

Parameterized Verification under TSO is PSPACE-Complete

PAROSH AZIZ ABDULLA, Uppsala University, Sweden

MOHAMED FAOUZI ATIG, Uppsala University, Sweden

ROJIN REZVAN, Sharif University, Iran

We consider parameterized verification of concurrent programs under the Total Store Order (TSO) semantics. A program consists of a set of processes that share a set of variables on which they can perform read and write operations. We show that the reachability problem for a system consisting of an arbitrary number of identical processes is PSPACE-complete. We prove that the complexity is reduced to polynomial time if the processes are not allowed to read the initial values of the variables in the memory. When the processes are allowed to perform atomic read-modify-write operations, the reachability problem has a non-primitive recursive complexity.

CCS Concepts: • **Theory of computation** → **Verification by model checking**; • **Software and its engineering** → **Formal software verification**.

Additional Key Words and Phrases: Model-Checking, Parameterized Verification, Weak Memory Models, Total Store Ordering

ACM Reference Format:

Parosh Aziz Abdulla, Mohamed Faouzi Atig, and Rojin Rezvan. 2020. Parameterized Verification under TSO is PSPACE-Complete. *Proc. ACM Program. Lang.* 4, POPL, Article 26 (January 2020), 29 pages. <https://doi.org/10.1145/3371094>

1 INTRODUCTION

A *parameterized system* consists of an arbitrary number of identical concurrent processes. *Parameterized verification* means that we analyze the correctness of the system regardless of the number of processes. Such systems have been extensively studied both theoretically and practically for almost three decades (see e.g. [Abdulla and Delzanno 2016; Apt and Kozen 1986; Bloem et al. 2016; German and Sistla 1992]), and it is the subject of one chapter of the recent Handbook of Model Checking [Abdulla et al. 2018c].

Most previous research on parameterized verification has been made under the fundamental assumption that the processes behave according to the classical Sequential Consistency (SC) semantics. Under SC, the processes perform read and write operations *atomically* on a set of shared variables, and the runs of the program consist of interleavings of the process executions. However, it is unrealistic to assume SC behaviors in modern applications. The reason is that, due to hardware and compiler optimizations, most modern platforms allow more relaxed program behaviors than those allowed under SC, leading to so called *weak memory models*. Weakly consistent platforms are found at all levels of system design such as multiprocessor architectures (e.g., [Sarkar et al. 2011; Sewell et al. 2010]), Cache protocols (e.g., [Elver and Nagarajan 2014; Ros and Kaxiras 2016]), language level concurrency (e.g., [Lahav et al. 2016]), and distributed data stores (e.g., [Burckhardt

Authors' addresses: Parosh Aziz Abdulla, Uppsala University, Sweden, parosh@it.uu.se; Mohamed Faouzi Atig, Uppsala University, Sweden, mohamed_faouzi.atig@it.uu.se; Rojin Rezvan, Sharif University, Iran, rojinrezvan@gmail.com.



This work is licensed under a Creative Commons Attribution 4.0 International License.

© 2020 Copyright held by the owner/author(s).

2475-1421/2020/1-ART26

<https://doi.org/10.1145/3371094>

2014]). Notwithstanding, very little research has been conducted on parameterized verification of concurrent programs under weak memory models. In fact, parameterized systems give rise to difficult verification problems already in the case of SC; and the task becomes even harder due to the intricate extra behaviors that are introduced due to weak consistency.

In this paper, we study parameterized verification of programs running on weak memory models. More precisely, we consider the decidability and complexity of parameterized verification, where an unbounded number of finite-state processes run concurrently under the *Total Store Order (TSO)* semantics. TSO is one of the most well-known memory models, and has previously been adopted by Sun's SPARC multiprocessors [Weaver and Germond 1994], and used as a formalization of the x86-TSO memory model [Owens et al. 2009; Sewell et al. 2010]. The operational model of programs running under TSO inserts an unbounded buffer between each process and the shared memory. The buffer contains a sequence of pending write messages that have been performed by the process but not yet reached the memory. Write operations are not atomic in the sense that they are initially stored in the buffer of the process, and hence they are not immediately visible to the rest of the processes. Messages can non-deterministically be used to *update* the memory, i.e., be moved from the buffer to the shared memory in a FIFO manner. A read operation on a variable x fetches the latest value of x stored in the buffer of the process. If there are no pending write operations on x in the buffer, the process reads the value of x from the memory. Since the buffers are unbounded, the system has an infinite state space even if the considered program is finite-state. In fact, the complexity of the reachability problem for finite-state programs is non-primitive recursive under the TSO semantics [Atig et al. 2010], as opposed to the well-known PSPACE complexity in the case of the SC semantics. Notice that the class of systems we consider in this paper are infinite in two dimensions, namely we have an unbounded number of processes, each with an unbounded buffer.

We analyze the complexity of the reachability problem for the case where we have an arbitrary number of finite-state processes. We prove that the problem is PSPACE-complete, thus reducing the non-primitive recursive complexity of the non-parameterized case. The proof of membership in PSPACE relies on a novel abstraction, called *pivot abstraction*, which is *exact* wrt. reachability, i.e., a given process state is reachable under the pivot semantics iff it is reachable under the concrete TSO semantics. The abstraction defines a scheme that translates each run of the concurrent program under TSO to a new run performed by a set of abstract processes that are executed in a sequential manner. The abstract states, called *views*, are defined to meet two necessary objectives, namely: (i) *exactness*: to enable the abstract processes to simulate the concrete program faithfully, and (ii) *compactness*: to allow reachability analysis to be performed using only a polynomial amount of space. The definition of views relies on the notion of an *assignment* which corresponds to a write operation performed by a process, assigning a given value to a shared variable. The definition of a view uses three key properties of assignments. (i) *Unbounded supply*: Due to parameterization, if any process can generate an assignment a , then any number of processes may generate a . In other words, there is an *unbounded supply* of a that can be used by read operations of the other processes. (ii) *Pivots*: These are points along a run of the program at which assignments are moved from the buffers of the processes to the memory. For each assignment a , we identify the unique pivot corresponding to the *first time* a hits the memory. (iii) *Rankings*: The pivot points induce a natural ranking on the set of assignments in which an assignment a is ranked lower than an assignment a' if the pivot point of a occurs before the pivot point of a' . An important observation is that, to generate a given assignment a , it is sufficient to generate all the assignments with a rank lower than that of a . Since there are finitely many assignments, we can run finitely many processes, called the (*assignment*) *providers* whose roles are to generate the assignments one after the other in increasing rank order. Pivot abstraction defines exactly the amount of information to store in the view so that we can run the providers while satisfying both the exactness and compactness criteria.

We also show a PSPACE lower bound for the reachability problem through a reduction from the reachability problem for 1-safe Petri nets. The reduction relies crucially on the use of initial values of the variables in the memory. This is also confirmed by the fact that, for the case where we do not allow the processes to read the initial values, we show that the reachability problem can be solved in polynomial time. The reason is that, now, we only need to keep track of the set of write operations that have been issued by the processes, leading to a polynomial-sized abstraction.

Finally, we explain why adding *read-modify-write* operations will make the model retrieve the non-primitive recursive complexity of the reachability problem. This is straightforward since the proof for the non-parameterized case involves only two processes [Atig et al. 2010]. If we allow read-modify-write operations (where a *sequence* of read and write operations are allowed to execute atomically), then such sequences can be used to identify two distinguished members among the set of processes, even when all the processes are initially identical, thus going back to the non-parameterized case.

Related Work. Parameterized verification has been studied for many years (see [Abdulla et al. 2018c; Bloem et al. 2016] for recent surveys of the field.) The problem was originally shown to be undecidable even when assuming that each process has a finite state space [Apt and Kozen 1986]. Therefore, special classes of systems have been studied. Such systems are characterized by their topology (un-ordered, arrays, trees, graphs, rings, etc), the allowed communication mechanisms (shared memory, rendez-vous, broadcast, lossy channels, etc), and the types of processes (anonymous, with IDs, with priorities, etc) [Delzanno et al. 2010; Emerson and Kahlon 2003, 2004; Esparza et al. 1999; Ganty and Majumdar 2012; Namjoshi and Trefler 2016]. Another line of research has been to define abstractions based on regular model checking [Abdulla 2012; Boigelot et al. 2003; Bouajjani et al. 2012; Kesten et al. 2001], monotonic abstraction [Abdulla et al. 2010], and symmetry reduction [Abdulla et al. 2016; Emerson et al. 2000; Kaiser et al. 2010].

One of the first decidability results was reported in [German and Sistla 1992], where the authors consider the verification of systems consisting of an arbitrary number of processes. In the model, the processes are finite-state machines that interact through rendez-vous communication. The paper shows that the model checking problem is EXPSpace-complete. In a series of more recent papers, parameterized verification has been considered in the case where the individual processes are push-down automata. In [Kahlon 2008] it is shown that the reachability problem is decidable when the system consists of an arbitrary number of identical push-down automata. The paper [Hague 2011] extends this result to the case of having one distinguished *leader* process together with arbitrarily many slave processes. In [Esparza et al. 2016] it is shown that the problem is in fact PSPACE-complete. The above results are extended further to the case of higher-order pushdown automata [La Torre et al. 2015], and to the case of push-down automata with dynamic thread creation [Muscholl et al. 2017]. In [Fortin et al. 2017] it is shown that the verification of regular properties of executions satisfying some stuttering conditions is NEXPTIME-complete.

Several papers have also considered parameterized verification of timed processes. The paper [Abdulla and Jonsson 2003] shows that reachability is decidable if each process has a single clock. The paper [Abdulla et al. 2004] shows that the problem becomes undecidable when allowing two clocks. The paper [Abdulla et al. 2018b] shows that the problem is PSPACE-complete.

In contrast to this paper, all the above works assume the SC semantics. The paper [Abdulla et al. 2018a] considers parameterized verification of programs running under TSO. However, the paper applies the framework of well-structured systems where the buffers of the processes are modelled as lossy channels, and hence the complexity of the algorithm is non-primitive recursive. In particular, the paper does not give any complexity bounds for the reachability problem (or any other verification problems). The PSPACE-completeness result of this paper represents a substantial

improvement from a theoretical point of view, and we believe that it will potentially help in designing more efficient algorithms for parameterized systems under TSO. The paper [Bouajjani et al. 2013] considers checking the robustness property against SC for parameterized systems running under the TSO semantics. However, the robustness problem is entirely different from reachability, and the techniques and results developed in the paper cannot be applied in our setting. In fact, the paper shows that the problem is EXPSpace-hard.

Outline. In the next two sections we give some preliminaries and recall the classical semantics of TSO. We define pivot abstraction in Section 4, and prove its correctness in Section 5. In Section 6 we show PSPACE-completeness of the reachability problem. In Section 7 we prove polynomial time complexity for the case where the processes are not allowed to read the initial values of the variables. Finally, in Section 8, we present some conclusions, discussions, and directions for future work.

2 PRELIMINARIES

Notation. We use \mathbb{N} and $\mathbb{B} = \{\text{true}, \text{false}\}$ to represent the sets of natural numbers and Boolean values respectively.

For sets A and B , we write $f : A \rightarrow B$ to denote that f is a (possibly partial) function that maps elements from A to B , and write $f(a) = \perp$ when f is undefined for a . We define $f[a \leftarrow b]$ to be the function f' such that $f'(a) = b$ and $f'(a') = f(a')$ if $a' \neq a$. We write $f : A \xrightarrow{\bullet} B$ to indicate that the function f is total. We use $[A \rightarrow B]$ and $[A \xrightarrow{\bullet} B]$ to represent the sets of functions resp. total functions from A to B .

Assume a finite set A . We let $|A|$ be the size of A . We use A^* to denote the set of finite words over A . For a word w , we let $|w|$ be the length of w , and for $i : 1 \leq i \leq |w|$, we let $w[i]$ be the i^{th} element of w , and $w[i \cdot \cdot j]$ to be the subword $w[i]w[i+1] \cdots w[j]$. We write $a \in w$ when $w[i] = a$ for some i . We define $\text{last}(w) := w[|w|]$, i.e., it is the last symbol that occurs in w . We say that w is *differentiated* if $w[i] \neq w[j]$ whenever $i \neq j$, i.e., w contains pairwise distinct elements. We use A^{Diff} to denote the set of all differentiated words over A . For a differentiated word $w \in A^{\text{Diff}}$ and $a \in w$, we define $\text{Pos}(w)(a)$ to be the unique i such that $w[i] = a$, i.e., it is the position in w where a occurs. We assume that $\text{Pos}(w)(a) = \perp$ in case $a \notin w$. For words w_1 and w_2 , we let $w_1 \bullet w_2$ be their concatenation.

We view a multiset over A as a function $M : A \xrightarrow{\bullet} \mathbb{N}$ where $M(a)$ gives the number of occurrences of a in M . We use A^* to be the set of finite multisets over A . For multisets $M_1, M_2 \in A^*$, we write $M_1 \leq M_2$ when $M_1(a) \leq M_2(a)$ for all $a \in A$. We define $M_1 + M_2 := M$ where $M(a) = M_1(a) + M_2(a)$ for all $a \in A$. If $M_1 \leq M_2$, we define $M_2 - M_1 := M$ where $M(a) = M_2(a) - M_1(a)$ for all $a \in A$.

Transition Systems. A *labeled transition system* is a triple $\langle C, C_{\text{init}}, L, \rightarrow \rangle$ where C is a (potentially infinite) set of *configurations*, $C_{\text{init}} \subseteq C$ is the set of *initial configurations*, L is a finite set of labels, and $\rightarrow \subseteq C \times L \times C$ is the *transition relation*. As usual, we write $c_1 \xrightarrow{l} c_2$ instead of $\langle c_1, l, c_2 \rangle \in \rightarrow$. We write $c_1 \rightarrow c_2$ to denote that $c_1 \xrightarrow{l} c_2$ for some $l \in L$. For sets of configurations $C_1, C_2 \subseteq C$, we write $C_1 \rightarrow C_2$ to denote that $c_1 \rightarrow c_2$ for some $c_1 \in C_1$ and $c_2 \in C_2$. We define \rightarrow^* to be the reflexive transitive closure of \rightarrow . If $C_1 \xrightarrow{*} C_2$ then we say that C_2 is *reachable* from C_1 . Sometimes, we write $C_1 \rightarrow c_2$ instead of $C_1 \rightarrow \{c_2\}$, and say that c_2 is reachable from C_1 , (and similarly for $\{c_1\} \rightarrow C_2$ and $\{c_1\} \rightarrow \{c_2\}$). A *run* ρ is a sequence

$$c_0 \xrightarrow{l_1} c_1 \xrightarrow{l_2} c_2 \cdots \xrightarrow{l_n} c_n$$

In such a case we write $c_0 \xrightarrow{n} c_n$ to emphasize that c_0 can reach c_n in n steps, and also write $c_0 \xrightarrow{\rho} c_n$ to emphasize that c_0 reaches c_n through the run ρ . Notice that $c \xrightarrow{*} c'$ iff $c \xrightarrow{n} c'$ for some $n \geq 0$ iff $c \xrightarrow{\rho} c'$ for some run ρ . Abusing notation, we write $c \in \rho$ to denote that $c = c_i$ for some $i : 0 \leq i \leq n$, i.e., c appears somewhere along the run. Similarly, we write $l \in \rho$ to denote that $l = l_i$ for some $i : 1 \leq i \leq n$. We define $\# \rho := n$. We say that ρ is *initialized* if $c_0 \in C_{init}$. We define $\text{start}(\rho) := c_0$ and $\text{end}(\rho) := c_n$, i.e., they are the first and end configurations in ρ respectively.

