Reachability and Safety Games under TSO Semantics

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

We consider games played on the transition graph of concurrent programs running under the Total Store Order (TSO) weak memory model. Games are frequently used to model the interaction between a system and its environment, in this case between the concurrent processes and the nondeterministic TSO buffer updates. In our formulation, the game is played by two players, who alternatingly make a move: The *process player* can execute any enabled instruction of the processes, while the *update player* takes care of updating the messages in the buffers that are between each process and the shared memory. We show that the reachability and safety problem of this game reduce to the analysis of single-process (non-concurrent) programs. In particular, they exhibit only finite-state behaviour. Because of this, we introduce different notions of *fairness*, which force the two players to behave in a more realistic way. Both the reachability and safety problem then become undecidable.

1 Introduction

In concurrent programs, different processes interact with each other through the use of shared memory. Programmers usually unconsciously assume that the semantics adhere to the Sequential Consistency (SC) memory model [17]. In SC, the execution of processes can be interleaved, but write instructions are visible in the memory in the exact order in which they were issued. However, most modern architectures, such as Intel x86 [16], SPARC [22], IBM's POWER [15], and ARM [6], implement several relaxations and optimisations that improve memory access latency but break SC assumptions. A standard model that is weaker than SC allows the reordering of reads and writes of the same process, as long as it maintains the appearance of SC from the perspective of each individual process. The implementation of this optimisation adds an unbounded first-in-first-out write buffer between each process and the shared memory. The buffer is used to delay write operations. This model is called Total Store Ordering (TSO) and is a faithful formalisation of SPARC and Intel x86 [19, 21].

Verification under TSO semantics is difficult due to the unboundedness of the buffers. Even if each process can be modelled as a finite-state system, the program itself has a state space of infinite size. The reachability problem for programs running under TSO semantics is to decide whether a target program state is reachable from a given initial state during program execution. If the target state is considered to be a bad state, it is also called the safety problem. Solving reachability and safety helps in deciding if a program is correct, i.e. if it adheres to a specification or if it can avoid states of undefined behaviour. Using alternative but equivalent semantics, it has been shown that the reachability problem is decidable [9, 2, 1]. Furthermore, lossy channel system [4, 14, 3, 20] can be simulated by programs running under TSO semantics [9]. This implies that the reachability problem is non-primitive recursive [20] and the repeated reachability problem is undecidable [3]. Additionally, the termination problem has been shown to be decidable [8] using the framework of well-structured transition systems [14, 4].

In this paper, we consider games played on the transition graph of concurrent programs running under TSO semantics. Formal games provide a framework to reason about the behaviour of a system and

A. Achilleos and A. Francalanza (Eds.): Fifteenth
International Symposium on Games, Automata, Logics,
and Formal Verification (GandALF 2024).
EPTCS 409, 2024, pp. 154–171, doi:10.4204/EPTCS.409.14

the interaction between the system and its environment. In particular, they have been extensively used in controller synthesis problems [7, 10, 12, 11, 5]. A previous paper introduces safety games in which two players alternatingly execute instructions of a concurrent program [24]. Motivated by this work, we propose a game setting that more closely models the interplay between a system and the environment: The first player controls the execution of the program instructions, while the second player handles the nondeterministic updates of the store buffers to the shared memory. This model sees the process and the update mechanism as antagonistic, and allows us to reason about the correctness of the program regardless of the update behaviour.

We consider two types of game objectives: In a reachability game, the process player tries to reach a given set of target states, while the update player tries to avoid this; In a safety game, these two roles are reversed. We show that in both cases finding the winner of the game reduces to the analysis of games being played on a program with just one process. Furthermore, we show that these games are bisimilar to finite-state games and thus decidable. In particular, the reachability and safety problem are PSPACE-complete. The reason that the concurrent programs exhibit a finite-state character lies in the optimal behaviour of the two players. If the player that controls the processes has a winning strategy, then she can win by playing in only one process, ignoring all the other processes of the program. On the other hand, if the player controlling the buffer is able to win, she can do so by never letting any write operation reach the memory. In both cases, there is no concurrency in the sense that the processes do not interact or communicate with each other. This is not realistic, since we should be able to assume that if the program runs a sufficiently long duration (1) every process will be executed and (2) every write stored in the buffer will be updated to the memory.

We rectify this issue by introducing two fairness conditions. First, in an infinite run the process player must execute each enabled process infinitely many times. Second, the update player must make sure that each write operation reaches the memory after finitely many steps. We show that both the reachability and safety problem become undecidable with these restrictions. To do so, we use a reduction from perfect channel systems adapted from [24].

Finally, we investigate an alternative TSO semantics in our game setting. The authors of [1] propose a load-buffer semantics for TSO which reverts the direction of the information flow between the processes and the shared memory. In their model, the buffer is filled with values from the memory which can later be read by the process. Using well-structured transition systems, they showed that it is equivalent to the classical TSO semantics with respect to state reachability. We explore whether the equivalence also holds in the two-player game, but come to the conclusion that this is not the case. In particular, we construct a concurrent program that is won by the update player under store-buffer semantics but by the process player under load-buffer semantics.

2 Preliminaries

Transition Systems A *(labelled) transition system* is a triple $\mathcal{T} = \langle \mathsf{C}, \mathsf{L}, \to \rangle$, where C is a set of *configurations*, L is a set of *labels*, and $\to \subseteq \mathsf{C} \times \mathsf{L} \times \mathsf{C}$ is a *transition relation*. We usually write $\mathsf{c}_1 \xrightarrow{\mathsf{label}} \mathsf{c}_2$ if $\langle \mathsf{c}_1, \mathsf{label}, \mathsf{c}_2 \rangle \in \to$. Furthermore, we write $\mathsf{c}_1 \to \mathsf{c}_2$ 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 $\mathsf{c}_0 \xrightarrow{\mathsf{label}_1} \mathsf{c}_1 \xrightarrow{\mathsf{label}_2} \mathsf{c}_2 \dots \xrightarrow{\mathsf{label}_n} \mathsf{c}_n$. 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 define $\mathrm{Pre}(\mathsf{c}) = \{\mathsf{c}' \mid \mathsf{c}' \to \mathsf{c}\}$ and $\mathrm{Post}(\mathsf{c}) = \{\mathsf{c}' \mid \mathsf{c} \to \mathsf{c}'\}$. We extend these notions to sets of configurations C' with $\mathrm{Pre}(\mathsf{C}') = \bigcup_{\mathsf{c} \in \mathsf{C}'} \mathrm{Pre}(\mathsf{c})$ and $\mathrm{Post}(\mathsf{C}') = \bigcup_{\mathsf{c} \in \mathsf{C}'} \mathrm{Post}(\mathsf{c})$.

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

Perfect Channel Systems Given a set of messages M, define the set of channel operations $\mathsf{Op} = \{!\mathsf{m}, ?\mathsf{m} \mid \mathsf{m} \in \mathsf{M}\} \cup \{\mathtt{skip}\}$. A *perfect channel system* (PCS) is a triple $\mathcal{L} = \langle \mathsf{S}, \mathsf{M}, \delta \rangle$, where S is a set of states, M is a set of messages, and $\delta \subseteq \mathsf{S} \times \mathsf{Op} \times \mathsf{S}$ is a transition relation. We write $\mathsf{s}_1 \xrightarrow{\mathsf{op}} \mathsf{s}_2$ if $\langle \mathsf{s}_1, \mathsf{op}, \mathsf{s}_2 \rangle \in \delta$.

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* $!\mathsf{m}$, the channel system appends m to the tail of the channel. A transition $?\mathsf{m}$ 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 $\mathcal{T}_{\mathcal{L}} = \langle \mathsf{C}_{\mathcal{L}}, \mathsf{L}_{\mathcal{L}}, \rightarrow_{\mathcal{L}} \rangle$, where $\mathsf{C}_{\mathcal{L}} = \mathsf{S} \times \mathsf{M}^*$, $\mathsf{L}_{\mathcal{L}} = \mathsf{Op}$ and the transition relation $\rightarrow_{\mathcal{L}}$ is the smallest relation given by:

- If $\mathsf{s}_1 \xrightarrow{!\mathsf{m}} \mathsf{s}_2$ and $\mathsf{w} \in \mathsf{M}^*$, then $\langle \mathsf{s}_1, \mathsf{w} \rangle \xrightarrow{!\mathsf{m}}_{\mathcal{L}} \langle \mathsf{s}_2, \mathsf{m} \cdot \mathsf{w} \rangle$.
- If $\mathsf{s}_1 \xrightarrow{?\mathsf{m}} \mathsf{s}_2$ and $\mathsf{w} \in \mathsf{M}^*$, then $\langle \mathsf{s}_1, \mathsf{w} \cdot \mathsf{m} \rangle \xrightarrow{?\mathsf{m}}_{\mathcal{L}} \langle \mathsf{s}_2, \mathsf{w} \rangle$.
- If $\mathsf{s}_1 \xrightarrow{\mathtt{skip}} \mathsf{s}_2$ and $\mathsf{w} \in \mathsf{M}^*$, then $\langle \mathsf{s}_1, \mathsf{w} \rangle \xrightarrow{\mathtt{skip}}_{\mathcal{L}} \langle \mathsf{s}_2, \mathsf{w} \rangle$.

