v}(\sigma_X(\mathsf{c}_{k-1})) = \mathsf{v}(\mathsf{c}_k) \ .$$

For $k > 0$ even,

$$\mathsf{c}_k = \sigma_Y(\mathsf{c}_{k-1}) = f(\mathsf{c}_{k-1},\sigma'_Y(\mathsf{v}(\mathsf{c}_{k-1}))) = f(\bar{\mathsf{c}}_{k-1},\sigma'_Y(\mathsf{v}_{k-1})) = f(\bar{\mathsf{c}}_{k-1},\mathsf{v}_k) = \bar{\mathsf{c}}_k \ .$$

Lastly,

$$\mathsf{v}(\mathsf{c}_k) = \mathsf{v}(\sigma_Y(\mathsf{c}_{k-1})) = \mathsf{v}(f(\mathsf{c}_{k-1}, \sigma'_Y(\mathsf{v}(\mathsf{c}_{k-1})))) = \mathsf{v}(f(\mathsf{c}_{k-1}, \sigma'_Y(\mathsf{v}_{k-1}))) = \mathsf{v}(f(\mathsf{c}_{k-1}, \mathsf{v}_k)) = \mathsf{v}_k \ ,$$

where the last equality follows from the definition of f.

Since σ_X is a winning strategy for c_0, P is a winning play for player X. From the definition of V_F it follows that P' is a winning play in $\mathcal{G}'$ and thus v_0 is winning for player X. Note that by Lemma 1, we could have chosen a positional strategy in place of σ'_X. Since we did not put any restrictions on the identity of player X, this concludes both the *if* and the *only if* direction of the proof for the case $\mathsf{c}_0 \in \mathsf{C}_X$.

Otherwise, if $\mathsf{c}_0 \in \mathsf{C}_Y$, we consider all configurations of $\mathrm{Post}(\mathsf{c}_0)$ instead. We have the following chain of equivalences: c_0 is winning $\iff$ all $\mathsf{c} \in \mathrm{Post}(\mathsf{c}_0)$ are winning $\iff$ all $\mathsf{v} \in \mathsf{v}(\mathrm{Post}(\mathsf{c}_0))$ are winning $\iff$ all $\mathsf{v} \in \mathrm{Post}(\mathsf{v}(\mathsf{c}_0))$ are winning $\iff$ $\mathsf{v}(\mathsf{c}_0)$ is winning. Here, the second equivalence applies the first case of this proof and the third equivalence uses $\mathrm{Post}(\mathsf{v}(\mathsf{c}_0)) = \mathsf{v}(\mathrm{Post}(\mathsf{c}_0))$, which follows from the definition of $\mathcal{G}'$. □

Theorem 15. *The safety problem for games in group IV is* EXPTIME*-complete.*

Proof. By Lemma 14, the safety problem for $\mathcal{G}$ is equivalent to the safety problem of $\mathcal{G}'$, which is played on views. Since there exist only exponentially many views, EXPTIME-completeness follows from Lemma 2 and Theorem 3, similar to Group I and II. □

9 Conclusion and Future Work

In this work we have addressed for the first time the game problem for programs running under weak memory models in general and TSO in particular. Surprisingly, our results show that depending on when the updates take place, the problem can turn out to be undecidable or decidable. In fact, there is a subtle difference between the decidable (group I, II and IV) and undecidable (group III) TSO games. For the former games, when a player is taking a turn, the system does not know who was responsible for the last update. But for the latter games, the last update can be attributed to a specific player. Another surprising finding is the complexity of the game problem for the groups I, II and IV which is EXPTIME-complete in contrast with the non-primitive recursive complexity of the reachability problem for programs running under TSO and the undecidability of the repeated reachability problem.

In future work, the games where exactly one player has control over the buffer seem to be the most natural ones to expand on. In particular, the A-TSO game (where player A can update before and after her move) and the B-TSO game (same, but for player B). On the other hand, the games of groups I, II and IV seem to be degenerate cases and therefore rather uninteresting. In particular, they do not seem to be more powerful than games on programs that follow SC semantics.

Another direction for future work is considering other memory models, such as the partial store ordering semantics, the release-acquire semantics, and the ARM semantics. It is also interesting to define stochastic games for programs running under TSO as extension of the probabilistic TSO semantics [1].