Two runs ρ_1 and ρ_2 are said to be *matching* if $\text{end}(\rho_1) = \text{start}(\rho_2)$. For matching runs $c_0 \xrightarrow{l_1} c_1 \cdots \xrightarrow{l_m} c_m$ and $c_m \xrightarrow{l_{m+1}} c_{m+1} \cdots \xrightarrow{l_n} c_n$, we define $\rho_1 \bullet \rho_2$ to be the run $c_0 \xrightarrow{l_1} c_1 \cdots \xrightarrow{l_m} c_m \xrightarrow{l_{m+1}} c_{m+1} \cdots \xrightarrow{l_n} c_n$. Furthermore, for runs ρ_1 and ρ_2 with $\text{end}(\rho_1) = c_1$, $\text{start}(\rho_2) = c_2$, and $c_1 \xrightarrow{l} c_2$, we define $[\rho_1] l [\rho_2]$ to be the run $\rho_1 \bullet (c_1 \xrightarrow{l} c_2) \bullet \rho_2$.

For $i, j : 0 \leq i, j \leq n$, we define the *sub-run* of ρ from i to j :

$$\rho[i \cdot j] := c_i \xrightarrow{l_{i+1}} c_{i+1} \xrightarrow{l_{i+2}} c_{i+2} \cdots \xrightarrow{l_j} c_j$$

In other words, it is the segment of ρ from i to j . For run ρ_1 and ρ_2 , we say that ρ_2 is a *sub-run* of ρ_1 if $\rho_2 = \rho_1[i \cdot j]$ for some i and j . If ρ_2 is a sub-run of ρ_1 then $\rho_1 = \rho_3 \bullet \rho_2 \bullet \rho_4$ for some ρ_3 and ρ_4 . As a special case, we define $\rho[i] := \rho[i \cdot i] = c_i$.

3 TOTAL STORE ORDERING (TSO)

In this section, we will recall the classical definition of the Total Store Order (TSO) semantics [Owens et al. 2009; Sewell et al. 2010], adapt it to the parameterized setting, and introduce the parameterized reachability problem over TSO.

3.1 Syntax

We assume a finite set \mathbb{X} of shared variables ranging over a finite domain \mathbb{D} of data values, where each variable $x \in \mathbb{X}$ has an initial value $\text{init}(x) \in \mathbb{D}$. We consider a set of processes that communicate through the shared variables. A *process definition* \mathcal{P} is a triple $\langle Q, q_{init}, \Delta \rangle$ where Q is a finite set of (process) states, $q_{init} \in Q$ is the *initial state*, and Δ is a finite set of *transitions*. A transition $\delta \in \Delta$ is a triple $\langle q, \sigma, q' \rangle$ where $q, q' \in Q$ are states and σ is an *instruction*. An instruction is either the empty instruction *skip*, a write instruction $w(x, d)$ where $x \in \mathbb{X}$ is a variable and $d \in \mathbb{D}$ is a value, a read instruction $r(x, d)$ with $x \in \mathbb{X}$ and $d \in \mathbb{D}$, or the memory fence instruction *mf*.

3.2 Semantics

We describe the concrete operational semantics of programs under TSO as a labeled transition system that is induced by a process definition. To that end, we will define the set of configurations and then define a transition relation on this set. In the TSO semantics, an unbounded FIFO buffer is inserted between each process and the shared memory. The buffer contains a sequence of “pending” write messages (write operations) of the process, each corresponding to the assignment of a given value to a given variable. More precisely, an *assignment* a is a pair $\langle x, d \rangle$ where $x \in \mathbb{X}$ is a variable and $d \in \mathbb{D}$ is a value. We let $\mathcal{A} = \mathbb{X} \times \mathbb{D}$ be the set of assignments. A *buffer state* represents the content of a buffer, and it is a member of the set \mathcal{A}^* . A (concrete) configuration of the system consists of the local states (process states and buffer contents) of a finite (but unbounded) set of processes, together with the state of the memory. Formally, a (concrete) *configuration* is a tuple $\gamma = \langle I, Q, \mathcal{B}, \mathcal{M} \rangle$ where I is a finite *index set*, each index representing one process, $Q : I \rightarrow Q$ defines the local states of the processes, $\mathcal{B} : I \rightarrow \mathcal{A}^*$ defines the buffer states, i.e., the buffer contents of the processes, and $\mathcal{M} : \mathbb{X} \rightarrow \mathbb{D}$ is the *memory state* defining the value of each variable

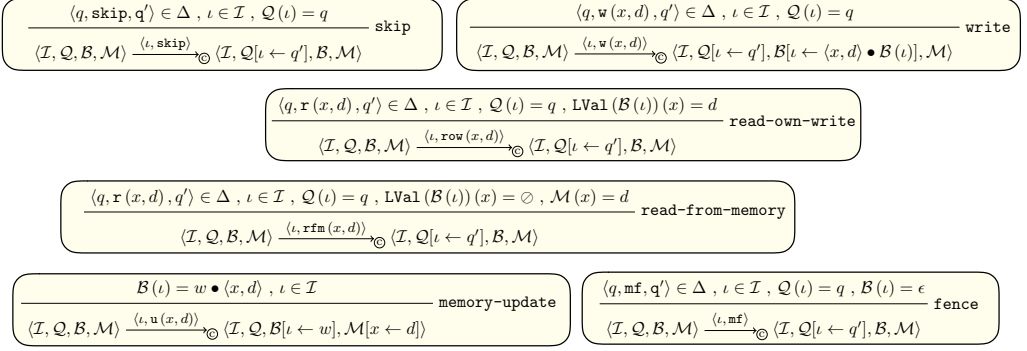


Fig. 1. The operational semantics of TSO. Here, we assume that $\iota \in \mathcal{I}$.

in the shared memory. We use C^\odot to denote the set of concrete configurations. Here, \odot stands for the *concrete* semantics as opposed to the different abstract semantics that we will introduce in the later sections. We define the function $\text{LVal} : \mathcal{A}^* \rightarrow [\mathbb{X} \rightarrow \mathbb{D} \cup \{\odot\}]$, where $\odot \notin \mathbb{D}$, such that (i) $\text{LVal}(w)(x) := d$ if $w = w_1 \bullet \langle x, d \rangle \bullet w_2$ and there is no $d' \in \mathbb{D}$ such that $\langle x, d' \rangle \in w_1$; and (ii) $\text{LVal}(w)(x) := \odot$ if there is no $d \in \mathbb{D}$ such that $\langle x, d \rangle \in w$. The function gives the value of the last pending write message on a variable if such a message exists; otherwise the value is given by the special “no-last” symbol \odot . We write $x \in w$ to denote that $\text{LVal}(w)(x) \neq \odot$, i.e., to indicate that x occurs in w .

The transition relation on configurations is defined through the set of inference rules shown in Fig. 1. The semantics defines a labeled transition relation $\xrightarrow[\odot]{\lambda}$ on the set of configurations. Each transition step is labeled with an *event* λ that corresponds to a process performing one step. Such a step is either the execution of an instruction, or performing a memory update. More precisely, an event is a pair $\lambda = \langle \iota, \text{op} \rangle$ where $\iota \in \mathcal{I}$ is the index of the process performing the transition, and op describes the operation performed by the process. An *operation* op is of one of the forms skip , $w(x, d)$, $\text{row}(x, d)$, $\text{rfm}(x, d)$, $u(x, d)$, or mf , where $x \in \mathbb{X}$ is a variable and $d \in \mathbb{D}$ is a value. The different types of operations are explained as we introduce the inference rules below. The skip operation only changes the state of the process. When a process executes a write instruction $w(x, d)$ the message $\langle x, d \rangle$ is appended to the end of its buffer. In order to perform a read instruction $r(x, d)$, the last write message on x in the buffer of the process should be of the form $\langle x, d \rangle$. In such a case we say that the process is performing a *read-own-write* operation. If there is no pending write message on x in the buffer of the process, the value of x in the memory should be d . In this case we say that the process is reading from the memory. At any point in the execution of the program, the write message (of form $\langle x, d \rangle$) at the head of the buffer of a process may non-deterministically be chosen, removed from the buffer, and used to update the value of x in the memory to d . A fence operation can be performed by a process only if its buffer is empty. We observe that the set of processes is not changed when performing transitions. However, the set \mathcal{I} of indices is not *a priori* bounded.

An *initial configuration* is of the form $\langle \mathcal{I}, \mathcal{Q}_{\text{init}}, \mathcal{B}_{\text{init}}, \mathcal{M}_{\text{init}} \rangle$, where $\mathcal{Q}_{\text{init}}(\iota) = q_{\text{init}}$, $\mathcal{B}_{\text{init}}(\iota) = \epsilon$ for all $\iota \in \mathcal{I}$, and $\mathcal{M}_{\text{init}}(x) = \text{init}(x)$ for all $x \in \mathbb{X}$. In other words, all the processes start from the initial state with an empty buffer, and the initial memory state is defined by the initial values of the variables. We use Γ_{init} to denote the set of initial configurations. The set Γ_{init} is infinite since there is no bound on the size of the set \mathcal{I} .

We define the types and arguments of operations. Consider an operation op . We define $op \cdot \text{type} \in \{\text{skip}, w, \text{row}, \text{rfm}, u, \text{mf}\}$ depending on the type of the operation. We define $op \cdot \text{var} := x$ and $op \cdot \text{val} := d$ where $x \in \mathbb{X}$ and $d \in \mathbb{D}$ are the variable and the value used in op respectively (if such an x and d exist). For instance, if op is of the form $\text{rfm}(x, d)$ then $op \cdot \text{type} = \text{rfm}$, $op \cdot \text{var} = x$, and $op \cdot \text{val} = d$. If op is of the form skip or mf then $op \cdot \text{var}$ and $op \cdot \text{val}$ are not defined. We extend these definitions to events. For an event $\lambda = \langle \iota, op \rangle$, we define $\lambda \cdot \text{index} := \iota$, $\lambda \cdot \text{opr} := op$, $\lambda \cdot \text{type} := op \cdot \text{type}$, $\lambda \cdot \text{var} := op \cdot \text{var}$, and $\lambda \cdot \text{val} := op \cdot \text{val}$.

We use \mathcal{R}^\odot to denote the set of initialized runs under the concrete semantics. Consider an initialized run:

$$\rho = \gamma_0 \xrightarrow{\lambda_1}_{\odot} \gamma_1 \xrightarrow{\lambda_2}_{\odot} \gamma_2 \cdots \xrightarrow{\lambda_n}_{\odot} \gamma_n \in \mathcal{R}^\odot$$

For an $i : 0 \leq i \leq n$ and a variable $x \in \mathbb{X}$, we define $\text{Clean}(\rho)(i)(x) := \text{true}$ if there are no $d \in \mathbb{D}$ and $j : 1 \leq j \leq i$ such that $\lambda_j \cdot \text{opr} = u(x, d)$, and define $\text{Clean}(\rho)(i)(x) := \text{false}$ otherwise. In other words, the value of x is *clean* in the memory up to the point i in ρ in the sense that the initial value of x has not been overwritten by any update operations. Notice that the runs of a program are *dirty-stable* in the sense that, once a variable is overwritten in the memory, i.e., once it becomes “dirty”, then it will remain dirty for the rest of the run.

3.3 The Reachability Problem

For a set Γ of configurations and a state q , we write $\Gamma \rightarrow_{\odot} q$ (resp. $\Gamma \xrightarrow{*}_{\odot} q$) to denote that there are configurations $\gamma \in \Gamma$, and $\gamma' = \langle I, Q, \mathcal{B}, \mathcal{M} \rangle$, such that $\gamma \rightarrow_{\odot} \gamma'$ (resp. $\gamma \xrightarrow{*}_{\odot} \gamma'$) and $Q(\iota) = q$ for some $\iota \in I$.

An instance of the *parameterized reachability problem* for TSO is given by a process definition $\langle Q, q_{\text{init}}, \Delta \rangle$ and a state $q_{\text{target}} \in Q$. The question is whether $\Gamma_{\text{init}} \xrightarrow{*}_{\odot} q_{\text{target}}$, i.e., whether q_{target} is reachable from some initial configuration.

Remark. In the class of systems we consider, all the processes start from the same initial state. However, the analysis we perform in the subsequent sections will go through even if we allow the processes to start from different initial processes, and thus allow different process definitions rather than a single one. The only requirement is that we allow an arbitrary number of processes to start from each initial state.

4 PIVOT ABSTRACTION

In this section we take the first step in showing that the parameterized reachability problem for TSO is in PSPACE. More precisely, we will define an abstraction, called *pivot abstraction*, that is *exact* wrt. the reachability problem in the sense that, for a given program and a given process state, the process state is reachable from an initial configuration under the pivot semantics iff it is reachable from an initial configuration under the concrete semantics.

Pivot abstraction relies on a scheme in which each concrete run $\rho \in \mathcal{R}^\odot$ of the concurrent program is translated to a new run ρ' performed by a set of processes that are executed sequentially (one after the other without overlapping). To derive the scheme, we will identify a set of *pivot* points along the original run ρ . Each pivot point corresponds to a unique assignment pair $a \in \mathcal{A}$. More precisely, it is the first update operation involving a that hits the shared memory in ρ . The pivot points induce a natural ranking on the set of assignments in which an assignment a is ranked lower than an assignment a' if the pivot point of a occurs before the pivot point of a' . A crucial observation is that to generate an assignment a , we only need to generate the assignments with a lower rank than a . The abstraction then runs one process at a time, each with the goal of generating

one particular assignment. The process generating the assignment a will start running only *after* the termination of all the processes that generate assignments that have lower ranks than a .

In the rest of the section we will formalize these ideas. First, we introduce the main concepts behind pivot abstraction, followed by an informal description of the pivot transition system. Then, we define the semantics formally by introducing the pivot transition system, i.e., the set of configurations and the transition relation on them, and adapt the definition of the reachability problem to the case of the pivot semantics. In the end of this section, we will describe some characteristics of program runs under the pivot semantics that we will use later to present the correctness arguments, and also illustrate the main concepts through a detailed example.

In the later sections, we show the correctness of pivot abstraction, i.e., show its soundness and completeness wrt. reachability, and also show that checking reachability on the set of abstract configurations can be carried out in polynomial space.

4.1 Concepts

Pivot abstraction keeps track of the order in which assignments are updated to the shared memory in a given concrete run of the program. In the rest of this sub-section, we fix an initialized concrete run

$$\rho = \gamma_0 \xrightarrow{\lambda_1}_{\odot} \gamma_1 \xrightarrow{\lambda_2}_{\odot} \gamma_2 \cdots \xrightarrow{\lambda_n}_{\odot} \gamma_n \in \mathcal{R}^{\odot}$$

We will identify certain *pivot points* and *pivot processes* along ρ that play a special role in our abstraction. We will do that in three steps: (i) We define the points at which a given assignment is updated to the memory for the first time. (ii) We identify the write operations that generate these updates. (iii) We identify the processes that generate these write operations. Furthermore, we will rank pivot updates according to the order in which they occur, and translate this into a ranking of assignments, write events, and processes.