A state $\mathsf{s}_F \in \mathsf{S}$ is *reachable* from a configuration $\mathsf{c}_0 \in \mathsf{C}_{\mathcal{L}}$, if there exists a configuration $\mathsf{c}_F = \langle \mathsf{s}_F, \mathsf{w}_F \rangle$ 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 $\mathsf{c}_0 \in \mathsf{C}_{\mathcal{L}}$ and a final state $\mathsf{s}_F \in \mathsf{S}$, to decide whether s_F is reachable from c_0 in $\mathcal{T}_{\mathcal{L}}$. It is undecidable [13].

3 Concurrent Programs

Syntax Let Dom be a finite data domain and Vars be a finite set of shared variables over Dom. We define the *instruction set* $\mathsf{Instrs} = \{\mathtt{rd}(\mathsf{x}, \mathsf{d}), \mathtt{wr}(\mathsf{x}, \mathsf{d}) \mid \mathsf{x} \in \mathsf{Vars}, \mathsf{d} \in \mathsf{Dom}\} \cup \{\mathtt{skip}, \mathtt{mf}\}$, which are called *read*, *write*, *skip* and *memory fence*, respectively. A process is represented by a finite state labelled transition system. It is given as the triple $\mathsf{Proc} = \langle \mathsf{Q}, \mathsf{Instrs}, \delta \rangle$, where Q is a finite set of *local states* and $\delta \subseteq \mathsf{Q} \times \mathsf{Instrs} \times \mathsf{Q}$ is the transition relation. As with transition systems, we write $\mathsf{q}_1 \xrightarrow{\mathsf{instr}} \mathsf{q}_2$ if $\langle \mathsf{q}_1, \mathsf{instr}, \mathsf{q}_2 \rangle \in \delta$ and $\mathsf{q}_1 \rightarrow \mathsf{q}_2$ if there exists some instr such that $\mathsf{q}_1 \xrightarrow{\mathsf{instr}} \mathsf{q}_2$.

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 $\mathsf{Proc}^\iota = \langle \mathsf{Q}^\iota, \mathsf{Instrs}, \delta^\iota \rangle$. A *global* state of $\mathcal{P}$ is a function $\mathcal{S} : \mathcal{I} \rightarrow \bigcup_{\iota \in \mathcal{I}} \mathsf{Q}^\iota$ that maps each process to its local state, i.e $\mathcal{S}(\iota) \in \mathsf{Q}^\iota$.

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 $\mathtt{wr}(\mathsf{x}, \mathsf{d})$, 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

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 1: TSO semantics

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 nondeterministic 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 and $\mathsf{L}_\mathcal{P}=\{\mathsf{instr}_\iota \mid \mathsf{instr}\in\mathsf{Instrs},\iota\in\mathcal{I}\}\cup\{\mathtt{up}_\iota \mid \iota\in\mathcal{I}\}$ is the set of labels. The transition relation $\to_\mathcal{P}$ is given by the rules in Figure 1, 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. Furthermore, 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.

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 $\mathcal{T}_\mathcal{P}$.

4 Games

A *game* is an unlabelled transition system, 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. In

a *reachability game*, the goal of player A is to reach a given set of target states, while player B tries to avoid this. In a *safety game*, the roles are swapped.

Formally, a game is defined as a tuple $\mathcal{G} = \langle \mathsf{C}, \mathsf{C}_A, \mathsf{C}_B, \rightarrow \rangle$, where C is the set of configurations, C_A and C_B form a partition of C, and $\rightarrow$ is a transition relation on C. For the games considered in this paper, the relation will always be restricted to $\rightarrow \subseteq (\mathsf{C}_A \times \mathsf{C}_B) \cup (\mathsf{C}_B \times \mathsf{C}_A)$, which means that the two players take turns alternatingly.

Plays and Winning Conditions An *infinite play* P of $\mathcal{G}$ is an infinite sequence $\mathsf{c}_0, \mathsf{c}_1, \ldots$ such that $\mathsf{c}_i \rightarrow \mathsf{c}_{i+1}$ for all $i \in \mathbb{N}$. Similarly, a *finite play* is a finite sequence $\mathsf{c}_0, \mathsf{c}_1, \ldots, \mathsf{c}_n$ such that $\mathsf{c}_i \rightarrow \mathsf{c}_{i+1}$ for all $i \in [0, \ldots, n-1]$ and $\mathrm{Post}(\mathsf{c}_n) = \emptyset$, i.e. the play ends in a deadlock. A *winning condition* W is a subset of all infinite plays. We say that player A is the winner of a play, if either the play is infinite and an element of W, or if it is finite and ends in a deadlock for player B, i.e. $\mathsf{c}_n \in \mathsf{C}_B$. Otherwise, player B wins the play.

In this work, we will consider two types of winning conditions. A *reachability condition* is given by a set $\mathsf{C}_R \subseteq \mathsf{C}$ which induces the winning condition $\mathsf{W}_R = \{\mathsf{P} = \mathsf{c}_0, \mathsf{c}_1, \cdots \mid \exists i \in \mathbb{N} : \mathsf{c}_i \in \mathsf{C}_R\}$, i.e. the set of all plays that visit a configuration in C_R. Accordingly, a *safety condition* is given by a set $\mathsf{C}_S \subseteq \mathsf{C}$ which induces the winning condition $\mathsf{W}_S = \{\mathsf{P} = \mathsf{c}_0, \mathsf{c}_1, \cdots \mid \forall i \in \mathbb{N} : \mathsf{c}_i \notin \mathsf{C}_S\}$, i.e. the set of all plays that never visit a configuration in C_S. Reachability games and safety games are dual to each other in the sense that a reachability game with winning condition C_R can be seen as a safety game with winning condition $\mathsf{C}_S = \mathsf{C} \setminus \mathsf{C}_R$, where the roles of players A and B are swapped.

Strategies 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 $\sigma_A : \mathsf{C}_A \rightarrow \mathsf{C}_B$. For player B, strategies are defined accordingly.

Two strategies σ_A and σ_B together with an initial configuration c_0 induce a finite or infinite 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 $\mathsf{c}_i \in \mathsf{C}_B$. Given a winning condition W, a strategy σ_A is *winning* from a configuration c_0, if for *all* strategies σ_B it holds that player A wins the play $\mathsf{P}(\mathsf{c}_0, \sigma_A, \sigma_B)$. That is, for each σ_B, either $\mathsf{P}(\mathsf{c}_0, \sigma_A, \sigma_B) \in \mathsf{W}$ or the play is finite and ends in a deadlock of player B. A configuration c_0 is *winning* for player A if she has a strategy that is winning from c_0. Equivalent notions exist for player B. Given a reachability condition W_R / a safety condition W_S, the **reachability problem / safety problem** for a game $\mathcal{G}$ and a configuration c_0 is to decide whether c_0 is winning for player A.

Lemma 1 (Proposition 2.21 in [18])**.** *In reachability and 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 reachability and safety games in this paper, all strategies will be positional.

5 Reachability and Safety Games under TSO Semantics

We model the execution of a TSO program as a game between two players: The *process player A* takes the role of the program and decides at each execution step which instruction to execute. The *update player B* is in charge of the buffer message updates in between.

Formally, a TSO program $\mathcal{P} = \langle \mathsf{Proc}^\iota \rangle_{\iota \in \mathcal{I}}$ induces a game $\mathcal{G}(\mathcal{P}) = \langle \mathsf{C}, \mathsf{C}_A, \mathsf{C}_B, \rightarrow \rangle$ 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 transition relation $\rightarrow$ is defined by the following rules:

- **Program** 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 $\mathsf{c}_A \xrightarrow{\mathsf{instr}_\iota} \mathsf{c}'_B$. This means that the process player can execute any program instruction.
- **Update** For each transition $\mathsf{c}_B \in \mathsf{C}_B$, it holds that $\mathsf{c}_B \xrightarrow{\mathtt{up}^*} \mathsf{c}'_A$ for all c' with $\mathsf{c} \xrightarrow{\mathtt{up}^*}_{\mathcal{P}} \mathsf{c}'$. This means that the update player can update any amount of buffer messages (including zero) between each of the turns of the process player.