References

[1] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Raj Aryan Agarwal, Adwait Godbole & Shankara Narayanan Krishna (2022): *Probabilistic Total Store Ordering*. In Ilya Sergey, editor: *Programming Languages and Systems - 31st European Symposium on Programming, ESOP 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings*, *Lecture Notes in Computer Science* 13240, Springer, pp. 317–345, doi:10.1007/978-3-030-99336-8_12.

[2] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Ahmed Bouajjani & Tuan Phong Ngo (2016): *The Benefits of Duality in Verifying Concurrent Programs under TSO*. In Josée Desharnais & Radha Jagadeesan, editors: *27th International Conference on Concurrency Theory, CONCUR 2016, August 23-26, 2016, Québec City, Canada*, *LIPIcs* 59, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 5:1–5:15, doi:10.4230/LIPIcs.CONCUR.2016.5.

[3] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Ahmed Bouajjani & Tuan Phong Ngo (2018): *A Load-Buffer Semantics for Total Store Ordering*. *Log. Methods Comput. Sci.* 14(1), doi:10.23638/LMCS-14(1:9)2018.

[4] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Carl Leonardsson & Ahmed Rezine (2012): *Counter-Example Guided Fence Insertion under TSO*. In Cormac Flanagan & Barbara König, editors: *Tools and Algorithms for the Construction and Analysis of Systems - 18th International Conference, TACAS 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*, *Lecture Notes in Computer Science* 7214, Springer, pp. 204–219, doi:10.1007/978-3-642-28756-5_15.

[5] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Florian Furbach, Adwait Amit Godbole, Yacoub G. Hendi, Shankara Narayanan Krishna & Stephan Spengler (2023): *Parameterized Verification under TSO with Data Types*. In Sriram Sankaranarayanan & Natasha Sharygina, editors: *Tools and Algorithms for the Construction and Analysis of Systems - 29th International Conference, TACAS 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Paris, France, April 22-27, 2023, Proceedings, Part I*, *Lecture Notes in Computer Science* 13993, Springer, pp. 588–606, doi:10.1007/978-3-031-30823-9_30.

[6] Parosh Aziz Abdulla, Mohamed Faouzi Atig & Ngo Tuan Phong (2015): *The Best of Both Worlds: Trading Efficiency and Optimality in Fence Insertion for TSO*. In Jan Vitek, editor: *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, *Lecture Notes in Computer Science* 9032, Springer, pp. 308–332, doi:10.1007/978-3-662-46669-8_13.

[7] Parosh Aziz Abdulla, Mohamed Faouzi Atig & Rojin Rezvan (2020): *Parameterized verification under TSO is PSPACE-complete*. *Proc. ACM Program. Lang.* 4(POPL), pp. 26:1–26:29, doi:10.1145/3371094.

[8] Parosh Aziz Abdulla, Ahmed Bouajjani & Julien d'Orso (2008): *Monotonic and Downward Closed Games*. *J. Log. Comput.* 18(1), pp. 153–169, doi:10.1093/logcom/exm062.

[9] Parosh Aziz Abdulla & Bengt Jonsson (1994): *Undecidable Verification Problems for Programs with Unreliable Channels*. In Serge Abiteboul & Eli Shamir, editors: *Automata, Languages and Programming, 21st International Colloquium, ICALP94, Jerusalem, Israel, July 11-14, 1994, Proceedings*, *Lecture Notes in Computer Science* 820, Springer, pp. 316–327, doi:10.1007/3-540-58201-0_78.

[10] Parosh Aziz Abdulla & Bengt Jonsson (1996): *Verifying Programs with Unreliable Channels*. *Inf. Comput.* 127(2), pp. 91–101, doi:10.1006/inco.1996.0053.

[11] ARM (2014): *ARM Architecture Reference Manual, ARMv7-A and ARMv7-R edition*. Available at https://developer.arm.com/documentation/ddi0406/latest/.

[12] Mohamed Faouzi Atig (2020): *What is decidable under the TSO memory model?* *ACM SIGLOG News* 7(4), pp. 4–19, doi:10.1145/3458593.3458595.