For an assignment $a \in \mathcal{A}$, we define the first position in ρ where an update is performed using a . Formally:

$$\text{first}(\rho)(a) := \min \{i \mid \lambda_i \cdot \text{opr} = u(a)\}$$

Although the assignment a may occur multiple times in the buffers of the processes, the function first only considers the first time a hits the memory. Also, $\text{first}(\rho)(a) \neq \perp$ iff $\lambda_i \cdot \text{opr} = u(a)$ for some $i : 1 \leq i \leq n$, i.e., the function is defined exactly for the set assignments that are updated to the memory along ρ . We define the *update pivot* points along ρ to be exactly the points that correspond to the “first-time” updates of the different assignments:

$$\text{UPivots}(\rho) = \{i \mid (1 \leq i \leq n) \wedge (\exists a \in \mathcal{A}. \text{first}(\rho)(a) = i)\}$$

We rank the set of update pivot points according to the order in which they occur in ρ . For an $i \in \text{UPivots}(\rho)$, we define:

$$\text{rank}(\rho)(i) := |\{j \mid (j \leq i) \wedge (j \in \text{UPivots}(\rho))\}|$$

We define the rank of a run ρ to be the largest rank of an update operation in ρ :

$$\text{rank}(\rho) := |\text{UPivots}(\rho)|$$

The value of $\text{rank}(\rho)$ is equal to the number of distinct assignments that appear in ρ . For $k : 1 \leq k \leq \text{rank}(\rho)$, we define $\text{UPivot}(\rho)(k)$ to be the unique $i \in \text{UPivots}(\rho)$ such that $\text{rank}(\rho)(i) = k$. We extend the ranking to the set of assignments. For an assignment $a \in \mathcal{A}$ such that $\text{first}(\rho)(a) \neq \perp$, we define:

$$\text{rank}(\rho)(a) := \text{rank}(\rho)(\text{first}(\rho)(a))$$

In other words, we rank the assignments according to the first times they hit the memory. Given two assignments a and a' , we rank a lower than a' if the first message carrying a reaches the

memory before the first message carrying a' reaches the memory. The order among the rest of the messages carrying a or a' is not relevant for the ranking. We define the *pivot assignment sequence*:

$$\text{PASEq}(\rho) := a_1, a_2 \cdots a_n \text{ where } n = \text{rank}(\rho) \text{ and } \text{rank}(\rho)(a_i) = i$$

In other words, it is the sequence of assignments that are updated to the memory along ρ , ordered according to their ranks in ρ . Notice that $\text{PASEq}(\rho) \in \mathcal{A}^{\text{Diff}}$, and hence the $|\text{PASEq}(\rho)| \leq |\mathbb{X}| \cdot |\mathbb{D}|$ for any $\rho \in \mathcal{R}^{\odot}$.

Next, we derive the set of *write pivot points* of ρ from the set of update pivot points. For an $i : 1 \leq i \leq n$, where $\lambda_i = \langle \iota, u(a) \rangle$, we define $\text{GetW}(\rho)(i)$ to be the unique j such that:

$$(\lambda_j = \langle \iota, w(a) \rangle) \wedge |\{k \mid (1 \leq k < i) \wedge (\lambda_k = \langle \iota, u(a) \rangle)\}| = |\{k \mid (1 \leq k < j) \wedge (\lambda_k = \langle \iota, w(a) \rangle)\}|$$

This means that we identify the write corresponding to an update, i.e., both the update and the write operations are performed by the same process ι , and the number of the update operations performed by ι before i is equal to the number of write operations performed by ι before j . For an $i : 1 \leq i \leq n$, where $\lambda_i = \langle \iota, w(a) \rangle$, we define $\text{GetU}(\rho)(i)$ to be the unique j (if j exists) such that $\text{GetW}(\rho)(j) = i$. Notice that $\text{GetU}(\rho)(i) = \perp$ if there is no corresponding update in ρ , i.e., if the write message remains in the buffer until the end of the run.

We define the set of *pivot write points* in ρ . They are the points corresponding to write operations that induce the pivot updates.

$$\text{WPivots}(\rho) := \{i \mid \exists j \in \text{UPivots}(\rho). i = \text{GetW}(\rho)(j)\} \cup \{n+1\}$$

For technical convenience, the definition also adds the point $n+1$ (which is outside the run ρ). We rank the members of $\text{WPivots}(\rho)$ according to the update operations they induce, i.e., for an $i \in \text{WPivots}(\rho)$, we have:

$$\text{rank}(\rho)(i) := \begin{cases} \text{rank}(\rho)(\text{GetU}(\rho)(i)) & \text{if } 1 \leq i \leq n \\ \text{rank}(\rho) + 1 & \text{if } i = n+1 \end{cases}$$

The definition assigns the special rank $\text{rank}(\rho) + 1$ to the position $n+1$. For $k : 1 \leq k \leq \text{rank}(\rho) + 1$, we define $\text{WPivot}(\rho)(k)$ to be the unique $i \in \text{WPivots}(\rho)$ such that $\text{rank}(\rho)(i) = k$.

Finally, we extend the notion of a pivot to the set of processes. A process may generate more than one pivot update. Therefore, we define the rank of a process, with index ι as a set:

$$\text{rank}(\rho)(\iota) := \{\text{rank}(\rho)(i) \mid (i \in \text{UPivots}(\rho)) \wedge (\lambda_i \cdot \text{index} = \iota)\} \cup \{\text{rank}(\rho) + 1\}$$

Notice that we assign the special rank $\text{rank}(\rho) + 1$ to all the processes.

4.2 Informal Description

The pivot semantics simulates the behavior of the processes in a concrete run using a sequence of abstract processes that run one after the other. Each abstract process simulates the behavior of a concrete process with rank k for a given k . If the set of ranks of a concrete process is not a singleton, it is simulated by several abstract processes in the pivot semantics, namely one process for each of its ranks. The goal of each abstract process is to eventually provide the update of rank k for a given k . We call such an abstract process a *k-provider*. The simulation runs the *k-providers* sequentially one after the other for increasing values of k . The *k-provider* then simulates a concrete process with rank k from its initial state up to the point where it generates the write operation of rank k . The only values that the *k-provider* reads from memory (and that are not initial values) are the ones that correspond to assignments with rank less than k , i.e., assignments provided by ℓ -providers with $\ell < k$. Additionally, there is an abstract process, called the *verifier*, which is run last and whose role is to reach the target state. The verifier can use the updates generated by all the providers but will itself not provide any updates used by the other processes. We can assume without loss of

generality that there is only one verifier in ρ . The reason is that the reachability problem asks for a target state, and it is sufficient that a single process reaches the target state in order to have a positive instance. Furthermore, the verifier does not provide any variable updates to the providers. For convenience, we sometime refer to the verifier process as the $(\text{rank}(\rho) + 1)$ -provider.

In a run of the system, the same assignment a may be used multiple times. However, the pivot semantics uses only one provider to provide all instances of a . The justification lies in the fact that, due to parameterization, if any process can provide a then an arbitrary number of processes may provide a . This implies an *unbounded-supply* property: if any process has a pending write message available in its buffer (corresponding to the assignment a), then there is an unbounded supply of such write messages that can be used by read operations of the other processes. In particular, once a has hit the memory, then an unbounded supply of a will be available, since any number of processes may perform identical steps, reaching a point where they can provide a . Notice that the property does not hold in a single run, but it holds over the set of all runs. However, this is exactly what we need to check safety properties: If a process can reach a state then, for any k , there is another run in which k processes will reach the same state.

4.3 The Pivot Transition System

Based on the concepts introduced above, we describe the pivot transition system. For a given program, the pivot transition system will simulate the concrete transition system of Section 3. Fix a process definition $\langle Q, q_{\text{init}}, \Delta \rangle$. First, we define the pivot configurations which we call *views*, and then define the pivot transition relation through a set of inference rules.

Roughly speaking, a view is a data-structure that describes the current state of some k -provider along a run with a given pivot sequence of assignments. The definition of views allows to achieve two goals, namely: (i) to enable the processes to carry out the simulation faithfully, and (ii) to allow reachability analysis to be performed using only a polynomial amount of space. We define the views in a stepwise manner, by identifying the types of information the providers need to store locally. To have a faithful abstraction, a provider needs to know (i) the state of the concrete process it is simulating, and (ii) the values the process can currently read from the different variables. Handling the state is straightforward, since we can simply allow a view to store the current state of the process. Handling readable values is more complicated, since write messages may be anywhere inside the (unbounded) buffer of the process, and also in the shared memory. To deal with this problem, we recall from the semantics of TSO (Section 3) that a process may fetch the value of a variable either by reading its own writes, or by reading from the memory. For reading own writes, we let the view data-structure store the latest value the process has written to each variable in the program. For reading from the memory, we let the view for a k -provider use the set of assignments up to rank $(k - 1)$. Such assignments are the values that the k -provider needs in order to perform its simulation. Finally, to simulate the reading of the initial value of a variable, we enable the provider to decide whether it has ever written to that variable, and whether the variable is clean in the memory. To that end, we let a view store the highest ranked assignment the process has observed up to the current point of the run. The provider can now tell precisely whether it has already observed an update on a variable or not. More precisely, it can check the sequence of seen assignments and check whether the given variable occurs in the sequence or not.

A *view* v is a tuple $\langle q, \mathcal{L}, \omega, \phi_E, \phi_L, \phi_P \rangle$. The tuple represents the “view” of a provider when simulating a concrete run $\rho \in \mathcal{R}^\odot$ of the program. Here, $q \in Q$ is a process state. The function $\mathcal{L} : \mathbb{X} \rightarrow \mathbb{D} \cup \{\odot\}$ describes the latest (i.e., most recent) write instruction the process has performed on a variable. If such an instruction exists and it is of the form $w(x, d)$ then $\mathcal{L}(x) = d$. If the process has not performed any write operations on x and x is clean in the memory then $\mathcal{L}(x) = \odot$.

We define \mathcal{L}_\emptyset such that $\mathcal{L}_\emptyset(x) = \emptyset$ for all $x \in \mathbb{X}$. Furthermore, $\omega \in \mathcal{A}^{\text{Diff}}$ is a differentiated word that gives the pivot assignment sequence of the simulated concrete run ρ . This word is not changed when performing transitions, and hence it will remain the same throughout the whole pivot run. Notice that $\text{Pos}(\omega)(a) = \text{rank}(\rho)(a)$. The *External pointer*, $\phi_E \in \{0, 1, \dots, |\omega|\}$ helps the provider to keep track of the sequence of assignments, possibly performed by other processes, that the simulated process has observed. More precisely, for any assignment $a \in \omega[1 \cdot \phi_E]$ either a itself or some assignment a' with $\text{rank}(\rho)(a') > \text{rank}(\rho)(a)$ and $\text{rank}(\rho)(a') \in \omega[1 \cdot \phi_E]$ has been observed by the process. The *Local pointer* $\phi_L : \mathbb{X} \rightarrow \{0, 1, \dots, |\omega|\}$ is a set of pointers, one for each variable $x \in \mathbb{X}$. It stores the highest ranked write operation the process itself has performed (on any variable) before it performed the latest write on x . We define $\phi_L^{\max} := \max\{\phi_L(x) \mid x \in \mathbb{X}\}$, i.e., it is the local pointer with the highest value, and define ϕ_L^0 such that $\phi_L^0(x) = 0$ for all variables $x \in \mathbb{X}$, i.e., the pointers are all in the leftmost position. The *Progress pointer* $\phi_P \in \{1, 2, \dots, |\omega| + 1\}$ gives the rank of the process the current provider is simulating. We define $v \cdot \text{state} := q$, $v \cdot \text{LWrite} := \mathcal{L}$, $v \cdot \text{eptr} := \phi_E$, $v \cdot \text{stamp} := \omega$, $v \cdot \text{lptr} := \phi_L$, $v \cdot \text{maxlptr} := \phi_L^{\max}$, and $v \cdot \text{pptr} := \phi_P$. We define the *signature* $\text{sig}(v) := \langle \phi_P, \phi_E \rangle$, i.e., it is the values of progress and external pointers of the view (in that order). As we shall see when we define the inference rules, all the views we generate will satisfy the invariant that $\phi_E < \phi_P$, and $\phi_L(x) < \phi_P$ for all variables $x \in \mathbb{X}$, i.e., the values of the external and internal pointers are all smaller than the progress pointer. This reflects the fact that a k -provider only uses (i.e., reads and writes) assignments with ranks lower than k . In particular, the second property implies that $\phi_L^{\max} < |\phi_P|$. We use \mathcal{V} to denote the set of views.

For $\omega \in \mathcal{A}^{\text{Diff}}$ and $k : 1 \leq k \leq |\omega| + 1$, we define the *initial view induced by ω and k* as $v_{\text{init}}(\omega)(k) := \langle q_{\text{init}}, \mathcal{L}_\emptyset, \omega, 0, \phi_L^0, k \rangle$. This view reflects initial state of the k -provider. The process is in its initial state q_{init} , and it has not performed any write operations (indicated by \mathcal{L}_\emptyset). Furthermore, it has not observed any assignments, and hence its external pointer is 0. Since the process has not performed any write operation yet, its local pointer is also 0 for every variable. The value of the progress pointer is k , i.e., the provider is allowed to use assignments up to rank $k - 1$.

We define the pivot semantics by defining a labeled transition relation $\xrightarrow{\text{op}}_{\text{P}}$ on the set of views, as described by the inference rules in Fig. 2. Since we do not have process indices in the pivot semantics, we represent an event simply by the corresponding operation which is of one of the forms skip , $w(1)(x, d)$, $w(2)(x, d)$, $r(1)(x, d)$, $r(2)(x, d)$, $r(3)(x, d)$, and mf , with $x \in \mathbb{X}$ and $d \in \mathbb{D}$.

The inference rule skip reflects the situation where there is a transition taking a process from a state q to a state q' through the skip instruction. The state of the process changes from q to q' while the latest-writes on variables, and the pointers are not affected.

We have two rules describing the execution of a write instruction $w(x, d)$. In $\text{write}(1)$, the process uses an assignment whose rank is smaller than its progress pointer. In such a case, the process updates its state and changes the value of its latest write operation on x to d . Furthermore, it updates the local pointer for x to indicate that its value will now be the maximal value among all local pointers and at least as large as the rank of the current assignment $\langle x, d \rangle$. Updating $\mathcal{L}(x)$ to d implies that the process has already performed a write instruction on x . While $\mathcal{L}(x)$ may later be updated to other values, its value will never change back to \emptyset in the run.

In $\text{write}(2)$, the process uses an assignment whose rank is equal to its progress pointer. This means that the current process, which is the ϕ_P -provider has “accomplished its mission” and generated the needed assignment, i.e., $\omega[\phi_P]$. In such a case, the process stops its execution, and initiates the next provider, i.e., the $(\phi_P + 1)$ -provider. The new provider starts from its initial view.

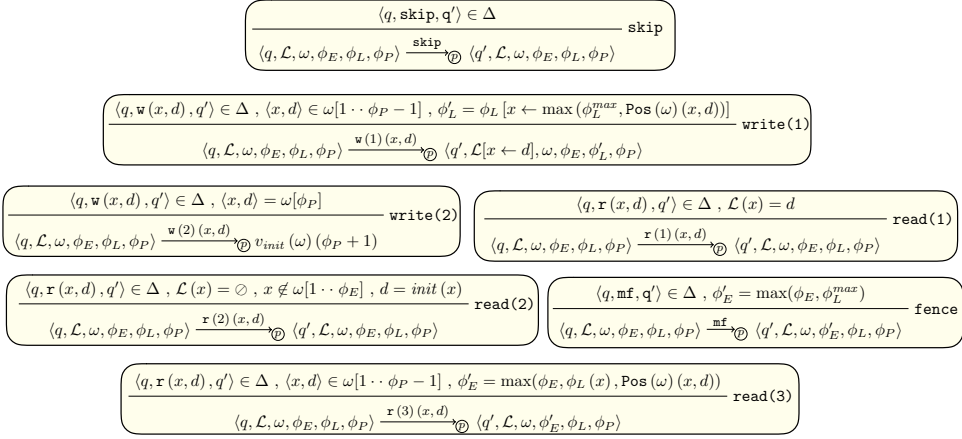


Fig. 2. Transitions of the pivot semantics.

We have three rules describing the execution of a read instruction $r(x, d)$. In $\text{read}(1)$, the latest write operation by the process on the variable assigns d to x . Therefore, the process has the possibility of performing a read-own-write operation. In such a case, the pointers are not affected.