In the remainder of this work, we will consider reachability or safety winning conditions induced by a set of local states $\mathsf{Q}^{\mathcal{P}}_{\mathsf{W}} \subset \mathsf{Q}^{\mathcal{P}}$. The corresponding set of configurations is $\mathsf{C}_{\mathsf{W}} = \{\mathsf{c} = \langle \mathcal{S}, \mathcal{B}, \mathcal{M} \rangle_X \mid X \in \{A, B\} \wedge \exists \iota : \mathcal{S}(\iota) \in \mathsf{Q}^{\mathcal{P}}_{\mathsf{W}}\}$, that is, the set of all configurations where at least one process is in a state of $\mathsf{Q}^{\mathcal{P}}_{\mathsf{W}}$. The set C_{W} can then induce either a reachability or safety winning condition. In the following, we will assume that the initial configuration (usually named c_0) is not contained in C_{W}, since otherwise the game is decided immediately. Furthermore, we desire that the process player immediately wins when reaching a target state in a reachability game, that is, we do not care whether the play can be extended infinitely or not. Formally, we require that in a reachability game, a process cannot deadlock from a target state, implying that the process player cannot lose after reaching it.

Games on Single-Process Programs This section introduces games on single-process programs which will help us analysing the general case. Given a game induced by a concurrent program $\mathcal{P}$, we compare it to the game on just one of the processes of $\mathcal{P}$. We show that if the process player wins the single-process game, then she also wins the original game. The main idea is that she achieves this by executing exactly the same instructions in both games.

For the remainder of this section, fix a program $\mathcal{P} = \langle \mathsf{Proc}^\iota \rangle_{\iota \in \mathcal{I}}$ and a process index $\iota \in \mathcal{I}$. Let $\mathcal{P}^\iota = \langle \mathsf{Proc}^\iota \rangle$, i.e. the restriction of $\mathcal{P}$ to only the process Proc^ι. Define $\mathcal{G} = \mathcal{G}(\mathcal{P})$ and $\mathcal{G}^\iota = \mathcal{G}(\mathcal{P}^\iota)$, that is, the games induced by $\mathcal{P}$ and $\mathcal{P}^\iota$, respectively. Let Q_{W} induce a reachability or safety winning condition for $\mathcal{G}$ and define the winning condition for $\mathcal{G}^\iota$ through $\mathsf{Q}^\iota_{\mathsf{W}} = \mathsf{Q}_{\mathsf{W}} \cap \mathsf{Q}^\iota$.

Now, fix a configuration $\mathsf{c}_0 \in \mathsf{C} \setminus \mathsf{C}_{\mathsf{W}}$ with empty buffers (i.e. $\mathcal{B}(\mathsf{c}_0) = \langle \varepsilon \rangle_{\iota \in \mathcal{I}}$). For $X \in \{A, B\}$ and a configuration $\mathsf{c} = \langle \mathcal{S}, \mathcal{B}, \mathcal{M} \rangle_X \in \mathsf{C}_X$, let $\mathsf{c} \downarrow^\iota = \langle \mathcal{S}(\iota), \mathcal{B}(\iota), \mathcal{M} \rangle_X \in \mathsf{C}^\iota_X$, which can be understood as the projection of c onto the process Proc^ι. Conversely, for a configuration $\mathsf{c}^\iota \in \mathsf{C}^\iota_X$ of $\mathcal{G}^\iota$, define $\mathsf{c}^\iota \uparrow_{\mathcal{P}} = \langle \mathcal{S}(\mathsf{c}_0)[\iota \leftarrow \mathcal{S}(\mathsf{c}^\iota)], \mathcal{B}(\mathsf{c}_0)[\iota \leftarrow \mathcal{B}(\mathsf{c}^\iota)], \mathcal{M}(\mathsf{c}^\iota) \rangle_X \in \mathsf{C}_X$, that is, the configuration of $\mathcal{G}$ which is like c^ι for the process Proc^ι, but the local states and buffers of all other processes are as in the initial configuration c_0. Note that for all c^ι of $\mathcal{G}^\iota$, it holds that $(\mathsf{c}^\iota \uparrow_{\mathcal{P}}) \downarrow^\iota = \mathsf{c}^\iota$. On the other hand, $(\mathsf{c} \downarrow^\iota) \uparrow_{\mathcal{P}} = \mathsf{c}$ only holds for some c of $\mathcal{G}$.

Lemma 2. *If the process player wins $\mathcal{G}^\iota$ starting from the configuration $\mathsf{c}^\iota_0 = \mathsf{c}_0 \downarrow^\iota$, then she also wins $\mathcal{G}$ starting from c_0.*

Proof idea. Given a winning strategy σ^ι_A for the process player in $\mathcal{G}^\iota$, define a strategy in $\mathcal{G}$ by $\sigma_A(\mathsf{c}) = \sigma^\iota_A(\mathsf{c} \downarrow^\iota) \uparrow_{\mathcal{P}}$. We can show by induction over the number of moves that both strategies force the play to visit the same sequence of local states. Since winning the game is defined in terms of which local states are visited, it follows that σ_A must be a winning strategy. The full proof can be found in the extended version of this paper [23]. □

It is easy to see that the converse statement of Lemma 2 cannot hold for all $\iota \in \mathcal{I}$. Rather, we only show that under certain conditions the process player is able to visit the same local states of a process

Proc^1 in both $\mathcal{G}$ and $\mathcal{G}^\iota$. The strategy to do so will take a specific play in $\mathcal{G}$ and mimic all instructions that have been played in Proc^ι, similar as in the previous proof.

Fix $\iota \in \mathcal{I}$. Let σ_A be a winning strategy for the process player and let σ_B be the strategy of the update player where she never updates any buffer messages to the memory. Consider the play $\mathsf{P}(\mathsf{c}_0, \sigma_A, \sigma_B)$ in $\mathcal{G}$. For $k = 1, 2, \ldots$, let $\bar{\mathsf{c}}_k \to \mathsf{c}_k$ be the k-th time in this play where either the local state or the buffer of Proc^ι changes. This transition is due to some unique instruction $\mathcal{S}(\bar{\mathsf{c}}_k)(\iota) \xrightarrow{\mathsf{instr}_k} \mathcal{S}(\mathsf{c}_k)(\iota)$ in Proc^ι. In particular, it cannot be due to a memory update, since σ_B was chosen that way. Note that there does not necessarily need to be an infinite amount of k with this property. We define the strategy σ_A^ι as follows: Whenever $\mathcal{G}^\iota$ is in the k-th round, the local state of the process is $\mathcal{S}(\bar{\mathsf{c}}_k)(\iota)$ and instr_k is enabled, execute this instruction to move to the unique configuration with local state $\mathcal{S}(\mathsf{c}_k)(\iota)$. Otherwise, make an arbitrary move. Let σ_B^ι be an arbitrary strategy of the update player for $\mathcal{G}^\iota$. After the k-th round of $\mathsf{P}(\mathsf{c}_0^\iota, \sigma_A^\iota, \sigma_B^\iota)$, the game is in some position c_k^ι.

Claim 3. *For all $k \in \mathbb{N}$ for which c_k is defined, it holds that $\mathsf{c}_k \downarrow^\iota \xrightarrow{\mathtt{up}^*} \mathsf{c}_k^\iota$.*

Proof idea. First, we show by induction over k that instr_k is enabled at c_{k-1}^ι. Let $\tilde{\mathsf{c}}_k^\iota$ be such that $\mathsf{c}_{k-1}^\iota \xrightarrow{\mathsf{instr}_k} \tilde{\mathsf{c}}_k^\iota$. We compare the buffer and memory of c_k and $\tilde{\mathsf{c}}_k^\iota$ to conclude $\mathsf{c}_k \downarrow^\iota \xrightarrow{\mathtt{up}^*} \tilde{\mathsf{c}}_k^\iota$. The claim follows from $\tilde{\mathsf{c}}_k^\iota \xrightarrow{\mathtt{up}^*} \mathsf{c}_k^\iota$. The full proof is again in the extended version [23]. □

Concurrent Games We combine the results for single-process games to obtain the following theorem.

Theorem 4. *The process player wins $\mathcal{G}$ starting from a configuration c_0 if and only if she also wins $\mathcal{G}^\iota$ starting from configuration $\mathsf{c}_0 \downarrow^\iota$ for at least one $\iota \in \mathcal{I}$.*

Proof. By Lemma 2, if the process player wins $\mathcal{G}^\iota$ for at least one $\iota \in \mathcal{I}$, then she also wins $\mathcal{G}$. For the other direction, consider the strategies as defined above. What is left to show is that σ_A^ι is winning for at least one $\iota \in \mathcal{I}$.