[13] Mohamed Faouzi Atig, Ahmed Bouajjani, Sebastian Burckhardt & Madanlal Musuvathi (2010): *On the verification problem for weak memory models*. In Manuel V. Hermenegildo & Jens Palsberg, editors: *Proceedings*

of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010, ACM, pp. 7–18, doi:10.1145/1706299.1706303.

[14] Ahmed Bouajjani, Egor Derevenetc & Roland Meyer (2013): *Checking and Enforcing Robustness against TSO*. In Matthias Felleisen & Philippa Gardner, editors: *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, *Lecture Notes in Computer Science* 7792, Springer, pp. 533–553, doi:10.1007/978-3-642-37036-6_29.

[15] Ahmed Bouajjani, Roland Meyer & Eike Möhlmann (2011): *Deciding Robustness against Total Store Ordering*. In Luca Aceto, Monika Henzinger & Jirí Sgall, editors: *Automata, Languages and Programming - 38th International Colloquium, ICALP 2011, Zurich, Switzerland, July 4-8, 2011, Proceedings, Part II*, *Lecture Notes in Computer Science* 6756, Springer, pp. 428–440, doi:10.1007/978-3-642-22012-8_34.

[16] Daniel Brand & Pitro Zafiropulo (1983): *On Communicating Finite-State Machines*. *J. ACM* 30(2), pp. 323–342, doi:10.1145/322374.322380.

[17] Ashok K. Chandra, Dexter Kozen & Larry J. Stockmeyer (1981): *Alternation*. *J. ACM* 28(1), pp. 114–133, doi:10.1145/322234.322243.

[18] Alain Finkel & Philippe Schnoebelen (2001): *Well-structured transition systems everywhere!* *Theor. Comput. Sci.* 256(1-2), pp. 63–92, doi:10.1016/S0304-3975(00)00102-X.

[19] Erich Grädel, Wolfgang Thomas & Thomas Wilke, editors (2002): *Automata, Logics, and Infinite Games: A Guide to Current Research [outcome of a Dagstuhl seminar, February 2001]*. *Lecture Notes in Computer Science* 2500, Springer, doi:10.1007/3-540-36387-4.

[20] IBM (2021): *Power ISA, Version 3.1b*. Available at https://files.openpower.foundation/s/dAYSdGzTfW4j2r2/download/OPF_PowerISA_v3.1B.pdf.

[21] Intel Corporation (2012): *Intel 64 and IA-32 Architectures Software Developers Manual*. Available at https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

[22] Leslie Lamport (1979): *How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs*. *IEEE Trans. Computers* 28(9), pp. 690–691, doi:10.1109/TC.1979.1675439.

[23] René Mazala (2001): *Infinite Games*. In Erich Grädel, Wolfgang Thomas & Thomas Wilke, editors: *Automata, Logics, and Infinite Games: A Guide to Current Research [outcome of a Dagstuhl seminar, February 2001]*, *Lecture Notes in Computer Science* 2500, Springer, pp. 23–42, doi:10.1007/3-540-36387-4_2.

[24] Scott Owens, Susmit Sarkar & Peter Sewell (2009): *A Better x86 Memory Model: x86-TSO*. In Stefan Berghofer, Tobias Nipkow, Christian Urban & Makarius Wenzel, editors: *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, *Lecture Notes in Computer Science* 5674, Springer, pp. 391–407, doi:10.1007/978-3-642-03359-9_27.

[25] Philippe Schnoebelen (2002): *Verifying lossy channel systems has nonprimitive recursive complexity*. *Inf. Process. Lett.* 83(5), pp. 251–261, doi:10.1016/S0020-0190(01)00337-4.

[26] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli & Magnus O. Myreen (2010): *x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors*. *Commun. ACM* 53(7), pp. 89–97, doi:10.1145/1785414.1785443.

[27] SPARC International, Inc. (1994): *SPARC Architecture Manual Version 9*. Available at https://sparc.org/wp-content/uploads/2014/01/SPARCV9.pdf.gz.

[28] Stephan Spengler & Sanchari Sil (2023): *TSO Games – On the decidability of safety games under the total store order semantics*, doi:10.48550/arXiv.2309.02862.