In the rule $\text{read}(2)$, the process has not performed any write operation on x (as implied by the condition $\mathcal{L}(x) = \emptyset$), and it has not seen any updates on x (as implied by the condition $x \notin \omega[1 \cdot \phi_E]$). Therefore, the process can read the initial value of x from the memory. Even here, the pointers are not affected.

In the rule $\text{read}(3)$, the process has either not performed any write operation on x or the latest value it has assigned to x is different from d . The given assignment should be within the progress pointer. We move the external pointer to the position given by $\max(\phi_E, \phi_L(x), \text{Pos}(\omega)(x, d))$. The first argument, i.e., ϕ_E implies that the value of the external pointer is never decreased, reflecting the fact that the sequence of seen updates will at least remain the same after a transition. The second argument, i.e., $\phi_L(x)$ means that the external pointer will become at least equal to the local pointer of x . This is because the message corresponding to the latest write operation by the process on x has already been moved to the memory (otherwise, we would not be able to read d from x .) This in turn implies that the assignments corresponding to all the preceding write operations must also have been transferred to the memory. Finally, the process has now observed the assignment $\langle x, d \rangle$ so its external pointer should reflect that.

In the rule fence , the process performs the fence operation. Since its buffer must be empty, all the assignments inside its buffer have been moved to the memory, and hence its external pointer is updated so that it is at least equal to the local pointers of all the variables.

Notice that all views in a run have the same stamp, since the stamp of the view is not changed by any inference rule. For a pivot run ρ , we define $\rho \cdot \text{stamp} := \omega$ where ω is the unique differentiated word such that $v \cdot \text{stamp} = \omega$ for all $v \in \rho$.

The set of initial views is given by $\mathcal{V}_{\text{init}} := \{v_{\text{init}}(\omega)(1) \mid \omega \in \mathcal{A}^{\text{Diff}}\}$. This is the set of possible initial views of the 1-provider. The set $\mathcal{V}_{\text{init}}$ is finite since the set $\mathcal{A}^{\text{Diff}}$ is finite. We use \mathcal{R}^{P} to denote the set of initialized runs under the pivot semantics.

4.4 The Reachability Problem under the Pivot Semantics

We define the reachability of process states in a similar manner to Section 3. For a state q and a set V of views, we write $V \xrightarrow{*}_{\mathcal{P}} q$ to denote that there are views $v \in V$ and v' such that $v' \cdot \text{state} = q$ and $v \xrightarrow{*}_{\mathcal{P}} v'$. An instance of the *reachability problem* under the pivot semantics is given by a process definition $\langle Q, q_{\text{init}}, \Delta \rangle$ and a state $q_{\text{target}} \in Q$. The question is whether $\mathcal{V}_{\text{init}} \xrightarrow{*}_{\mathcal{P}} q_{\text{target}}$.

4.5 Epochs, Phases, and Stages

We will take a closer look at runs under the pivot semantics and extract three concepts, namely *epochs*, *phases*, and *stages*, that we will later use to prove the correctness of the abstraction. Consider an initialized run $\rho \in \mathcal{R}^{\mathcal{P}}$ in the pivot semantics:

$$v_0 \xrightarrow{\text{op}_1}_{\mathcal{P}} v_1 \xrightarrow{\text{op}_2}_{\mathcal{P}} v_2 \cdots \xrightarrow{\text{op}_n}_{\mathcal{P}} v_n$$

From the manner in which the inference rules in Fig. 2 are defined, the run ρ contains always certain patterns, as described below.

Let i_1, i_2, \dots, i_m be the maximal sub-sequence of $1, 2, \dots, n$ such that $\text{op}_{i_j} = w(2) \left(a_j \right)$ for some $a_j \in \mathcal{A}$. Since any a can be the argument of a $w(2)$ operation at most once, we have that $a_l \neq a_j$ if $l \neq j$. Define $\omega := a_1.a_2 \cdots a_m \in \mathcal{A}^{\text{Diff}}$.

The run ρ consists of a number of consecutive segments, called *epochs*, along which all the views have progress pointers with identical values. The operations during an epoch are performed by a given k -provider, where $k : 1 \leq k \leq \text{rank}(\rho) + 1$, and hence the values of all the progress pointers are equal to k . For $k : 1 \leq k \leq |\omega| + 1$, we define the k -epoch of ρ , denoted $\text{epoch}(\rho)(k)$, to be the maximal sub-run ρ' of ρ such that $v \cdot \text{pptr} = k$ for all $v \in \rho'$. Now, we can write ρ as $\rho = [\rho_1] \text{op}_1 [\rho_2] \text{op}_2 [\rho_3] \cdots [\rho_m] \text{op}_m [\rho_{m+1}]$ where $\rho_k = \text{epoch}(\rho)(k)$, and $\text{op}_k = w(2)(a_k)$. In other words, the run ρ is the concatenation of $m + 1$ sub-runs, each with views whose progress pointers have a value that is one larger than of those in the previous sub-run. The transition from the k^{th} -sub-run to the $(k + 1)^{\text{th}}$ -sub-run is performed by a write transition of type $w(2)$ on the k^{th} element of the sequence ω , i.e., the assignment a_k . This follows from the fact that the only inference rule that changes the value of the progress pointer is $\text{write}(2)$ which increases the value by exactly one.

The k -epoch is itself composed of a number consecutive segments, called *phases*, where the views appearing along a phase have all identical signatures $\langle k, \ell \rangle$. For a run ρ , a $k : 1 \leq k \leq \text{rank}(\rho) + 1$, and an $\ell : 0 \leq \ell < k$, we define the $\langle k, \ell \rangle$ -phase of ρ , denoted $\text{phase}(\rho)(k)(\ell)$, to be the maximal sub-run ρ' such that $\text{sig}(v) = \langle k, \ell \rangle$ for all $v \in \rho'$. It is possible that $\text{phase}(\rho)(k)(\ell) = \epsilon$, which will be the case when no view v with $\text{sig}(v) = \langle k, \ell \rangle$ occurs along the run. We define the set of phases of the k -provider:

$$\text{phases}(\rho)(k) := \{\ell \mid \text{phase}(\rho)(k)(\ell) \neq \epsilon\}$$

By definition, it is always the case that $0 \in \text{phases}(\rho)(k)$, and $\ell \notin \text{phases}(\rho)(k)$ if $\ell \geq k$. We can write $\text{epoch}(\rho)(k)$ as $[\rho_{i_0}] \text{op}_{i_1} [\rho_{i_1}] \text{op}_{i_2} [\rho_{i_2}] \cdots [\rho_{i_m-1}] \text{op}_{i_m} [\rho_{i_m}]$ for some $m \geq 0$, where (i) $\text{phases}(\rho)(k) = \{i_0, i_1, i_2, \dots, i_m\}$. (ii) $0 = i_1 < i_2 < \cdots < i_m < k$. (iii) $\text{sig}(v) = \langle k, i_j \rangle$ for each $j : 1 \leq j \leq m$ and $v \in \rho_{i_j}$. (iv) either $\text{op}_{i_j} \cdot \text{type} = r(3)$ or $\text{op}_{i_j} \cdot \text{type} = \text{mf}$. In other words, the k -phase is the concatenation of m sub-runs for some m , each with views whose progress pointers are identical, and whose external pointers are identical within the same sub-run but strictly increasing from one sub-run to the next. This follows from the fact that the only inference rules that change the value of the external pointer are $\text{read}(3)$ and fence . These two rules never decrease the value of the external pointer but they may increase the external pointer by more than one.

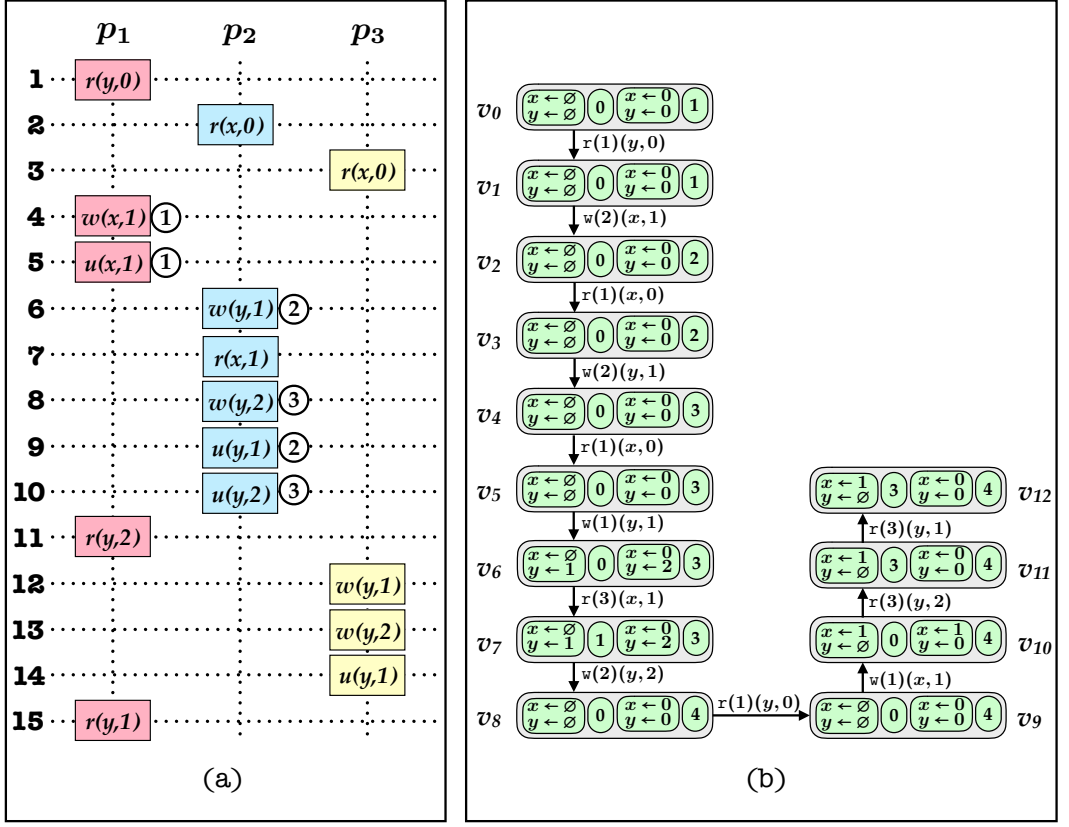


Fig. 3. (a) A concrete run ρ performed by three processes, with indices p_1 , p_2 , and p_3 respectively. The sequence of events is depicted from top to bottom. (b) The corresponding run ρ' in the pivot semantics.

Finally, a *stage* i in the $\langle k, \ell \rangle$ -phase is the i^{th} -view generated by the k -provider during phase ℓ . Formally, if $\ell \in \text{phases}(\rho)(k)$, and $i : 0 \leq i \leq \#(\text{phase}(\rho)(k)(\ell))$, then we define $\text{stage}(\rho)(k)(\ell)(i) := \text{phase}(\rho)(k)(\ell)[i]$. We define $\text{maxstage}(\rho)(k)(\ell) := \text{stage}(\rho)(k)(\ell)(\#(\text{phase}(\rho)(k)(\ell)))$, i.e., it is the last view that occurs in the $\langle k, \ell \rangle$ -phase.

4.6 Example

We illustrate the main concepts in the definition of the pivot semantics informally using the example of Fig. 3. We consider a concrete run ρ in Fig. 3(a), performed by three processes with indices p_1 , p_2 , and p_3 respectively. The run consists of 15 events. To simplify the notation in this sub-section, we represent each event by its operation. The events of one process are aligned vertically, and the events of all the processes are ordered from top to bottom (numbered 1 to 15). For instance, the first event of ρ is $r(y,0)$, performed by p_1 , and the second event is $r(x,0)$, performed by p_2 , etc. The update pivot points are 5, 9, and 10 with ranks 1, 2, and 3 respectively. In particular, this means that the rank of ρ is 3. The ranks are depicted as circles beside the corresponding events. Notice that the event 14 is not a pivot. The assignments $\langle x, 1 \rangle$, $\langle y, 1 \rangle$, and $\langle y, 2 \rangle$ have ranks 1, 2, and 3 respectively, and the pivot assignment sequence is $\langle x, 1 \rangle \langle y, 1 \rangle \langle y, 2 \rangle$. The write pivot points are 4, 6, 8, and 16 with ranks 1, 2, 3, and 4 respectively. The ranks of p_1 , p_2 , and p_3 are given by the sets $\{1, 4\}$, $\{2, 3, 4\}$, and $\{4\}$ respectively.

Fig. 3 (b) depicts the run ρ' that simulates ρ in the pivot semantics. It contains thirteen views v_0, \dots, v_{12} . The differentiated words of all these views are the same, namely the word $\langle x, 1 \rangle \langle y, 1 \rangle \langle y, 2 \rangle$. To simplify the presentation, we leave out the local states and the differentiated words from the representations of the views. We represent the other components, namely \mathcal{L} , ϕ_E , ϕ_L , and ϕ_P by four consecutive blocks. For instance, in the view v_7 , we have $(v_7 \cdot \text{LWrite})(x) = \emptyset$, $(v_7 \cdot \text{LWrite})(y) = 1$, $v_7 \cdot \text{eptr} = 1$, $(v_7 \cdot \text{lptr})(x) = 0$, $(v_7 \cdot \text{lptr})(y) = 2$, and $v_7 \cdot \text{pptr} = 3$. The run has four epochs, namely epoch 1 given by $v_0 v_1$, epoch 2 given by $v_2 v_3$, epoch 3 given by $v_4 v_5 v_6 v_7$, and epoch 4 given by $v_8 v_9 v_{10} v_{11} v_{12}$. Notice that the values of the progress pointers are equal for all the views within the same epoch, and that they increase by one from one epoch to the next.

The epochs 1 and 2 consists of one phase each. The epoch 3 consists of two phases, namely $\langle 3, 0 \rangle$ given by $v_4 v_5 v_6$, and $\langle 3, 1 \rangle$ which is the single view v_7 . The epoch 4 consists of two phases, namely $\langle 4, 0 \rangle$ given by $v_8 v_9 v_{10}$, and $\langle 4, 3 \rangle$ given by $v_{11} v_{12}$.

5 CORRECTNESS

We prove the correctness of the pivot semantics in the sense that the reachability problem under the concrete semantics is reducible to the reachability problem under the pivot semantics. This is achieved by Theorem 5.1, stated below. In the section, we fix a process definition $\langle Q, q_{\text{init}}, \Delta \rangle$.

THEOREM 5.1. $\Gamma_{\text{init}} \xrightarrow{*}_{\odot} q$ iff $\mathcal{V}_{\text{init}} \xrightarrow{*}_{\odot} q$, for any $q \in Q$.

We devote the rest of the section to give the main ideas in the proof of Theorem 5.1.

We carry out the proof in two steps. In Section 5.1 (Lemma 5.3) we show that the pivot semantics can simulate the concrete semantics. In Section 5.2 (Lemma 5.8) we show that the concrete semantics can simulate the pivot semantics.