If the process player has a reachability objective, $\mathsf{P}(\mathsf{c}_0, \sigma_A, \sigma_B)$ visits at least one target state in some process Proc^ι. Note again that the update player cannot be deadlocked and therefore the play must be infinite. From Claim 3 it follows that $\mathsf{P}(\mathsf{c}_0^\iota, \sigma_A^\iota, \sigma_B^\iota)$ visits the same local states of Proc^ι than $\mathsf{P}(\mathsf{c}_0, \sigma_A, \sigma_B)$ and in particular, it visits the same target state. Since σ_B^ι was chosen arbitrarily, it means that σ_A^ι is a winning strategy. Otherwise, if the process player has a safety objective, $\mathsf{P}(\mathsf{c}_0, \sigma_A, \sigma_B)$ executes an infinite amount of instructions in at least one process Proc^ι, but never visits any of its target states. Using the same arguments as previously, it follows that $\mathsf{P}(\mathsf{c}_0^\iota, \sigma_A^\iota, \sigma_B^\iota)$ is also a winning play and σ_A^ι is a winning strategy. □

Theorem 4 reduces the reachability problem for games on concurrent programs to the single-process case. Although the game $\mathcal{G}^\iota$ still has an infinite amount of configurations, many of them are indistinguishable in the sense that they have the same local state and allow the same sequences of instructions to be executed. Section 6 formally constructs a finite game that is a so-called *bisimulation* of $\mathcal{G}^\iota$. This shows that the reachability and safety problems for TSO games are decidable. In fact, they are PSPACE-complete.

Fairness Conditions In the previous section we have seen that the game played under TSO semantics reduces to the analysis of games on single-process programs. This is somewhat unsatisfying, since those games do not exhibit any concurrent behaviour that arises from the communication between multiple

processes. The underlying reason is the structure of the optimal strategies of the two players: If the process player wins, it is because she only plays in one single process. Otherwise, the update player wins by never updating any buffer messages. Both of these behaviours are not natural in the sense that they will not occur in any reasonable program environment: We should be able to assume that eventually (1) every buffer message will be updated to the memory and (2) every process will execute an instruction. In Section 7 and Section 8 we will impose additional restrictions on the two players to enforce this behaviour.

6 Decidability of Single-Process Games

TSO Views In the single-process program $\mathcal{P}^\iota$, there is no communication between different processes. A read operation of the process Proc^ι on a variable x either reads the initial value from the (shared) memory, or the value of the last write on x done by Proc^ι, if such a write operation has happened. In the latter case, the value of the read operation can come from either the buffer of Proc^ι or directly from the memory. But a single process cannot distinguish between these two cases. To be exact, the information that the process can obtain from the buffer and the memory is the value that Proc^ι can read from each variable, and whether the process can execute a memory fence instruction or not. Together with the local state of Proc^ι at the current configuration, this completely determines the enabled transitions for the process.

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

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

Given a view $\mathsf{v} = \langle \mathcal{S}, \mathcal{V}, \mathcal{F} \rangle$, we write $\mathcal{S}(\mathsf{v})$, $\mathcal{V}(\mathsf{v})$ and $\mathcal{F}(\mathsf{v})$ for the local state $\mathcal{S}$, the value state $\mathcal{V}$ and the fence state $\mathcal{F}$ of v, respectively. The view of a configuration $\mathsf{c} \in \mathsf{C}^\iota$ is denoted by $\mathsf{v}(\mathsf{c})$ and defined in the following way. First, $\mathcal{S}(\mathsf{v}(\mathsf{c})) = \mathcal{S}(\mathsf{c})$. For all $\mathsf{x} \in \mathsf{Vars}$, if $\mathcal{B}(\mathsf{c})|_{\{\mathsf{x}\} \times \mathsf{Dom}} = \langle \mathsf{x}, \mathsf{d} \rangle \cdot \mathsf{w}$, then $\mathcal{V}(\mathsf{v}(\mathsf{c}))(\mathsf{x}) = \mathsf{d}$. Otherwise, $\mathcal{V}(\mathsf{v}(\mathsf{c}))(\mathsf{x}) = \mathcal{M}(\mathsf{c})(\mathsf{x})$. Lastly, $\mathcal{F}(\mathsf{v}(\mathsf{c})) = \mathsf{true}$ if and only if $\mathcal{B}(\mathsf{c}) = \varepsilon$.

For $\mathsf{c}_1, \mathsf{c}_2 \in \mathsf{C}$, if $\mathsf{v}(\mathsf{c}_1) = \mathsf{v}(\mathsf{c}_2)$, then we write $\mathsf{c}_1 \equiv \mathsf{c}_2$. In such a case, the process Proc^ι cannot differentiate between c_1 and c_2 in the sense that the enabled transitions in both configurations are the same. This is shown in Figure 2 and formally captured in Lemma 5.

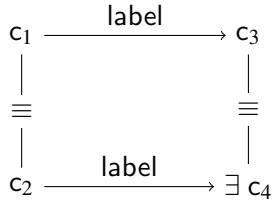


Figure 2: The configurations of Lemma 5.

Lemma 5. *For all* $\mathsf{c}_1, \mathsf{c}_3, \mathsf{c}_2 \in \mathsf{C}^\iota$ *and* $\mathsf{label} \in \mathsf{Instrs} \cup \{\mathtt{up}^*\}$ *with* $\mathsf{c}_1 \equiv \mathsf{c}_2$ *and* $\mathsf{c}_1 \xrightarrow{\mathsf{label}} \mathsf{c}_3$*, there exists a* $\mathsf{c}_4 \in \mathsf{C}^\iota$ *such that* $\mathsf{c}_3 \equiv \mathsf{c}_4$ *and* $\mathsf{c}_2 \xrightarrow{\mathsf{label}} \mathsf{c}_4$.

Proof. If $\mathsf{label} = \mathtt{up}^*$, this clearly holds for $\mathsf{c}_4 = \mathsf{c}_2$. Otherwise, we first show that $\mathsf{label} \in \mathsf{Instrs}$ is enabled at c_2. Since $\mathsf{c}_1 \equiv \mathsf{c}_2$, it holds that $\mathcal{S}(\mathsf{c}_1) = \mathcal{S}(\mathsf{c}_2)$. Furthermore, if $\mathsf{label} = \mathtt{rd}(\mathsf{x},\mathsf{d})$, then $\mathcal{V}(\mathsf{v}(\mathsf{c}_1))(\mathsf{x}) = \mathcal{V}(\mathsf{v}(\mathsf{c}_2))(\mathsf{x}) = \mathsf{d}$. Also, if $\mathsf{label} = \mathtt{mf}$, then $\mathcal{B}(\mathsf{v}(\mathsf{c}_1)) = \varepsilon$ and since $\mathcal{F}(\mathsf{v}(\mathsf{c}_1)) = \mathcal{F}(\mathsf{v}(\mathsf{c}_2)) = \mathsf{true}$ it follows that $\mathcal{B}(\mathsf{v}(\mathsf{c}_2)) = \varepsilon$. From these considerations and the definition of the TSO semantics (see Figure 1), it follows that label is indeed enabled at c_2.

Let c_4 be the unique configuration obtained after executing the transition $\mathcal{S}(\mathsf{c}_1) \xrightarrow{\mathsf{label}} \mathcal{S}(\mathsf{c}_3)$ at c_2, i.e. $\mathsf{c}_2 \xrightarrow{\mathsf{label}} \mathsf{c}_4$ and $\mathcal{S}(\mathsf{c}_4) = \mathcal{S}(\mathsf{c}_3)$. If $\mathsf{label} = \mathtt{wr}(\mathsf{x},\mathsf{d})$, then $\mathcal{V}(\mathsf{v}(\mathsf{c}_4)) = \mathcal{V}(\mathsf{v}(\mathsf{c}_3)) = \mathcal{V}(\mathsf{v}(\mathsf{c}_1))[\mathsf{x} \leftarrow \mathsf{d}]$ and $\mathcal{F}(\mathsf{v}(\mathsf{c}_4)) = \mathcal{F}(\mathsf{v}(\mathsf{c}_3)) = \mathsf{false}$. Otherwise, $\mathcal{V}(\mathsf{v}(\mathsf{c}_4)) = \mathcal{V}(\mathsf{v}(\mathsf{c}_3)) = \mathcal{V}(\mathsf{v}(\mathsf{c}_1))$ and $\mathcal{F}(\mathsf{v}(\mathsf{c}_4)) = \mathcal{F}(\mathsf{v}(\mathsf{c}_3)) = \mathcal{F}(\mathsf{v}(\mathsf{c}_1))$. In all cases it follows that $\mathsf{c}_3 \equiv \mathsf{c}_4$. □

Bisimulations A *colouring* of a game $\mathcal{G}$ is a function $\lambda : \mathsf{C} \to \mathcal{C}$ from the set of configurations into some set of colours $\mathcal{C}$. Consider two games $\mathcal{G}$ and $\mathcal{G}'$ with colouring functions λ and λ', respectively. A *bisimulation* (also called *zig-zag relation*) between $\mathcal{G}$ and $\mathcal{G}'$ is a relation $\mathsf{Z} \subseteq \mathsf{C} \times \mathsf{C}'$ such that for all pair of related configurations $(\mathsf{c}_1,\mathsf{c}_2) \in \mathsf{Z}$ it holds that:

- c_1 and c_2 agree on their colour: $\lambda(\mathsf{c}_1) = \lambda'(\mathsf{c}_2)$
- (*zig*) for each transition $\mathsf{c}_1 \xrightarrow{\mathsf{label}} \mathsf{c}_3$ there is a transition $\mathsf{c}_2 \xrightarrow{\mathsf{label}} \mathsf{c}_4$ such that $(\mathsf{c}_3,\mathsf{c}_4) \in \mathsf{Z}$.
- (*zag*) for each transition $\mathsf{c}_2 \xrightarrow{\mathsf{label}} \mathsf{c}_4$ there is a transition $\mathsf{c}_1 \xrightarrow{\mathsf{label}} \mathsf{c}_3$ such that $(\mathsf{c}_3,\mathsf{c}_4) \in \mathsf{Z}$.

We say that two related configurations c_1 and c_2 are *bisimilar* and write $\mathsf{c} \approx \mathsf{c}'$. We call $\mathcal{G}$ and $\mathcal{G}'$ *bisimilar* if there is a bisimulation between them.

It is common knowledge in game theory that winning strategies are preserved under bisimulations if the colourings are a refinement of the winning condition in the following sense. Consider two bisimilar games $\mathcal{G}$ and $\mathcal{G}'$ with winning conditions given by C_W and C'_W, respectively. Let λ be a colouring function for $\mathcal{G}$ such that the configurations in C_W have different colours than the rest of the configurations, i.e. $\lambda^{-1}(\lambda(\mathsf{C}_\mathsf{W})) = \mathsf{C}_\mathsf{W}$. Define λ' as a colouring function for $\mathcal{G}'$ accordingly.

Lemma 6. *Given two bisimilar configurations $\mathsf{c}_0 \in \mathsf{C}$ and $\mathsf{c}'_0 \in \mathsf{C}'$, it holds that c_0 is a winning configuration in $\mathcal{G}$ if and only if c'_0 is a winning configuration in $\mathcal{G}'$.*

We define a game on views $\mathcal{G}^\mathsf{V} = \langle \mathsf{V}, \mathsf{V}_A, \mathsf{V}_B, \to_\mathsf{v} \rangle$ that is bisimilar to the single-process game $\mathcal{G}^\iota$. Let $\mathsf{V}_X = \{\mathsf{v}(\mathsf{c})_X \mid \mathsf{c} \in \mathsf{C}^\iota\}$ for $X \in \{A,B\}$ and $\mathsf{V} = \mathsf{V}_A \cup \mathsf{V}_B$. We extend the notation of the function v to game configurations by $\mathsf{v}(\mathsf{c}_X) = \mathsf{v}(\mathsf{c})_X$ for $X \in \{A,B\}$ and $\mathsf{c} \in \mathsf{C}^\iota$. Now, we can define $\to_\mathsf{v}$ by $\mathsf{v}(\mathsf{c}) \xrightarrow{\mathsf{label}}_\mathsf{v} \mathsf{v}(\mathsf{c}')$ if and only if $\mathsf{c} \xrightarrow{\mathsf{label}} \mathsf{c}'$ for some $\mathsf{c},\mathsf{c}' \in \mathsf{C}^\iota$.

Theorem 7. *The relation $\mathsf{Z} = \{(\mathsf{c},\mathsf{v}(\mathsf{c})) \mid \mathsf{c} \in \mathsf{C}^\iota\} \subset \mathsf{C}^\iota \times \mathsf{C}^\mathsf{V}$ is a bisimulation between $\mathcal{G}^\iota$ and $\mathcal{G}^\mathsf{V}$ with colouring functions $\lambda^\iota : \mathsf{C}^\iota \to \mathsf{V}, \mathsf{c} \mapsto \mathsf{v}(\mathsf{c})$ and $\lambda^\mathsf{V} = \mathrm{id}_\mathsf{V}$, respectively.*

Proof. From the definition it follows directly that related configurations agree on their colour and that $\mathcal{G}^\mathsf{V}$ can simulate $\mathcal{G}^\iota$. What is left to show is that $\mathcal{G}^\iota$ can also simulate $\mathcal{G}^\mathsf{V}$, i.e. that for all $\mathsf{c} \approx \mathsf{v}$ and $\mathsf{v} \xrightarrow{\mathsf{label}}_\mathsf{v} \tilde{\mathsf{v}}$ there is $\mathsf{c} \xrightarrow{\mathsf{label}} \tilde{\mathsf{c}}$ with $\tilde{\mathsf{c}} \approx \tilde{\mathsf{v}}$. The transition $\mathsf{v} \xrightarrow{\mathsf{label}}_\mathsf{v} \tilde{\mathsf{v}}$ is due to a transition $\mathsf{d} \xrightarrow{\mathsf{label}} \tilde{\mathsf{d}}$ for some $\mathsf{d}, \tilde{\mathsf{d}} \in \mathsf{C}^\iota$ with $\mathsf{d} \approx \mathsf{v}$ and $\tilde{\mathsf{d}} \approx \tilde{\mathsf{v}}$. Since $\mathsf{v}(\mathsf{c}) = \mathsf{v} = \mathsf{v}(\mathsf{d})$, it follows that $\mathsf{c} \equiv \mathsf{d}$. Apply Lemma 5 to $\mathsf{c}_1 = \mathsf{d}$, $\mathsf{c}_2 = \mathsf{c}$ and $\mathsf{c}_3 = \tilde{\mathsf{d}}$ to obtain a configuration $\tilde{\mathsf{c}} = \mathsf{c}_4$ with the desired properties. □

Since C^V is finite, it is rather evident that the reachability and safety problem are decidable, e.g. by applying a backward induction algorithm. In fact, both problems are PSPACE-complete. Intuitively, this makes sense since each variable can be seen as a single cell of a bounded Turing machine. The extended

version of this paper [23] gives a polynomial-space algorithm to show the upper complexity bound and constructs a reduction from TQBF for the lower bound. These results then immediately translate to the single-process TSO game $\mathcal{G}^{\iota}$, since it is bisimilar to $\mathcal{G}^{\mathsf{V}}$.

7 Update Fairness in Reachability Games

In this section, we introduce *update fairness*, which we require the update player to satisfy. The core idea of update fairness is that eventually, each buffer message will be updated to the shared memory. This means that the process player can in some sense *wait* for the buffer messages to arrive in the memory. In safety games, delaying the run indefinitely favours the process player. Thus, we will focus on reachability games in this section.

We implement update fairness as follows. Whenever the program is in a configuration at which no program instruction is enabled (a deadlock), the system waits for the update player to update buffer messages to the memory, until the program exits the deadlock (or all buffers are empty). To simplify the formalisation, we will assume that the update player does so in her very next turn. This idea can be equivalently expressed by saying that if it is the process player's turn and the system is deadlocked, then it follows that all buffers must be empty.

Let $\mathcal{P} = \langle \mathsf{Proc}^{\iota} \rangle_{\iota \in \mathcal{I}}$ be a TSO program with induced game $\mathcal{G}(\mathcal{P})$. We define W_U as the set of all plays $\mathsf{P} = \mathsf{c}_0, \mathsf{c}_1, \ldots$ in $\mathcal{G}(\mathcal{P})$ that satisfy update fairness:

$$\mathsf{P} \in \mathsf{W}_U \iff \forall\, k \in \mathbb{N}, \mathsf{c}_k \in \mathsf{C}_A : (\mathrm{Post}(\mathsf{c}_k) = \emptyset \implies \forall\, \iota \in \mathcal{I} : \mathcal{B}(\mathsf{c}_k)(\iota) = \varepsilon)$$

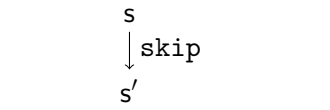
For a reachability condition W_R, let the set $\mathsf{W}_{RU} = \mathsf{W}_R \cup \overline{\mathsf{W}_U} = \{\mathsf{P} \mid \mathsf{P} \in \mathsf{W}_R \vee \mathsf{P} \notin \mathsf{W}_U\}$ be the set of winning plays for the process player, i.e. the set of all plays that either reach a target state or that do not admit update fairness. The remainder of this section will be dedicated to show that the reachability problem under update fairness is undecidable. We will achieve this by reducing the state reachability problem of perfect channel systems, which is undecidable, to the reachability problem of $\mathcal{G}(\mathcal{P})$ with respect to W_{RU}. The main ideas of the reduction are similar to those in [24].

Given a perfect channel system $\mathcal{L} = \langle \mathsf{S}, \mathsf{M}, \delta \rangle$, we construct a TSO program $\mathcal{P}$ that simulates $\mathcal{L}$. The process player will decide which transitions of the PCS to simulate, while the update player only takes care of the buffer updates. The program consists of two processes Proc^1 and Proc^2, which are shown in Figure 3 and Figure 4, respectively.