5.1 From the Concrete Semantics to the Pivot Semantics

We show that each run in the concrete semantics can be simulated by a run in the pivot semantics. We do that in several steps. First, we extend the notions of external and internal pointers, that we introduced for the pivot semantics, to the concrete semantics. The extended definitions correspond to concrete interpretations of the pointers at different points of the concrete run. Next, we define a simulation relation that will allow us to formally capture the manner in which the pivot semantics simulates the concrete semantics. Finally, we give the proof of Lemma 5.3, based on the simulation relation. In the rest of this sub-section, we assume an initialized run in the concrete semantics:

$$\rho = \gamma_0 \xrightarrow{\lambda_1}_{\odot} \gamma_1 \xrightarrow{\lambda_2}_{\odot} \gamma_2 \cdots \xrightarrow{\lambda_n}_{\odot} \gamma_n \in \mathcal{R}^{\odot}, \text{ where } \gamma_i = \langle I, Q_i, \mathcal{B}_i, \mathcal{M}_i \rangle.$$

Pointers. We define the external pointer at a given point i along ρ to be the rank of the prefix of the run up to that point (equivalently, the highest rank of an assignment appearing in the prefix).

$$\psi_E(\rho)(i) := \text{rank}(\rho[1 \cdot i])$$

For a process with index $\iota \in I$, we define the internal pointer of ι wrt. to variable $x \in \mathbb{X}$ to be the highest rank of all the assignments that are in the buffer of ι and that have been issued before the latest assignment on x . We do that in two steps. For a word $w \in \mathcal{A}^*$ over the set of assignments, we define:

$$\psi_L(w)(x) := \max \{ \text{rank}(\rho)(a) \mid \exists w_1, w_2, d. (w = w_1 \bullet \langle x, d \rangle \bullet w_2) \wedge (a \in \langle x, d \rangle \bullet w_2) \}$$

and

$$\psi_L(\rho)(\iota)(i)(x) := \psi_L(\mathcal{B}_i(\iota))(x)$$

We define $\psi_L^{\max}(w) := \max \{ \psi_L(w)(x) \mid x \in \mathbb{X} \}$, and $\psi_L^{\max}(\rho)(\iota)(i) := \psi_L^{\max}(\mathcal{B}_i(\iota))$, i.e., it is the local pointer with the highest value.

Simulation Relation. Roughly speaking, the simulation relation describes how a given view $v = \langle q, \mathcal{L}, \omega, \phi_E, \phi_L, \phi_P \rangle$ captures the information about the current configuration of the system, from the perspective of a concrete process (with index $\iota \in I$) at a given point $i : 1 \leq i \leq n$ of the run ρ . Formally, we write $\rho \models_{\iota}^i v$ to denote that the following conditions are satisfied for each variable $x \in \mathbb{X}$:

- (1) $q = Q_{\iota}(\iota)$. The state of the process should be consistent with the state of the view.
- (2) If $\text{LVal}(\mathcal{B}_{\iota}(\iota))(x) = d \in \mathbb{D}$ then $\mathcal{L}(x) = d$. If the latest value written by the process on a variable x is d and the corresponding write message is still in the buffer of the process, then the same value should be available in the view.
- (3) If $\text{LVal}(\mathcal{B}_{\iota}(\iota))(x) = \emptyset$ and $\text{Clean}(\rho)(\iota)(x) = \text{true}$ then $\mathcal{L}(x) = \emptyset$ and $x \notin \omega[1 \dots \phi_E]$. This case is for handling the initial value of x . If the process has not issued a write operation on x and the value of x is clean in the memory then, according to the view, the process has not issued a write operation on x , and furthermore no assignment on x has been observed by the process.
- (4) $\max(\phi_E, \phi_L(x)) \leq \max(\psi_E(\rho)(\iota), \psi_L(\rho)(\iota)(x))$. The pointers of the view lag their counterparts in the concrete run. As seen in the inference rules of Fig. 2, the pointers of a view are advanced lazily (on demand). This ensures that the view can access at least the same values of the variables as the ones that are available at the current point of the concrete run. The reason is that at least the same set of initial values are available due to the pointer lag, and furthermore the possibility of advancing the pointers makes the reading of the rest of the values possible.

The following lemma shows that the pivot semantics can simulate the run ρ .

LEMMA 5.2. *For each $\iota \in I$, there is a v such that $v_{\text{init}}(\text{PSeq}(\rho))(1) \xrightarrow{*}_{\text{P}} v$ and $\rho \models_n^{\iota} v$.*

This gives the main lemma of the sub-section.

LEMMA 5.3. *For any $q \in Q$, if $\Gamma_{\text{init}} \xrightarrow{*}_{\text{C}} q$ then $\mathcal{V}_{\text{init}} \xrightarrow{*}_{\text{P}} q$.*

PROOF. Assume that $\Gamma_{\text{init}} \xrightarrow{*}_{\text{C}} q$. This means that $\gamma \xrightarrow{\rho}_{\text{C}} q$ for some $\gamma \in \Gamma_{\text{init}}$ and $\rho \in \mathcal{R}^{\text{C}}$ where $\text{end}(\rho)$ is of the form $\langle I, Q, \mathcal{B}, M \rangle$ where $Q(\iota) = q$ for some $\iota \in I$. Let $\omega = \text{PSeq}(\rho)$. By Lemma 5.2 it follows that there is a v such that $v_{\text{init}}(\omega)(0) \xrightarrow{*}_{\text{P}} v$ and $\rho \models_n^{\iota} v$ where $n = \#\rho$. This means that $\mathcal{V}_{\text{init}} \xrightarrow{*}_{\text{P}} v$. Since $\rho \models_n^{\iota} v$ it follows that $v \cdot \text{state} = q$. \square

5.2 From the Pivot Semantics to the Concrete Semantics

We show that each run in the pivot semantics can be simulated by a run in the concrete semantics. We do that in two steps. First, we introduce an “intermediate” semantics, which we call the *group* semantics in the form of a transition system that collects (groups) all the local states of the processes along a given run and puts them in one set. The configurations in the group semantics store two types of information, namely (i) the set of (concrete) local states of the processes, i.e., the set of process states together with the buffer states that have been seen so far in the execution of the program, and (ii) the current (concrete) memory state. The main difference between this transition system, and the concrete transition system is that the system is monotone. More precisely, the set of process states never shrinks through the application of an inference rule.

We will show that, for any program: (i) the group semantics simulates the pivot semantics, and (ii) the concrete semantics simulates the the group semantics. Properties (i) and (ii) immediately imply the main lemma of this sub-section (Lemma 5.8).

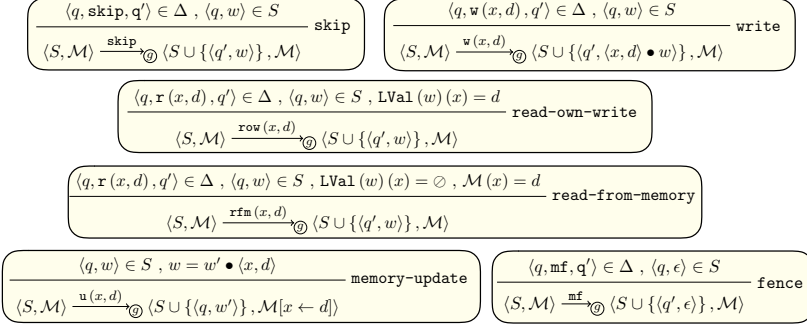


Fig. 4. The group transition relation.

5.2.1 The Group Semantics. We start by introducing the group transition system. A *local state* σ is a pair $\langle q, w \rangle$ where $q \in Q$ is a process state, and $w \in \mathcal{A}^*$ represents the content of the buffer. We define $\sigma \cdot \text{state} := q$, and $\sigma \cdot \text{buffer} := w$. A *group configuration* β is a pair $\langle S, \mathcal{M} \rangle$ where S is a set of local states, and $\mathcal{M} : \mathbb{X} \rightarrow \mathbb{D}$ is a memory state. We define the group operational semantics by defining a transition relation $\xrightarrow[\text{op}]{\text{op}}$ on the set of group configurations, as described by the inference rules in Fig. 4. The rules follow a similar pattern as the ones for the concrete semantics (Fig. 1). The main difference is in the monotonicity property, i.e., when a process performs a transition then its old state is not removed from the set of local states in the group configuration. Instead, we simply add the new state to the set. Therefore, the set of local states never shrinks through the application of the inference rules. Also here, we do not have process indices, and hence an event is simply given by the corresponding operation. An operation is of the same form as in the concrete semantics. We define the function Clean in the same manner as for the concrete semantics, i.e., it tells us whether there has been an update on a given variable along the run. We use C^{op} to denote the set of group configurations.

The *initial group configuration*, denoted β_{init} , is of the form $\langle S_{\text{init}}, \mathcal{M}_{\text{init}} \rangle$ where $S_{\text{init}} = \{\langle q_{\text{init}}, \epsilon \rangle\}$, and $\mathcal{M}_{\text{init}}$ is as defined in Section 3, i.e., $\mathcal{M}_{\text{init}}(x) = \text{init}(x)$, for all variables $x \in \mathbb{X}$.

We use \mathcal{R}^{op} to be the set of initialized runs in the group semantics.

5.2.2 From the Pivot Semantics to the Group Semantics. A run ρ_2 in the group semantics simulates a run ρ_1 in the pivot semantics by following the moves of the k -providers, all at the same time. At any point of time, the run ρ_2 is in some *phase* ℓ where it is simulating the ℓ -phases of all the k -providers in ρ_1 . During phase ℓ , each step of ρ_2 corresponds to performing one more step belonging to phase ℓ of a k -provider. When all the steps of phase ℓ of all the k -providers have been performed, the run ρ_2 will initiate the next phase, namely $\ell + 1$. Below, we will formalize these ideas by introducing a simulation relation.

To define our simulation relation, let us consider an initialized run $\rho_1 \in \mathcal{R}^{\text{p}}$ in the pivot semantics. We will derive an initialized run $\rho_2 \in \mathcal{R}^{\text{op}}$ in the group semantics that simulates ρ_1 . We will build the run ρ_2 incrementally, by observing the views that are generated in the steps of ρ_1 , while all the time ensuring that the simulation holds. Since $\rho_2 \in \mathcal{R}^{\text{op}}$, the set of local states along ρ_2 is non-decreasing, and hence the latest set of local states contains all the local states that have been generated up to now. Therefore, the simulation only needs to consider the latest set of local states along ρ_2 together with the latest memory state. The run ρ_2 will then store more and more local states corresponding to the views that are generated along ρ_1 . The simulation relation reflects the local states that should be added, based on the next view v that is observed along ρ_1 .

Formally, consider $k : 1 \leq k \leq \text{rank}(\rho_1) + 1$, $\ell \in \text{phases}(\rho_1)(k)$, and $0 \leq i \leq \text{maxstage}(\rho_1)(k)(\ell)$. Let $\text{stage}(\rho_1)(k)(\ell)(i) = v = \langle q, \mathcal{L}, \omega, \phi_E, \phi_L, \phi_P \rangle$ (Recall that $\ell = \phi_E$.) Let $\text{end}(\rho_2) := \langle S, \mathcal{M} \rangle$, and let $\sigma = \langle q, w \rangle \in S$. We use $\rho_1 \models_{\sigma}^{k, \ell, i} \rho_2$, denote that the following conditions are satisfied for each variable $x \in \mathbb{X}$ and each assignment $a \in \mathcal{A}$:

- (1) If $\mathcal{L}(x) = d \in \mathbb{D}$ then either (i) $\text{LVal}(w)(x) = d$, or (ii) $\text{LVal}(w)(x) = \emptyset$ and there is a $\theta \in S$ such that $\text{last}(\theta \cdot \text{buffer}) = \langle x, d \rangle$. If the latest write on x according to v is d , then either the latest write message on x carries the value d , or there is no write message on x in the buffer, but there is another process which has the assignment $\langle x, d \rangle$ last in its buffer. In the latter case, the second process can provide the assignment $\langle x, d \rangle$ through a single update operation in which no other variables but x will be overwritten in the memory.
- (2) If $x \notin \omega[1 \cdot \phi_E]$ and $\mathcal{L}(x) = \emptyset$ then $\text{LVal}(w)(x) = \emptyset$ and $\text{Clean}(\rho_2)(x) = \text{true}$. If v states that no write operations have been observed on x and the process itself has not issued any such write operations, then the buffer in σ does not contain any assignments on x , and furthermore x is clean in the memory. In both cases, the property implies that the process can read the initial value of x .
- (3) $\psi_L(w)(x) \leq \phi_L(x)$. The local pointers of the σ lag their counterparts in v .
- (4) If $\text{Pos}(a)(\omega) \leq \phi_E$ then there is a $\theta \in S$ such that $\text{last}(\theta \cdot \text{buffer}) = a$. If an assignment a has been noticed in v then there is a process that provides the same assignment as the last message in its buffer.

We use $\rho_1 \models_{\sigma}^{k, \ell, i} \rho_2$ to denote that $\rho_1 \models_{\sigma}^{k, \ell, i} \rho_2$, for some $\sigma \in S$. For $k : 1 \leq k \leq \text{rank}(\rho_1) + 1$, and $\ell : 0 \leq \ell < k$, we write $\rho_1 \models^{k, \ell} \rho_2$ to denote that either (i) $\ell \notin \text{phases}(\rho_1)(k)$, or (ii) $\rho_1 \models_{\sigma}^{k, \ell, i} \rho_2$ where $i = \text{maxstage}(\rho_1)(k)(\ell)$.

The following lemma shows that the group semantics can simulate the pivot semantics.

LEMMA 5.4. *For $\rho_1 \in \mathcal{R}^{\mathcal{D}}$ and $\ell \in \text{rank}(\rho_1) + 1$, there is a $\rho_2 \in \mathcal{R}^{\mathcal{G}}$ such that $\rho_1 \models^{\text{rank}(\rho_1)+1, \ell} \rho_2$.*

This leads to the following lemma.

LEMMA 5.5. *For any $q \in Q$, if $\mathcal{V}_{\text{init}} \xrightarrow{*}_{\mathcal{P}} q$ then $\beta_{\text{init}} \xrightarrow{*}_{\mathcal{C}} q$.*

PROOF. Assume that $\mathcal{V}_{\text{init}} \xrightarrow{*}_{\mathcal{C}} q$, i.e., $v_{\text{init}}(\omega)(0) \xrightarrow{*}_{\mathcal{P}} q$ for some $\omega \in A^{\text{Diff}}$, i.e., $v_{\text{init}}(\omega)(0) \xrightarrow{\rho_1}_{\mathcal{P}} q$ for some $\rho_1 \in \mathcal{R}^{\mathcal{D}}$. From Lemma 5.4 it follows that there is a run $\rho_2 \in \mathcal{R}^{\mathcal{G}}$ such that $\rho_1 \models^{\text{rank}(\rho)+1, \ell} \rho_2$ where ℓ is the largest number such that $\ell \in \text{rank}(\rho_1) + 1$. Recall that $\text{end}(\rho_1) = \text{maxstage}(\rho_1)(\text{rank}(\rho_1) + 1)(\ell)$. It follows that $\text{maxstage}(\rho_1)(\text{rank}(\rho) + 1)(\ell)$ is of the form $\langle q, \mathcal{L}, \omega, \phi_E, \phi_L, \phi_P \rangle$. Let $\text{end}(\rho_2) = \langle S, \mathcal{M} \rangle$. Since $\rho_1 \models^{\text{rank}(\rho)+1, \ell} \rho_2$ it follows that $\rho_1 \models_{\sigma}^{\text{rank}(\rho)+1, \ell} \rho_2$ for some $\sigma \in S$ of the form $\langle q, w \rangle$. The result follows immediately. \square

5.2.3 From the Group Semantics to the Concrete Semantics. A run ρ_2 in the concrete semantics simulates a run ρ_1 in the group semantics by allowing an *arbitrary* number of processes follow the moves of any process in ρ_1 all at the same time. This is again captured by a simulation relation, as described below.

Consider a group configuration $\beta = \langle S, \mathcal{M} \rangle$ and a concrete configuration $\gamma = \langle \mathcal{I}, Q, \mathcal{B}, \mathcal{M} \rangle$. For a natural number $n \in \mathbb{N}$, we write $\beta \models^n \gamma$ to denote that for each $\sigma = \langle q, w \rangle \in S$ there is an $\mathcal{I}' \subseteq \mathcal{I}$ such that the following conditions are satisfied:

- $|\mathcal{I}'| \geq n$
- $Q(i) = q$ for each $i \in \mathcal{I}'$
- $\mathcal{B}(i) = w$ for each $i \in \mathcal{I}'$.