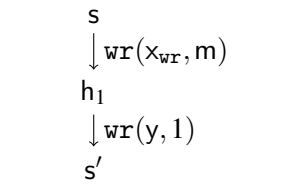
The first process keeps track of the configuration of the channel system and simulates the control flow. For each transition in $\mathcal{L}$, we construct a sequence of transitions in Proc^1 that simulates both the state change and the channel behaviour of the $\mathcal{L}$-transition. To achieve this, 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 $\langle \mathsf{x}_{\mathtt{wr}}, \mathsf{m} \rangle$ 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 the second process Proc^2 to read the message from memory and copies it to the variable $\mathsf{x}_{\mathtt{rd}}$, where it can be read by Proc^1. We call the combination of reading a message m from $\mathsf{x}_{\mathtt{wr}}$ and writing it to $\mathsf{x}_{\mathtt{rd}}$ 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.

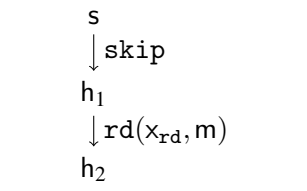
First, the update player might choose to lose a channel message by updating more than one message during a rotation. Consider an execution of $\mathcal{P}$ that simulates two send operations $!\mathsf{m}_1$ and $!\mathsf{m}_2$, i.e.



(a) skip operation $s \xrightarrow{\texttt{skip}}_{\mathcal{L}} s'$



(b) send operation $s \xrightarrow{!m}_{\mathcal{L}} s'$



(c) receive operation $s \xrightarrow{?m}_{\mathcal{L}} s'$

Figure 3: Proc^1 of the reduction from PCS.

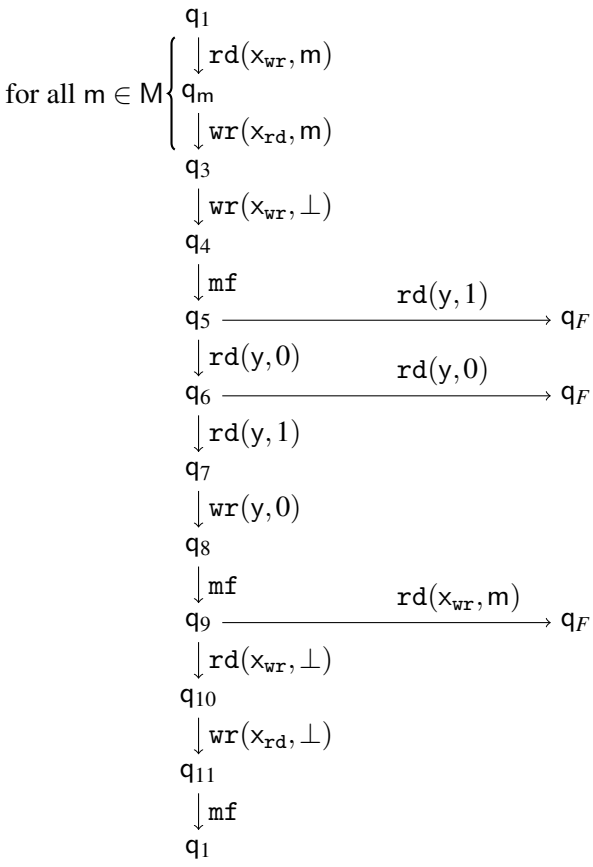


Figure 4: Proc^2 of the reduction from PCS.

Proc^1 adds $\langle x_{wr}, m_1 \rangle$ and $\langle x_{wr}, m_2 \rangle$ to its buffer. Now, if the process player wants to simulate a receive operation and initiates a message rotation, the update player can update both messages $\langle x_{wr}, m_1 \rangle$ and $\langle x_{wr}, m_2 \rangle$ to the memory before Proc^2 reads from x_{wr}. Thus, the first message m_1 is overwritten by the second message m_2 and is lost without ever being received. To prevent this, we implement a protocol that ensures that in each message rotation, exactly one channel message is being updated.

We extend the construction of Proc^1 such that it inserts an auxiliary message $\langle y, 1 \rangle$ into its buffer after the simulation of each send operation. After a message rotation, that is, after Proc^2 copied a message from x_{wr} to x_{rd}, the process then resets the value of x_{wr} to its initial value $\bot$. Next, the process checks that y contains the value 0, which indicates that only one message was updated to the memory. Now, the update player is allowed to update exactly one $\langle y, 1 \rangle$ buffer message, after which Proc^2 resets y to 0. To ensure that the update player has actually updated only one message in this step, Proc^2 then checks that x_{wr} is still empty. If this protocol is violated at any point, Proc^2 enables the process player to immediately move to a winning state.

Although we have established that during each message rotation exactly one channel message will be rotated, we also need to ensure that for each rotation, Proc^1 will simulate exactly one receive operation. This is achieved by another protocol between Proc^1 and Proc^2, which gives the update player the tools to

enforce correct behaviour. To begin, the process player needs to initiate the simulation of a receive operation by moving to the first auxiliary state h_1 shown in Figure 3c. Only then is the program in a deadlock and the update player is forced to perform a message update. When reaching the first memory fence in Proc^2, the system is deadlocked again. Of course, the update player will not update the next message in the buffer of Proc^1, since it will lead to the process player immediately winning later on. Thus, she updates the message $\langle \mathsf{x}_{\mathtt{rd}}, \mathsf{m} \rangle$ to the memory, which enables both processes to continue. The next time that the update player is forced to update is when Proc^1 reaches the second auxiliary state h_2 and Proc^2 reaches the second memory fence. Only emptying the buffer of Proc^2 allows the program to continue. After three more instructions, Proc^2 will reach the third memory fence. Again, the update player needs to empty the buffer of Proc^2 which updates $\langle \mathsf{x}_{\mathtt{rd}}, \bot \rangle$ and enables Proc^1 to finish the simulation of the receive operation.

This concludes the mechanisms implemented to ensure that each channel message is received *once and only once*. We have constructed a TSO game with update fairness that simulates a perfect channel system. The winning condition of the game will be the reachability condition induced by the final states of the PCS together with the update fairness condition. We summarise our results in the following theorem.

Theorem 8. *The reachability problem for TSO games with update fairness is undecidable.*

8 Process Fairness in Safety Games

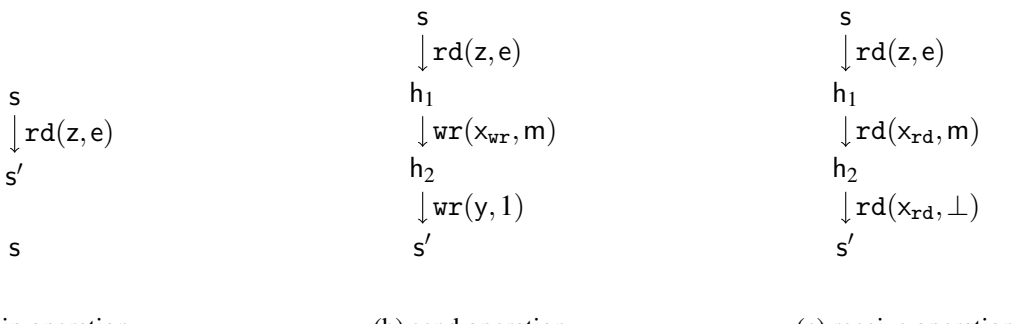
In the previous section, we have limited the behaviour of the update player. Now, we introduce *process fairness*, which will impact the capabilities of the process player. Process fairness means that for each process that is enabled infinitely many times during a run, the process player executes an instruction in that process infinitely often. In reachability games, this is no real restriction to the process player: If she can reach the set of winning states, then she can do so in finitely many moves. Thus, any finite prefix of a play that reaches a winning state can then trivially be extended to an infinite play that admits process fairness. Because of this, we will target our attention only towards safety games, where the process player cannot win in a finite amount of moves.

We formalise process fairness as follows. Let $\mathcal{P} = \langle \mathsf{Proc}^\iota \rangle_{\iota \in \mathcal{I}}$ be a TSO program with induced game $\mathcal{G}(\mathcal{P})$. We define W_P as the set of all plays $\mathsf{P} = \mathsf{c}_0, \mathsf{c}_1, \ldots$ in $\mathcal{G}(\mathcal{P})$ that satisfy process fairness:

$$\mathsf{P} \in \mathsf{W}_P \iff \forall\, \iota \in \mathcal{I} : \left(\exists^\infty\, k \in \mathbb{N}, \mathsf{c}_k \in \mathsf{C}_A : \mathsf{c}_k \xrightarrow{\mathsf{instr}_\iota} \mathsf{c}' \implies \exists^\infty\, k' \in \mathbb{N}, \mathsf{c}_{k'} \in \mathsf{C}_A : \mathsf{c}_{k'} \xrightarrow{\mathsf{instr}'_\iota} \mathsf{c}_{k'+1} \right)$$

Given a safety condition W_S, the intersection $\mathsf{W}_{SP} = \mathsf{W}_S \cap \mathsf{W}_P$ defines the set of winning plays that admit process fairness. In the remainder of this section we will show that the safety problem under process fairness is undecidable. To do so, we use a construction very similar to the one from the previous section to reduce the state reachability problem of perfect channel systems, to the safety problem of $\mathcal{G}(\mathcal{P})$. Before, it was the process player who decided which transition of the channel system to simulate. This time, it will be the update player who has this task.