In other words, γ agrees with β on the memory state, and contains at least n copies of each process in β . The following lemma follows immediately from the unbounded-supply property explained in Section 4.

LEMMA 5.6. For each $n \in \mathbb{N}$ and $\beta \in C^{\textcircled{g}}$, if $\beta_{\text{init}} \xrightarrow{*}_{\textcircled{g}} \beta$, then there is a $\gamma \in C^{\textcircled{c}}$ such that $\beta \models^n \gamma$ and $\Gamma_{\text{init}} \xrightarrow{*}_{\textcircled{c}} \gamma$.

This gives the following lemma.

LEMMA 5.7. For any $q \in Q$, if $\beta_{\text{init}} \xrightarrow{*}_{\textcircled{g}} q$ then $\Gamma_{\text{init}} \xrightarrow{*}_{\textcircled{c}} q$.

5.2.4 From the Pivot Semantics to the Concrete Semantics. Combining Lemma 5.5 and Lemma 5.7, we get the following lemma.

LEMMA 5.8. For any $q \in Q$, if $\mathcal{V}_{\text{init}} \xrightarrow{*}_{\textcircled{p}} q$ then $\Gamma_{\text{init}} \xrightarrow{*}_{\textcircled{c}} q$.

6 PSPACE-COMPLETENESS

In this section we prove the following theorem.

THEOREM 6.1. The parameterized reachability problem for TSO is PSPACE-complete.

The proof of Theorem 6.1 follows from Lemma 6.4 which shows membership in PSPACE, and Lemma 6.5 which shows PSPACE-hardness. Both proofs are achieved through reductions from/to the reachability problem for 1-safe Petri nets.

6.1 1-Safe Petri Nets

We recall the standard model of Petri nets. A *Petri Net* N is a tuple $\langle P, T, F \rangle$ where P and T are finite sets of *places* and *transitions* respectively, and $F \subseteq (P \times T) \cup (T \times P)$ is the *flow relation*. For a transition $t \in T$, we define *Input* (t) to be the multiset $M \in P^*$ over places such that $M(p) = 1$ if $\langle p, t \rangle \in F$ and $M(p) = 0$ otherwise. Sometimes, we view *Input* (t) as a set of places, which we call the set of input places of t , where the set contains a place p iff *Input* (t) (p) = 1. We define *Output* (t) analogously.

We define a transition system induced by N . A configuration of N , traditionally called a *marking*, is a multiset $M \in P^*$ over the set P . Sometimes, when $M(p) = k$, we say that the marking M puts k *tokens* in the place p . We define a transition relation \rightarrow_{PN} on the set of markings such that, for a transition $t \in T$, we have $M_1 \xrightarrow{t}_{\text{PN}} M_2$ if *Input* (t) $\leq M_1$ and $M_2 = M_1 - \text{Input}(t) + \text{Output}(t)$. We write $M_1 \rightarrow_{\text{PN}} M_2$ to denote that $M \xrightarrow{t}_{\text{PN}} M_2$ for some transition $t \in T$. A marking M is said to be *1-safe* if $M(p) \leq 1$ for all places $p \in P$. We say that N is 1-safe from a marking M if every reachable marking from M is 1-safe. The *reachability problem* for 1-safe Petri nets consists of a Petri net N and a marking M_{init} such that N is 1-safe from M_{init} , together with a place $p_{\text{target}} \in P$. The question is whether there is a marking M such that M is reachable from M_{init} , and $M(p_{\text{target}}) = 1$. In other words, we ask whether we can start from the initial marking and succeed in putting a token in p_{target} . The reachability problem for 1-safe Petri nets is PSPACE-complete [Cheng et al. 1995].

6.2 Membership

In this sub-section, we prove that the parameterized reachability problem for TSO is in PSPACE. We achieve that by reducing the reachability problem under the pivot semantics to the reachability problem for 1-safe Petri nets. We do that in two steps. First we show that the result holds in case when we only consider runs that consist of a single epoch. Then, we extend the result to general (initialized) runs.

6.2.1 *Single-Epoch Case*. Consider a (not necessarily initialized) run ρ in the pivot semantics, a differentiated word $\omega \in \mathcal{A}^{\text{Diff}}$ of assignments, and $k : 1 \leq k \leq |\omega| + 1$. We say that ρ is a $\langle \omega, k \rangle$ -epoch if (i) $v \cdot \text{stamp} = \omega$ for all $v \in \rho$, (ii) $v \cdot \text{pptr} = k$ for all $v \in \rho$, and (iii) $\text{start}(\rho) = v_{\text{init}}(\omega)(k)$.

Intuitively ρ is the k -epoch of some initialized run. We use Epochs $(\omega) (k)$ to denote the set of all $\langle \omega, k \rangle$ -epochs.

An instance of the *epoch reachability problem* is given by a process definition $\mathcal{P} = \langle Q, q_{init}, \Delta \rangle$, a word $\omega \in \mathcal{A}^{\text{Diff}}$, a $k : 1 \leq k \leq |\omega| + 1$, and a state $q_{target} \in Q$. The question is whether $v_{init}(\omega) (k) \xrightarrow{P} q_{target}$ for some $\rho \in \text{Epochs}(\omega) (k)$.

Assume that we are given an instance of the epoch reachability problem as defined above. We derive an instance of the reachability problem for 1-safe Petri nets as follows. We construct a Petri net $\llbracket \mathcal{P}, \omega, k, q_{target} \rrbracket_N := \langle P, \Delta, F \rangle$. Each marking of $\llbracket \mathcal{P}, \omega, k, q_{target} \rrbracket_N$ will represent a view $v = \langle q, \mathcal{L}, \omega, \phi_E, \phi_L, k \rangle$ along the given epoch, while the transitions will mimic the transitions of the pivot semantics that are executed along the epoch.

The set of places $P := P_1 \cup \dots \cup P_5$ is defined as the union of five disjoint sets, described below.

- P_1 contains a place $\text{St}(q)$ for each state $q \in Q$. A token in $\text{St}(q)$ means that the state of the current view is q . The transitions of N will preserve the invariant that $\sum_{q \in Q} M(\text{St}(q)) = 1$, which means that there is exactly one token in the places belonging to P_1 .
- P_2 contains a place $\text{L}(x, d)$, for each $\langle x, d \rangle \in \mathbb{X} \times (\mathbb{D} \cup \emptyset)$. A token in $\text{L}(x, d)$ means that $\mathcal{L}(x) = d$. The invariant $\sum_{d \in \mathbb{D} \cup \{\emptyset\}} M(\text{L}(x, d)) = 1$ is preserved for each $x \in \mathbb{X}$.
- P_3 contains a place $\text{EP}(i)$ for each $i : 0 \leq i \leq |\omega|$. A token in $\text{EP}(i)$ means that the current value of ϕ_E is equal to i . The invariant $\sum_{0 \leq i \leq |\omega|} M(\text{EP}(i)) = 1$ is preserved.
- P_4 contains a place $\text{LP}(x)(i)$ for each variable $x \in \mathbb{X}$ and $i : 0 \leq i \leq |\omega|$. A token in $\text{LP}(x)(i)$ means that the current value of $\phi_L(x)$ is equal to i . The invariant $\sum_{0 \leq i \leq |\omega|} M(\text{LP}(x)(i)) = 1$ is preserved for each variable $x \in \mathbb{X}$.
- P_5 contains a place $\text{MAXLP}(i)$ for each $i : 0 \leq i \leq |\omega|$. A token in $\text{MAXLP}(i)$ means that the current value of ϕ_L^{max} is equal to i . The invariant $\sum_{0 \leq i \leq |\omega|} M(\text{MAXLP}(i)) = 1$ is preserved.

The set of transitions $T := T_1 \cup \dots \cup T_6$ is defined as the union of six disjoint sets, defined as follows.

- For each $\langle q, \text{skip}, q' \rangle \in \Delta$, the set T_1 contains a transition t with $\text{Input}(t) = \{\text{St}(q)\}$ and $\text{Output}(t) = \{\text{St}(q')\}$. The transition mimics the rule `skip` in Fig. 2, changing the process state from q to q' while not changing the pointers.
- The set T_2 contains transitions each mimicking the application of the inference rule `write(1)` in Fig. 2. The set is itself defined as the union of two disjoint subsets $T_2 := T_{21} \cup T_{22}$. These two subsets reflect the manner in which the new position of the internal x -pointer is calculated, depending on the relative current values of (i) the maximal local pointer, and (ii) the position in ω of the assignment on which the write operation is performed. The value of external pointer should be updated to the larger of these two values. For each $\langle q, w(x, d), q' \rangle \in \Delta$, $\langle x, d \rangle \in \omega$, $\ell = \text{Pos}(\omega)(x, d) < k$, $i : 0 \leq i < k$, and $j : 0 \leq j < k$, the set Δ_2 contains the following transitions:
 - If $j \leq \ell$ then the set T_{21} contains a transition t with $\text{Input}(t) = \{\text{St}(q), \text{LP}(x)(i), \text{MAXLP}(j)\}$ and $\text{Output}(t) = \{\text{St}(q'), \text{LP}(x)(\ell), \text{MAXLP}(\ell)\}$. This is the case when the position of $\langle x, d \rangle$ in ω is higher than the current value of the maximal local pointer.
 - If $\ell < j$ then the set T_{22} contains a transition t with $\text{Input}(t) = \{\text{St}(q), \text{LP}(x)(i), \text{MAXLP}(j)\}$ and $\text{Output}(t) = \{\text{St}(q'), \text{LP}(x)(j), \text{MAXLP}(j)\}$. This is the case when the value of the maximal local pointer is higher than the position of $\langle x, d \rangle$ in ω .
- For each $\langle q, r(x, d), q' \rangle \in \Delta$, the set T_3 contains a transition t with $\text{Input}(t) = \{\text{St}(q), \text{L}(x, d)\}$ and $\text{Output}(t) = \{\text{St}(q'), \text{L}(x, d)\}$. The transition mimics the rule `read(1)` in Fig. 2.

- For each $\langle q, r(x, d), q' \rangle \in \Delta$ with $d = \text{init}(x)$, $i : 0 \leq i < k$ with $x \notin \omega[1 \cdot \cdot i]$, the set T_4 contains a transition t with $\text{Input}(t) = \{\text{St}(q), L(x, \emptyset), \text{EP}(i)\}$ and $\text{Output}(t) = \{\text{St}(q'), L(x, \emptyset), \text{EP}(i)\}$. The transition mimics the rule $\text{read}(2)$ in Fig. 2.
- The set T_5 contains a set of transitions each mimicking the application of the inference rule $\text{read}(3)$ in Fig. 2. In a similar manner to the set T_2 , the set is defined as the union of three disjoint subsets $T_5 := T_{51} \cup T_{52} \cup T_{53}$ reflecting the relative current values of the external pointer, the local pointer of variable x , and the position in ω of assignment on which the write operation is performed. For each $\langle q, r(x, d), q' \rangle \in \Delta$, $\langle x, d \rangle \in \omega$, $\ell = \text{Pos}(\omega)(x, d) < k$, $i : 0 \leq i < k$, and $j : 0 \leq j < k$, the set Δ_5 contains the following transitions:
 - If $i \leq \ell$ and $j \leq \ell$ then the set T_{51} contains a transition t with $\text{Input}(t) = \{\text{St}(q), \text{EP}(i), \text{LP}(x)(j)\}$ and $\text{Output}(t) = \{\text{St}(q'), \text{EP}(\ell), \text{LP}(x)(j)\}$.
 - If $i \leq j$ and $\ell < j$ then the set T_{52} contains a transition t with $\text{Input}(t) = \{\text{St}(q), \text{EP}(i), \text{LP}(x)(j)\}$ and $\text{Output}(t) = \{\text{St}(q'), \text{EP}(j), \text{LP}(x)(j)\}$.
 - If $j < i$ and $\ell < i$ then the set T_{53} contains a transition t with $\text{Input}(t) = \{\text{St}(q), \text{EP}(i), \text{LP}(x)(j)\}$ and $\text{Output}(t) = \{\text{St}(q'), \text{EP}(i), \text{LP}(x)(j)\}$.
- The set T_6 contains a set of transitions each mimicking the application of the inference rule fence in Fig. 2. The set is defined as the union of two disjoint subsets $T_6 := T_{61} \cup T_{62}$, reflecting the relative current values of the external pointer and the maximal local pointer. For each $\langle q, \text{mf}, q' \rangle \in \Delta$, $i : 0 \leq i < k$, $j : 0 \leq j < k$, the set Δ_6 contains the following transitions:
 - If $i \leq j$ then the set T_{61} contains a transition t with $\text{Input}(t) = \{\text{St}(q), \text{EP}(i), \text{MAXLP}(j)\}$ and $\text{Output}(t) = \{\text{St}(q'), \text{EP}(j), \text{MAXLP}(j)\}$.
 - If $j < i$ then the set T_{62} contains a transition t with $\text{Input}(t) = \{\text{St}(q), \text{EP}(i), \text{MAXLP}(j)\}$ and $\text{Output}(t) = \{\text{St}(q'), \text{EP}(i), \text{MAXLP}(j)\}$.

We define the initial marking:

$$M_{\text{init}} := \{\text{St}(q_{\text{init}}), \text{EP}(0), \text{MAXLP}(0)\} \cup \{L(x, \emptyset) \mid x \in \mathbb{X}\} \cup \{\text{LP}(x)(0) \mid x \in \mathbb{X}\}$$

This reflects the fact that we start the epoch from the initial process state, and with the external and all internal the pointers being equal to 0. We define the target place $p_{\text{target}} := \text{St}(q_{\text{target}})$.

Let us consider the size of the Petri net $\llbracket \mathcal{P}, \omega, k, q_{\text{target}} \rrbracket_N$. We observe that $|\omega| \in \mathcal{O}(|\mathbb{X}| \cdot |\mathbb{D}|)$, and hence $|P_1| \in \mathcal{O}(|Q|)$, $|P_2| \in \mathcal{O}(|\mathbb{X}| \cdot |\mathbb{D}|)$, $|P_3| \in \mathcal{O}(|\mathbb{X}| \cdot |\mathbb{D}|)$, $|P_4| \in \mathcal{O}(|\mathbb{X}|^2 \cdot |\mathbb{D}|)$, and $|P_5| \in \mathcal{O}(|\mathbb{X}| \cdot |\mathbb{D}|)$. Also, $|T_1| \in \mathcal{O}(|\Delta|)$, $|T_2| \in \mathcal{O}(|\Delta| \cdot (|\mathbb{X}|)^3 \cdot (|\mathbb{D}|)^3)$, $|T_3| \in \mathcal{O}(|\Delta|)$, $|T_4| \in \mathcal{O}(|\Delta|)$, $|T_5| \in \mathcal{O}(|\Delta| \cdot (|\mathbb{X}|)^3 \cdot (|\mathbb{D}|)^3)$, and $|T_6| \in \mathcal{O}(|\Delta| \cdot (|\mathbb{X}|)^3 \cdot (|\mathbb{D}|)^3)$. This gives the following lemma.

LEMMA 6.2. *The epoch reachability problem is polynomial-time reducible to the reachability problem for 1-safe Petri nets.*

From Lemma 6.2 and PSPACE-membership of the reachability problem for 1-safe Petri nets [Cheng et al. 1995] we get the following lemma.