Consider again a perfect channel system $\mathcal{L} = \langle \mathsf{S}, \mathsf{M}, \delta \rangle$. We modify the construction from the previous section. First, we introduce another shared variable z. Then, for each transition $\mathsf{e} \in \delta$ of the perfect channel system, we add an auxiliary process $\mathsf{Proc}^{\mathsf{e}}$. It consists of exactly one state q_{e} and one looping transition $\mathsf{q}_{\mathsf{e}} \xrightarrow{\mathtt{wr}(\mathsf{z},\mathsf{e})} \mathsf{q}_{\mathsf{e}}$. Furthermore, we prepend the gadget of process Proc^1 that simulates e with a transition $\mathtt{rd}(\mathsf{z},\mathsf{e})$. The result of this is shown in Figure 5. Process Proc^2 is taken from the previous construction without any changes and can be found in Figure 4.



(a) skip operation $e = s \xrightarrow{\texttt{skip}}_{\mathcal{L}} s' \in \delta$

(b) send operation $e = s \xrightarrow{!m}_{\mathcal{L}} s' \in \delta$

(c) receive operation $e = s \xrightarrow{?m}_{\mathcal{L}} s' \in \delta$

Figure 5: Proc^1 of the reduction from PCS to a TSO game with process fairness.

The main idea of these modifications is that the update player can use the variable z to control which channel operation will be simulated. At the start of the run, both z and x_{wr} contain the initial value $\perp$, which means that neither Proc^1 nor Proc^2 are enabled. Thus, the process player needs to begin playing in some process Proc^e, writing the message $\langle z, e\rangle$ to its buffer. This will continue until the update player decides to update one of these messages. But, due to process fairness, the process player is forced to eventually play in *all* enabled processes. In particular, she has to do so infinitely many times during any infinite play. This means that the update player can simply wait until each transition $e \in \delta$ was sufficiently many times added to the buffer of Proc^e to simulate a run of the PCS that reaches a final state. At that point, the update player starts updating the messages $\langle z, e\rangle$ one by one, each time waiting until the process player has finished simulating the unique operation that is enabled in Proc^1.

In more detail, to simulate the execution of a channel operation $e \in \delta$, the update player updates the buffer message $\langle z, e\rangle$ to the memory. Due to process fairness, we know that the process player eventually has to play in Proc^1, since it is now enabled. She takes the transition $\texttt{rd}(z, e)$ and then proceeds (although not necessarily immediately) with simulating e as was presented for the reachability case in the previous section. If e is a receive operation, the update player has to update a $\langle x_{wr}, m\rangle$ buffer message at some point to enable Proc^2 and start a message rotation. Again, process fairness forces the process player to eventually finish the rotation protocol. In any case, the simulation of the channel operation is guaranteed to terminate after finitely many steps. Now, the update player starts the next simulation by updating the corresponding buffer message.

Also in this construction we need to ensure that we do not introduce any behaviour that does not correspond exactly to what the PCS can do. Message loss due to updating two messages without rotation in between is handled in the same way as previously, using the auxiliary variable y. The same goes for message duplication, which is covered by the protocol between Proc^1 and Proc^2. What is left is message loss due to performing two rotations without simulating a receive operation. In the previous section, this could only happen if the process player decides to do so, since she was the one controlling the simulation. This is still prevented by the aforementioned protocol: The update player is not forced to let Proc^2 proceed beyond the second and third memory fences before Proc^1 keeps up with the protocol. But in this construction, the roles and capabilities of the two players have changed slightly and allow for additional behaviour: Without enabling a receive operation in Proc^1, the update player could update a message $\langle x_{wr}, m\rangle$, which enables Proc^2 instead. Due to process fairness, the process player would eventually have to perform a full message rotation.

We prevent this by adding for every channel message m a transition $\mathsf{q} \xrightarrow{\mathtt{rd}(\mathsf{x}_{\mathtt{rd}},\mathsf{m})} \mathsf{q}_F$ to *every* state q of Proc^1 *except* the states h_1 and h_2 (cf. Figure 5c) of the receive operations of message m. Here, q_F is a sink state which is safe for the process player and blocks the update player from winning the game. The idea of this transition is that during a message rotation, the value of $\mathsf{x}_{\mathtt{rd}}$ in the memory is m. Thus, the update player is not immediately losing only if Proc^1 is currently in one of the intermediate states of a receive operation. In particular, the process player can now move to h_2. Then, after the rotation has finished with the third memory fence, the variable $\mathsf{x}_{\mathtt{rd}}$ contains the value $\perp$ again, which means that Proc^1 is enabled and the process player can finish the simulation of the receive operation. We conclude that she can prevent the update player from performing a rotation without simulating a receive operation.

In summary, we have shown again that each channel message is read once and only once. The winning condition of the TSO game is given by the safety condition induced by the final states of the PCS together with the process fairness condition. This gives rise to the following theorem.

Theorem 9. *The safety problem for TSO games with process fairness is undecidable.*

9 Load Buffer Semantics in TSO Games

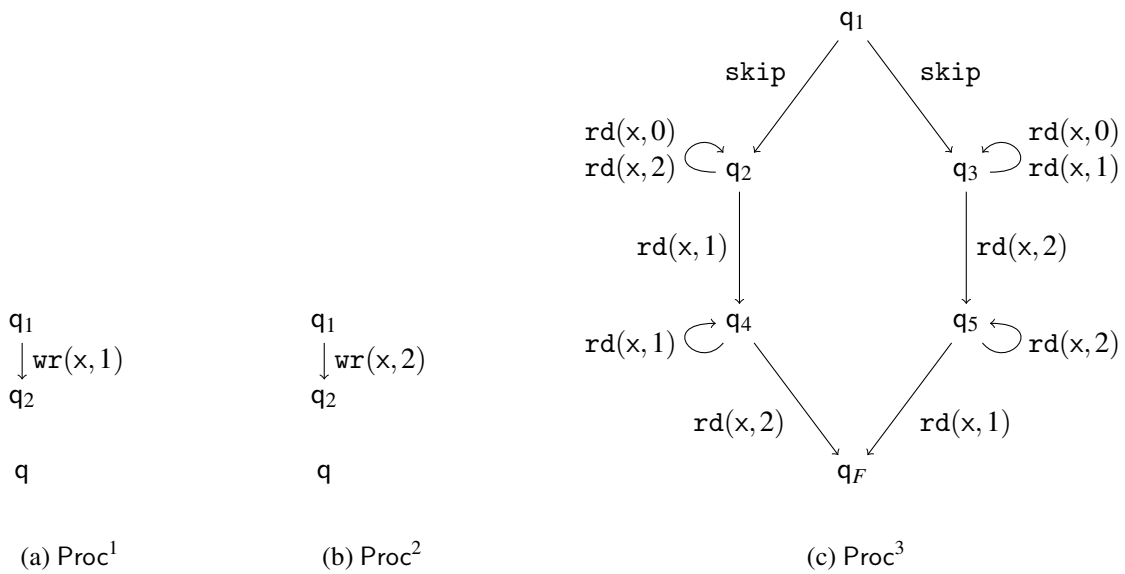
In [1], the authors introduced an alternative semantics for TSO, called *load buffer semantics*. It is equivalent to the traditional *store buffer semantics* in the sense that a global state $\mathcal{S}$ of the system is reachable under load-buffer semantics if and only if it is reachable under store buffer semantics. The alternative semantics have been proven to be useful in efficiently performing algorithmic verification or presenting simpler decidability proofs of safety properties. A natural question in the context of this paper is to ask whether these results transfer to the game setting. In particular, we want to know if a game is won by the same player when played under both semantics. Unfortunately, it turns out that this is not the case.

Load Buffer Semantics Under the new semantics, the store buffer between each process and the shared memory is replaced by a load buffer instead. This means that the information flow reverses its direction: Instead of write operations, the buffer now contains potential *read* operations that *might* be performed by the process. Each buffer message is either a pair $\langle \mathsf{x}, \mathsf{d} \rangle$ or a triple $\langle \mathsf{x}, \mathsf{d}, \mathsf{own} \rangle$, where the latter is called an *own-message*.