LEMMA 6.3. *The epoch reachability problem is in PSPACE.*

6.2.2 General Case. We show how we can extend the PSPACE-membership from the pivot reachability problem to the general reachability problem under the pivot semantics (where we consider general runs rather than epochs). Recall from Section 4 that any initialized run $\rho \in \mathcal{R}^{\text{P}}$ can be written as $\rho = [\rho_1] \text{op}_1 [\rho_2] \text{op}_2 [\rho_2] \cdots [\rho_m] \text{op}_m [\rho_{m+1}]$ where $\rho_k = \text{epoch}(\rho)(k)$, and $\text{op}_k = \text{w}(2)(a_k)$ for some a_k . Indeed, each ρ_k is an $\langle \omega, k \rangle$ -epoch where $\omega = a_1 a_2 \cdots a_m$.

Algorithm 1 uses this pattern of pivot runs, and solves the pivot reachability problem non-deterministically. First, it guesses the sequence ω (line 1). The for-loop at line 2 checks that ω

Algorithm 1: A non-deterministic algorithm for solving the pivot reachability problem.

Input: $\mathcal{P} = \langle Q, q_{init}, \Delta \rangle$: process definition, $q_{target} \in Q$: process state.

Output: $\mathcal{V}_{init} \xrightarrow{*}_{\mathcal{P}} q_{target}?$

```

1  Guess some  $\omega \in (\mathbb{X} \times \mathbb{D})^{\text{Diff}}$ ;
2  for each  $k : 1 \leq k \leq |\omega|$  do
3       $flag := \text{false}$ ;
4      for each  $q, q' \in Q$  do
5          if  $\langle q, w(\omega[k]), q' \rangle \in \Delta$  then
6              if  $\exists \rho \in \text{Epochs}(\omega)(k) . v_{init}(\omega)(k) \xrightarrow{\rho}_{\mathcal{P}} q$  then
7                   $flag := \text{true}$ ;
8          if  $flag = \text{false}$  then
9              Return (false);
10 if  $\exists \rho \in \text{Epochs}(\omega)(|\omega| + 1) . v_{init}(\omega)(|\omega| + 1) \xrightarrow{\rho}_{\mathcal{P}} q_{target}$  then
11     Return (true)
12 else
13     Return (false)

```

can indeed be generated. To that end, the algorithm checks, for each $k : 1 \leq k \leq |\omega|$, that the $\langle \omega, k \rangle$ -epoch can generate $\omega[k]$. Finally, the if-statement of line 10 checks whether the target state can be generated in the last epoch, i.e., the $\langle \omega, |\omega| + 1 \rangle$ -epoch.

Algorithm 1 executes line 5 $O(|\mathbb{X}| \cdot |\mathbb{D}| \cdot |\Delta|)$ times, and executes line 10 once. According to Lemma 6.3 each execution of these two lines can be carried out in polynomial space. This means that Algorithm 1 operates in non-deterministic polynomial space. Applying Savitch's theorem [Savitch 1970], we get the following lemma:

LEMMA 6.4. *The parameterized reachability problem for TSO is in PSPACE.*

6.3 Hardness

In this section, we prove that the parameterized reachability problem for TSO is PSPACE-hard.

The proof is achieved through a reduction from the reachability problem for 1-safe Petri nets.

6.3.1 Reduction. Suppose that we are given an instance of the 1-safe reachability problem, consisting of a Petri net $N = \langle P, T, F \rangle$ together with two markings M_{init} and M_{target} . We will derive an equivalent instance of the parameterized reachability problem for TSO, consisting of a process definition $\langle Q, q_{init}, \Delta \rangle$ together with a state $q_{target} \in Q$. The processes participating in a run of the concurrent program may non-deterministically choose to play one of three different roles. The main role is that of a *simulator*, where the process (i) simulates the Petri net N and (ii) verifies that the target marking has been reached. The second role is that of a *trigger*, where the process signals (triggers) the end of the simulation process. The third role is that of a *sanity checker*, where the process verifies that no memory updates have been performed by a simulator. In fact, due to parameterization, an arbitrary number of processes may play the three different roles at the same time. However, this will not be important for the simulation. In particular, each simulator is made to work “in isolation” in the sense that it is not allowed to read from or update the memory (until the end of the simulation). This means that the simulators will not interfere with each other. This also implies that, at any point, a simulator may only read the latest value that it has written to a given variable, and not read from write operations performed by the other processes. For each

place $p \in P$, the set Q contains a variable x_p , ranging over the set $\{0, 1, 2\}$ with an initial value 2. Intuitively, the value of x_p is 1 if there is one token in p , while the value of x_p is 0 if p is empty (recall that p will contain at most one token since N is 1-safe from M_{init} .) Thus a marking M is encoded by assigning 1 to each variable x_p where $M(p) = 1$, and assigning 0 to x_p if $M(p) = 0$. We will also use an extra variable a that will be used to ensure that each variable x_p will not be updated in the memory. This variable also takes one of the values $\{0, 1, 2\}$ and its initial value is 0. In the main phase of the operation of the system, called the *simulation phase*, a simulator selects a transition in $t \in T$ non-deterministically, and simulates it by checking that all its input places contain tokens (the corresponding variables have the value 1) and by putting tokens in its output places (assigning the value 1 to the corresponding variables). Furthermore, we have an *initialization phase*, where we generate the encoding of the marking M_{init} , and a *final phase*, where we check: (i) that we have obtained the encoding of the marking M_{target} , and (ii) that there has not been any memory updates of the variables x_p in the memory during the simulation phase.

The Initialization Phase. The initialization phase is performed by the simulators, and its purpose is to generate the encoding of M_{init} . Let $\{p_1, \dots, p_m\}$ be the set of places such that $M_{init}(p_i) = 0$ for $i : 1 \leq i \leq m$; and let $\{q_1, \dots, q_n\}$ be the set of places such that $M_{init}(q_i) = 1$ for $i : 1 \leq i \leq n$. Notice that $P = \{p_1, \dots, p_m\} \cup \{q_1, \dots, q_n\}$. The set Q contains two sets of states $\{\text{init}_0, \dots, \text{init}_m\}$ and $\{\text{init}'_0, \dots, \text{init}'_n\}$. From the initial state q_{init} , a process may non-deterministically decide to become a simulator, and start the initialization phase. To that end, it performs a sequence of transitions initializing the values of the variables. More precisely, for each place p_i , the set Δ contains a transition $\langle \text{init}_{i-1}, w(x_{p_i}, 0), \text{init}_i \rangle$ which indicates that p_i is empty in M_{init} . Also, for each place q_i , the set Δ contains a transition $\langle \text{init}'_{i-1}, w(x_{q_i}, 1), \text{init}'_i \rangle$ which indicates that q_i contains one token in M_{init} . After the last transition in the sequence, the process enters the state origin from which the simulation of the transitions in T starts.

The Simulation Phase. The set Q contains the state origin from which the simulations of the transitions in T are started and ended. Consider a transition $t \in T$ with $\text{Input}(t) = \{p_1, \dots, p_m\}$ and $\text{Output}(t) = \{q_1, \dots, q_n\}$. The set Δ contains three sets of states $\{t_0^1, \dots, t_m^1\}$, $\{t_0^2, \dots, t_m^2\}$, and $\{t_0^3, \dots, t_n^3\}$. Starting from origin, the process may non-deterministically choose to execute the sequence of transitions corresponding to t . For each input place p_i , the sequence contains one transition of the form $\langle t_{i-1}, r(x_{p_i}, 1), t_i \rangle$ that checks that it is indeed the case that p_i contains a token in the current marking. Also, the sequence contains one transition of the form $\langle t_{i-1}, w(x_{p_i}, 0), t_i \rangle$ that simulates removing a token from p_i . Finally, for each output place q_i , the sequence contains one transition of the form $\langle t_{i-1}, w(x_{q_i}, 1), t_i \rangle$ that simulates putting a token in q_i . When the processes has executed the entire sequence, it goes back to the state origin.

The Final Phase. The final phase is initiated non-deterministically from the state origin. The goal of this phase is twofold. First, it checks that the marking M_{target} has been reached. Let $\{p_1, \dots, p_m\}$ be the set of places such that $M_{target}(p_i) = 0$ for $i : 1 \leq i \leq m$; and let $\{q_1, \dots, q_n\}$ be the set of places such that $M_{target}(q_i) = 1$ for $i : 1 \leq i \leq n$. The set Q contains the sets of states $\{\text{target}_0, \dots, \text{target}_m\}$ and $\{\text{target}'_0, \dots, \text{target}'_n\}$. For each place p_i , the set Δ contains a transition $\langle \text{target}_{i-1}, r(x_{p_i}, 0), \text{target}_i \rangle$ which indicates that p_i is empty in M_{target} . Analogously, for each place q_i , the set Δ contains a transition $\langle \text{target}'_{i-1}, r(x_{q_i}, 1), \text{target}'_i \rangle$ which indicates that p_i contains a token in M_{target} . After the last transition in the sequence, the process performs two additional transitions that are not part of the simulation of the Petri net N , namely it checks that the value of the special variable a is still equal to its initial value 0 in the memory, and that this

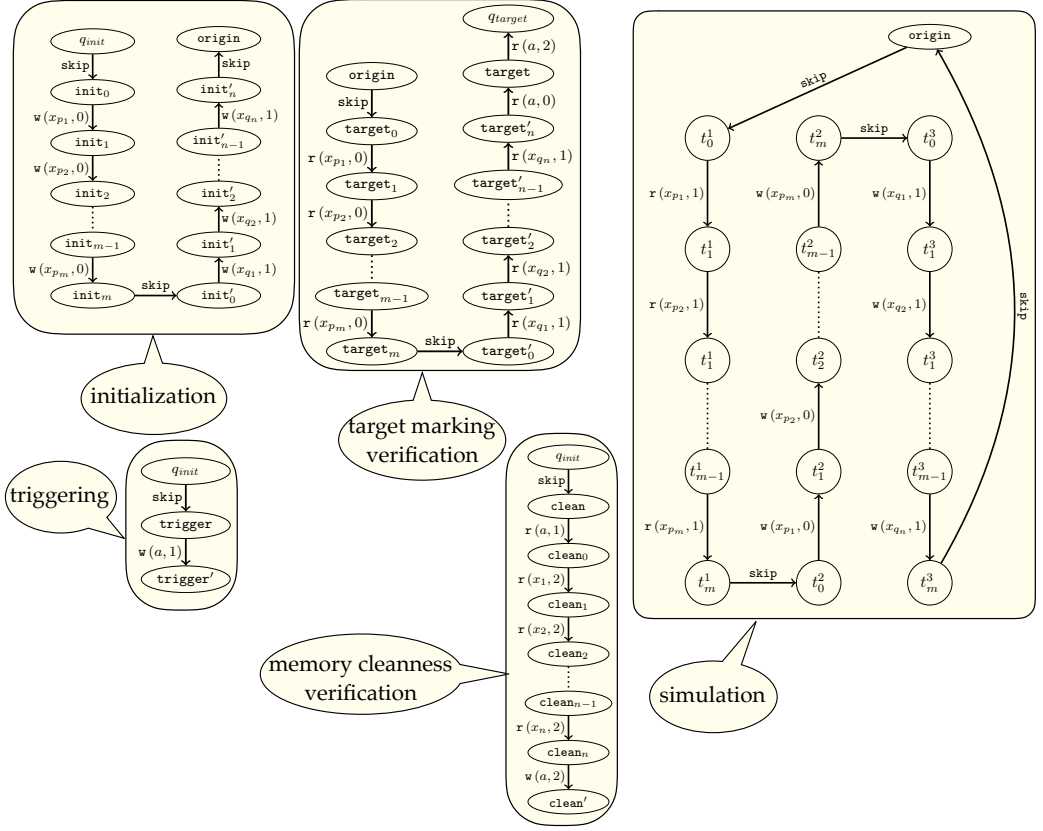


Fig. 5. Simulating a 1-safe Petri net.

value has later been updated to 2. If these two tests are satisfied, the simulator moves to the given target state q_{target} .

Also, in the final phase, we check that for any variable x_p , corresponding to a place p , the value of x_p in the memory has not been updated during the simulation by the simulator (the variable x_p is clean.) This will be carried out by the trigger and sanity checker processes. A process may non-deterministically decide to become a trigger from its initial state. A trigger performs only one operation, namely to assign the value 1 to the variable a . Also, a process may non-deterministically decide to become a sanity checker, and perform the following transitions. Let $\mathbb{X} = \{x_1, \dots, x_n\}$ be the set of variables. The set Q contains the set of states $\{clean, clean_0, \dots, clean_n\}$. It first checks that the special variable a has been updated to 1. This means that the subsequent transitions by the sanity checker are performed after the simulator has read the value 0 in a , i.e., after it has finished the simulation of N . After that, the process checks that all the other variables still carry their initial values, i.e, the value 2, in the memory. In its final step, it writes the value 2 to a thus signalling the end of the sanity checking and allowing the simulator to reach q_{target} .

From the reduction we immediately get the following lemma.

LEMMA 6.5. *The parameterized reachability problem for TSO is PSPACE-hard.*

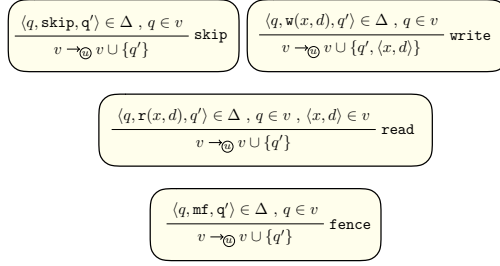


Fig. 6. The abstract semantics of TSO under the assumption of no initial values.

Notice that we do not need the fences to show PSPACE-hardness. This obviously implies that PSPACE-hardness holds in the presence of fences. Since we show PSPACE-membership in the presence of fences, the full problem is PSPACE-complete.

Also, observe that the reduction uses a domain containing three values, 0, 1, and 2. However, the result goes through even for Boolean programs. More precisely, we can use one extra variable that keeps track of whether the memory has been updated or not.

7 UNINITIALIZED MEMORY

In this section, we consider the case where the memory is uninitialized, i.e., the processes are not allowed to read from the initial values of the variables in the memory.

Consider a process definition $\langle Q, q_{init}, \Delta \rangle$. We assume that whenever $\mathcal{M}_{init}(x) = d_1$ and $\langle q, r(x, d_2), q' \rangle \in \Delta$ then $d_1 \neq d_2$. This means that no process will read the initial value of a variable in the memory. We define an abstract semantics for programs running under TSO under this condition. In a similar manner to Section 4 we define a set of abstract configurations and a transition relation on them. Here, the abstraction captures the fact that a process can always choose to read from a write operation performed by another process. Therefore, it is sufficient for the abstract configurations to store the states of the processes that have been observed, and the write operations that have been observed up to the current point of the execution of the program. A *view* v then is a set of process states and write operations i.e., $v \in Q \cup (\mathbb{X} \times \mathbb{D})$. We use \mathcal{V} to denote the set of views.

We define an abstract operational semantics by re-defining the transition relation on the set of views to obtain $\rightarrow_{@}$ (the u stands for uninitialized) as described by the inference rules in Fig. 6. The rule *skip* reflects the situation where there is a transition taking a process from a state q to a state q' through the *skip*-instruction. The *write* rule means that if a write transition $\langle q, w(x, d), q' \rangle$ is available, then any process in state q may perform the transition making both q' and the operation $\langle x, d \rangle$ available. We need only one rule for read transitions. More precisely, the rule *read* reflects the fact that if a process needs to perform a read transition $\langle q, r(x, d), q' \rangle$, then the transition may be performed provided that another process has already performed the write operation $\langle x, d \rangle$. As a result the state q' will become available in the new view. Notice that no new values of the variables will become available in this rule. Finally, the *fence* rule corresponds to the fact that a fence operation is always enabled since a process has always the possibility to flush its buffer and then perform the transition.