At any point during the run, the system can nondeterministically choose a variable x and its corresponding value d from the memory and add a message $\langle \mathsf{x}, \mathsf{d} \rangle$ to the tail of the buffer of one of the processes. This is called *read propagation* and speculates on a future read operation on x. Conversely, a *delete* operation removes the oldest message at the head of the buffer of some process and is also performed nondeterministically at any time.

A *write* instruction $\mathtt{wr}(\mathsf{x}, \mathsf{d})$ of a process Proc immediately updates the value d of the variable x in the memory. Then, it adds the own-message $\langle \mathsf{x}, \mathsf{d}, \mathsf{own} \rangle$ to the buffer of Proc. The behaviour of a *read* instruction $\mathtt{rd}(\mathsf{x}, \mathsf{d})$ depends on the contents of the buffer. If there is an own-message on the variable x, then the most recent one must correspond to the value d. Otherwise, if there is no such message, the head of the buffer must be a message $\langle \mathsf{x}, \mathsf{d} \rangle$. If this is not the case, the read instruction is disabled. The last two instructions, which are *skip* and *memory fence*, work exactly as in the classical TSO semantics: They only change the local state but not the memory or buffer, and the fence is only enabled if its buffer is empty.

For a formal definition of the semantics and the configurations of the induced transition system, we refer to [1].



(a) Proc^1 (b) Proc^2 (c) Proc^3

Figure 6: A concurrent program consisting of three processes.

Games Comparing the alternative to the classical TSO semantics, we see that the order in which the variables in the memory are updated, now depends directly on the order of execution of the corresponding write instructions, and not on the update order of the buffer messages. Conversely, the buffer does not delay the time when a write operation arrives at the memory, but instead delays when the change in the memory is visible to each process. Intuitively, we can already guess that the two semantics differ in the game setting, since the information available to the two players during the execution is different in both cases.

In the plain TSO game without fairness, there is actually no change. Since both reachability and safety games degenerate to single-process games with no communication between processes, the exact update semantics do not matter.

This is not the case if we add fairness conditions. Consider the safety game with process fairness played on the program shown in Figure 6. The target state is q_F in Proc^3 and the initial value of x is 0. Since the process player cannot be deadlocked in any other state of Proc^3, the only way for the update player to win is to force the play into q_F. In our classical game setting under store buffer semantics, the update player is able to achieve this. We describe a winning strategy for her.

Due to process fairness, the process player needs to eventually play in all three processes. The update player waits until processes Proc^1 and Proc^2 are in their respective q_2, and Proc_3 is in either q_2 or q_3. In the first case, if Proc_3 is in state q_2, she updates the buffer message of Proc^1, which writes the value 1 to the memory for variable x. The only enabled instruction for the process player is to move from q_2 to q_4 in Proc^3. Now, the update player updates the message from the buffer of Proc^2, which forces the process player to move to the target state and lose. In the other case, the update player performs the two update operations in the reverse order, which again forces the process player to enter the target state after two moves.

Next, consider the same game but under load buffer semantics. We have not yet formally defined how they should work, but this is not necessary for our argument. Assume that the process player as usual controls the program instructions and the update player in some way controls the nondeterministic

buffer behaviour. We will outline how the process player wins this game.

First, she plays in Proc^1, then in Proc^2. At this point, the value of x in the memory is 2, but the buffer of Proc^3 might already contain messages of the form $\langle \mathsf{x}, 1\rangle$ and $\langle \mathsf{x}, 2\rangle$. Note that it is only possible to have them in this exact order, i.e. it cannot be that there is some message $\langle \mathsf{x}, 2\rangle$ that is older than another message $\langle \mathsf{x}, 1\rangle$. Furthermore, since the program has no other reachable write instructions, any message that will be added in the future must be $\langle \mathsf{x}, 2\rangle$. Now, the process player plays in Proc^3 and moves to q_3. The update player needs the process player to eventually move to q_5, which means she has to enable the instruction $\mathtt{rd}(\mathsf{x}, 2)$. To do so, she deletes messages at the head of the buffer of Proc_3 until it reaches a message $\langle \mathsf{x}, 2\rangle$. But due to the order of the messages in the buffer, this means that it lost all messages $\langle \mathsf{x}, 1\rangle$ and also, as said previously, cannot add any more of them. It follows that the process can never execute the next instruction $\mathtt{rd}(\mathsf{x}, 1)$ and is thus stuck in q_5. Since this is not a deadlock for the process player, it results in a winning play for her.

We conclude that TSO safety games with process fairness do not have the same winning configurations under store buffer semantics and load buffer semantics, respectively. The same can be shown for reachability games with update fairness. Since it does not yield any additional insights, we do not present the argument here.

10 Conclusion and Future Work

In this paper, we continue the work on two-player games played on programs running under TSO semantics. We present a game model where one player controls the instructions of the program and the other player controls the buffer updates. Our results show that both the reachability problem and the safety problem for these games reduce to the analysis of games on single-process programs. Moreover, we show a bisimilarity to a game with a finite amount of configurations and use it to prove that the problems are in fact PSPACE-complete.

The reduced complexity comes from the optimal behaviour of the two players. The process player can always win by playing in only one single process, while the best strategy of the update player is to stay passive and not perform any buffer updates. We rectify this by introducing fairness conditions for both players. In reachability games, the update player is required to update each message eventually. This allows the process player to wait for a write instruction to arrive in the memory. In safety games, during an infinite run the process player has to perform instructions in all enabled processes infinitely often. Both restrictions lead to the respective problems being undecidable.

Finally, we connect the game model to the alternative load buffer semantics of TSO. We show that the equivalence between load buffer and store buffer that exists for classical TSO reachability does not carry over to the game setting.

This work analyses the basic winning conditions reachability and safety. Future work may expand the focus to more expressive winning conditions, like Büchi (i.e. repeated reachability), Co-Büchi, Parity, Rabin, Streett or Muller. Another way to expand is to look at other fairness conditions for the two players, for example transition fairness. These two directions of research are not orthogonal to each other since Muller or even Streett conditions might be able to encode some forms of fairness conditions.

References

[1] 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.

[2] 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.

[3] 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.

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

[5] Renato Acampora, Luca Geatti, Nicola Gigante, Angelo Montanari & Valentino Picotti (2022): *Controller Synthesis for Timeline-based Games*. In Pierre Ganty & Dario Della Monica, editors: *Proceedings of the 13th International Symposium on Games, Automata, Logics and Formal Verification, GandALF 2022, Madrid, Spain, September 21-23, 2022*, *EPTCS* 370, pp. 131–146, doi:10.4204/EPTCS.370.9.

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

[7] André Arnold, Aymeric Vincent & Igor Walukiewicz (2003): *Games for synthesis of controllers with partial observation*. *Theor. Comput. Sci.* 303(1), pp. 7–34, doi:10.1016/S0304-3975(02)00442-5.

[8] 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.

[9] 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.

[10] Ralph-Johan Back & Cristina Cerschi Seceleanu (2004): *Contracts and Games in Controller Synthesis for Discrete Systems*. In: *11th IEEE International Conference on the Engineering of Computer-Based Systems (ECBS 2004), 24-27 May 2004, Brno, Czech Republic*, IEEE Computer Society, pp. 307–315, doi:10.1109/ECBS.2004.1316713.

[11] Ayca Balkan, Moshe Y. Vardi & Paulo Tabuada (2015): *Controller Synthesis for Mode-Target Games*. In Magnus Egerstedt & Yorai Wardi, editors: *5th IFAC Conference on Analysis and Design of Hybrid Systems, ADHS 2015, Atlanta, GA, USA, October 14-16, 2015*, *IFAC-PapersOnLine* 48, Elsevier, pp. 343–350, doi:10.1016/J.IFACOL.2015.11.198.

[12] Nicolas Basset, Marta Z. Kwiatkowska & Clemens Wiltsche (2014): *Compositional Controller Synthesis for Stochastic Games*. In Paolo Baldan & Daniele Gorla, editors: *CONCUR 2014 - Concurrency Theory - 25th International Conference, CONCUR 2014, Rome, Italy, September 2-5, 2014. Proceedings*, *Lecture Notes in Computer Science* 8704, Springer, pp. 173–187, doi:10.1007/978-3-662-44584-6_13.

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

[14] 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.

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

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

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

[23] Stephan Spengler (2024): *Reachability and Safety Games under TSO Semantics (Extended Version)*. arXiv:2405.20804.

[24] Stephan Spengler & Sanchari Sil (2023): *TSO Games - On the decidability of safety games under the total store order semantics*. In Antonis Achilleos & Dario Della Monica, editors: *Proceedings of the Fourteenth International Symposium on Games, Automata, Logics, and Formal Verification, GandALF 2023, Udine, Italy, 18-20th September 2023*, *EPTCS* 390, pp. 82–98, doi:10.4204/EPTCS.390.6.