We observe that $|\mathcal{V}| = |Q| + |\mathbb{X}| \cdot |\mathbb{D}|$. Therefore, the reachability problem amounts to perform searching in a graph of polynomial size, thus giving a polynomial time complexity.

8 CONCLUSION, DISCUSSION AND FUTURE WORK

Bugs are found repeatedly in memory models, and in programs running on them. This makes the need for verification tools quite urgent. A crucial step in algorithm design and tool building is to understand the complexity of the given memory model and the corresponding verification problems. In this paper, we have taken the first step to study the complexity of parameterized verification under weak memory models. Concretely, we have presented decidability and complexity results for parameterized concurrent programs running on the classical TSO memory model. More precisely, we have shown that the reachability problem is PSPACE-complete when the system consists of an arbitrary number of identical processes. The complexity reduces to polynomial time when the processes are not allowed to read the initial values of the variables in the memory.

It is interesting to re-consider the problem under the assumption of having one or more distinguished processes (so called *leader processes*). In fact, the parameterized reachability problem when allowing two leaders has non-primitive recursive time complexity. This is a consequence of the fact that the reachability problem has a non-primitive time complexity in the non-parameterized case when we have two or more processes [Atig et al. 2010]. We can let the two leaders simulate the two processes, and make the rest of the processes (the slave processes) passive, by letting them have an empty set of transitions. The case with a single leader is less clear. For the class of finite-state processes, interacting through rendez-vous communication, under the SC semantics, the complexity of parameterized verification jumps from polynomial time to EXPSpace when adding a single leader [German and Sistla 1992]. We believe that pivot abstraction can be extended to the case of a single leader. The question then is whether we will stay within polynomial space, or exponential space is required to solve the problem. In fact, forbidding the processes from reading the initial values of the variables will not reduce the complexity to polynomial time in the presence of leaders, since a leader may run a preliminary phase where it initializes the values of the variables.

Also, the complexity of the reachability problem jumps to being non-primitive recursive when allowing atomic read-modify-write (RMW) operations. The reason is that two processes can initially use the RMW operation to declare themselves as leaders by modifying the shared variables atomically (e.g., by setting the values of two special flags), while blocking the other processes, thus reducing the problem to the case of having two leaders.

The complexity of checking correctness of parameterized system under TSO, assuming that the system is correct under the SC model, is still PSPACE-complete since our lower-bound reduction will still go through under this assumption.

It would be interesting to consider the decidability of the model checking problem wrt. logics such as LTL or CTL, or to allow additional program features such as dynamic creation and deletion of processes. Also, it is relevant to consider other memory models than TSO. This includes memory models such as PSO that are “similar” to TSO, and memory models such as POWER, ARM, and (fragments of) C11 that have entirely different behaviors. In particular, we would like to see whether parameterization allows to get decidability for models for which the reachability problem is undecidable in the non-parameterized case such as POWER [Derevenetc 2015] and the Released-Acquire fragment of C11 [Abdulla et al. 2019].

A difficult challenge is to consider parameterized verification for infinite-state processes. In the context of SC, it took almost two decades before the step from finite-state to infinite-state processes finally became possible. We believe this time span will be shorter in the case of weak memory models, given our knowledge of parameterized verification in the case of SC, and given that we already know how to extend the non-parameterized case from the SC context to the weak context.

REFERENCES

- Parosh Aziz Abdulla. 2012. Regular model checking. *STTT* 14, 2 (2012), 109–118. <https://doi.org/10.1007/s10009-011-0216-8>
- Parosh Aziz Abdulla, Jatin Arora, Mohamed Faouzi Atig, and Shankara Narayanan Krishna. 2019. Verification of programs under the release-acquire semantics. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019.*, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM, 1117–1132. <https://doi.org/10.1145/3314221.3314649>
- Parosh Aziz Abdulla, Mohamed Faouzi Atig, Ahmed Bouajjani, and Tuan Phong Ngo. 2018a. A Load-Buffer Semantics for Total Store Ordering. *Logical Methods in Computer Science* 14, 1 (2018). [https://doi.org/10.23638/LMCS-14\(1:9\)2018](https://doi.org/10.23638/LMCS-14(1:9)2018)
- Parosh Aziz Abdulla, Mohamed Faouzi Atig, Radu Ciobanu, Richard Mayr, and Patrick Totzke. 2018b. Universal Safety for Timed Petri Nets is PSPACE-complete. In *29th International Conference on Concurrency Theory, CONCUR 2018, September 4-7, 2018, Beijing, China (LIPIcs)*, Sven Schewe and Lijun Zhang (Eds.), Vol. 118. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 6:1–6:15. <https://doi.org/10.4230/LIPIcs.CONCUR.2018.6>
- Parosh Aziz Abdulla, Yu-Fang Chen, Giorgio Delzanno, Frédéric Haziza, Chih-Duo Hong, and Ahmed Rezine. 2010. Constrained Monotonic Abstraction: A CEGAR for Parameterized Verification, See [Gastin and Laroussinie 2010], 86–101. https://doi.org/10.1007/978-3-642-15375-4_7
- Parosh Aziz Abdulla and Giorgio Delzanno. 2016. Parameterized verification. *STTT* 18, 5 (2016), 469–473. <https://doi.org/10.1007/s10009-016-0424-3>
- Parosh Aziz Abdulla, Johann Deneux, and Pritha Mahata. 2004. Multi-Clock Timed Networks. In *19th IEEE Symposium on Logic in Computer Science (LICS 2004)*, 14-17 July 2004, Turku, Finland, *Proceedings*. IEEE Computer Society, 345–354. <https://doi.org/10.1109/LICS.2004.1319629>
- Parosh Aziz Abdulla, Frédéric Haziza, and Lukás Holík. 2016. Parameterized verification through view abstraction. *STTT* 18, 5 (2016), 495–516. <https://doi.org/10.1007/s10009-015-0406-x>
- Parosh Aziz Abdulla and Bengt Jonsson. 2003. Model checking of systems with many identical timed processes. *Theor. Comput. Sci.* 290, 1 (2003), 241–264. [https://doi.org/10.1016/S0304-3975\(01\)00330-9](https://doi.org/10.1016/S0304-3975(01)00330-9)
- Parosh Aziz Abdulla, A. Prasad Sistla, and Muralidhar Talupur. 2018c. Model Checking Parameterized Systems. In *Handbook of Model Checking.*, Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem (Eds.). Springer, 685–725. https://doi.org/10.1007/978-3-319-10575-8_21
- Krzysztof R. Apt and Dexter Kozen. 1986. Limits for Automatic Verification of Finite-State Concurrent Systems. *Inf. Process. Lett.* 22, 6 (1986), 307–309. [https://doi.org/10.1016/0020-0190\(86\)90071-2](https://doi.org/10.1016/0020-0190(86)90071-2)
- Mohamed Faouzi Atig, Ahmed Bouajjani, Sebastian Burckhardt, and Madanlal Musuvathi. 2010. On the verification problem for weak memory models. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, Manuel V. Hermenegildo and Jens Palsberg (Eds.). ACM, 7–18. <https://doi.org/10.1145/1706299.1706303>
- Roderick Bloem, Swen Jacobs, Ayrat Khalimov, Igor Konnov, Sasha Rubin, Helmut Veith, and Josef Widder. 2016. Decidability in Parameterized Verification. *SIGACT News* 47, 2 (2016), 53–64. <https://doi.org/10.1145/2951860.2951873>
- Bernard Boigelot, Axel Legay, and Pierre Wolper. 2003. Iterating Transducers in the Large (Extended Abstract). In *Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003, Proceedings (Lecture Notes in Computer Science)*, Warren A. Hunt Jr. and Fabio Somenzi (Eds.), Vol. 2725. Springer, 223–235. https://doi.org/10.1007/978-3-540-45069-6_24
- Ahmed Bouajjani, Egor Derevenetc, and Roland Meyer. 2013. Checking and Enforcing Robustness against TSO. In *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)*, Matthias Felleisen and Philippa Gardner (Eds.), Vol. 7792. Springer, 533–553. https://doi.org/10.1007/978-3-642-37036-6_29
- Ahmed Bouajjani, Peter Habermehl, Adam Rogalewicz, and Tomás Vojnar. 2012. Abstract regular (tree) model checking. *STTT* 14, 2 (2012), 167–191. <https://doi.org/10.1007/s10009-011-0205-y>
- Sebastian Burckhardt. 2014. Principles of Eventual Consistency. *Foundations and Trends in Programming Languages* 1, 1-2 (2014), 1–150.
- Allan Cheng, Javier Esparza, and Jens Palsberg. 1995. Complexity Results for 1-Safe Nets. *Theor. Comput. Sci.* 147, 1&2 (1995), 117–136. [https://doi.org/10.1016/0304-3975\(94\)00231-7](https://doi.org/10.1016/0304-3975(94)00231-7)
- Giorgio Delzanno, Arnaud Sangnier, and Gianluigi Zavattaro. 2010. Parameterized Verification of Ad Hoc Networks, See [Gastin and Laroussinie 2010], 313–327. https://doi.org/10.1007/978-3-642-15375-4_22
- Egor Derevenetc. 2015. *Robustness against Relaxed Memory Models*. Ph.D. Dissertation. University of Kaiserslautern. <http://kluedo.ub.uni-kl.de/frontdoor/index/index/docId/4074>
- Marco Elver and Vijay Nagarajan. 2014. TSO-CC: Consistency directed cache coherence for TSO. In *HPCA 2014*. IEEE, 165–176.

- E. Allen Emerson, John Havlicek, and Richard J. Trefler. 2000. Virtual Symmetry Reduction. In *15th Annual IEEE Symposium on Logic in Computer Science, Santa Barbara, California, USA, June 26-29, 2000*. IEEE Computer Society, 121–131. <https://doi.org/10.1109/LICS.2000.855761>
- E. Allen Emerson and Vineet Kahlon. 2003. Exact and Efficient Verification of Parameterized Cache Coherence Protocols. In *Correct Hardware Design and Verification Methods, 12th IFIP WG 10.5 Advanced Research Working Conference, CHARME 2003, L'Aquila, Italy, October 21-24, 2003, Proceedings (Lecture Notes in Computer Science)*, Daniel Geist and Enrico Tronci (Eds.), Vol. 2860. Springer, 247–262. https://doi.org/10.1007/978-3-540-39724-3_22
- E. Allen Emerson and Vineet Kahlon. 2004. Parameterized Model Checking of Ring-Based Message Passing Systems. In *Computer Science Logic, 18th International Workshop, CSL 2004, 13th Annual Conference of the EACSL, Karpacz, Poland, September 20-24, 2004, Proceedings (Lecture Notes in Computer Science)*, Jerzy Marcinkowski and Andrzej Tarlecki (Eds.), Vol. 3210. Springer, 325–339. https://doi.org/10.1007/978-3-540-30124-0_26
- Javier Esparza, Alain Finkel, and Richard Mayr. 1999. On the Verification of Broadcast Protocols. In *14th Annual IEEE Symposium on Logic in Computer Science, Trento, Italy, July 2-5, 1999*. IEEE Computer Society, 352–359. <https://doi.org/10.1109/LICS.1999.782630>
- Javier Esparza, Pierre Ganty, and Rupak Majumdar. 2016. Parameterized Verification of Asynchronous Shared-Memory Systems. *J. ACM* 63, 1 (2016), 10:1–10:48. <https://doi.org/10.1145/2842603>
- Marie Fortin, Anca Muscholl, and Igor Walukiewicz. 2017. Model-Checking Linear-Time Properties of Parametrized Asynchronous Shared-Memory Pushdown Systems. In *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II (Lecture Notes in Computer Science)*, Rupak Majumdar and Viktor Kuncak (Eds.), Vol. 10427. Springer, 155–175. https://doi.org/10.1007/978-3-319-63390-9_9
- Pierre Ganty and Rupak Majumdar. 2012. Algorithmic verification of asynchronous programs. *ACM Trans. Program. Lang. Syst.* 34, 1 (2012), 6:1–6:48. <https://doi.org/10.1145/2160910.2160915>
- Paul Gastin and François Laroussinie (Eds.). 2010. *CONCUR 2010 - Concurrency Theory, 21th International Conference, CONCUR 2010, Paris, France, August 31-September 3, 2010. Proceedings*. Lecture Notes in Computer Science, Vol. 6269. Springer. <https://doi.org/10.1007/978-3-642-15375-4>
- Steven M. German and A. Prasad Sistla. 1992. Reasoning about Systems with Many Processes. *J. ACM* 39, 3 (1992), 675–735. <https://doi.org/10.1145/146637.146681>
- Matthew Hague. 2011. Parameterised Pushdown Systems with Non-Atomic Writes. In *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2011, December 12-14, 2011, Mumbai, India (LIPIcs)*, Supratik Chakraborty and Amit Kumar (Eds.), Vol. 13. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 457–468. <https://doi.org/10.4230/LIPIcs.FSTTCS.2011.457>
- Vineet Kahlon. 2008. Parameterization as Abstraction: A Tractable Approach to the Dataflow Analysis of Concurrent Programs. In *Proceedings of the Twenty-Third Annual IEEE Symposium on Logic in Computer Science, LICS 2008, 24-27 June 2008, Pittsburgh, PA, USA*. IEEE Computer Society, 181–192. <https://doi.org/10.1109/LICS.2008.37>
- Alexander Kaiser, Daniel Kroening, and Thomas Wahl. 2010. Dynamic Cutoff Detection in Parameterized Concurrent Programs. In *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings (Lecture Notes in Computer Science)*, Tayssir Touili, Byron Cook, and Paul B. Jackson (Eds.), Vol. 6174. Springer, 645–659. https://doi.org/10.1007/978-3-642-14295-6_55
- Yonit Kesten, Oded Maler, Monica Marcus, Amir Pnueli, and Elad Shahar. 2001. Symbolic model checking with rich assertional languages. *Theor. Comput. Sci.* 256, 1-2 (2001), 93–112. [https://doi.org/10.1016/S0304-3975\(00\)00103-1](https://doi.org/10.1016/S0304-3975(00)00103-1)
- Salvatore La Torre, Anca Muscholl, and Igor Walukiewicz. 2015. Safety of Parametrized Asynchronous Shared-Memory Systems is Almost Always Decidable. In *26th International Conference on Concurrency Theory, CONCUR 2015, Madrid, Spain, September 1-4, 2015 (LIPIcs)*, Luca Aceto and David de Frutos-Escrig (Eds.), Vol. 42. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 72–84. <https://doi.org/10.4230/LIPIcs.CONCUR.2015.72>
- Ori Lahav, Nick Giannarakis, and Viktor Vafeiadis. 2016. Taming release-acquire consistency. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, Rastislav Bodik and Rupak Majumdar (Eds.). ACM, 649–662.
- Anca Muscholl, Helmut Seidl, and Igor Walukiewicz. 2017. Reachability for Dynamic Parametric Processes. In *Verification, Model Checking, and Abstract Interpretation - 18th International Conference, VMCAI 2017, Paris, France, January 15-17, 2017, Proceedings (Lecture Notes in Computer Science)*, Ahmed Bouajjani and David Monniaux (Eds.), Vol. 10145. Springer, 424–441. https://doi.org/10.1007/978-3-319-52234-0_23
- Kedar S. Namjoshi and Richard J. Trefler. 2016. Parameterized Compositional Model Checking. In *Tools and Algorithms for the Construction and Analysis of Systems - 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings (Lecture Notes in Computer Science)*, Marsha Chechik and Jean-François Raskin (Eds.), Vol. 9636. Springer, 589–606. https://doi.org/10.1007/978-3-662-49674-9_39

- Scott Owens, Susmit Sarkar, and Peter Sewell. 2009. A Better x86 Memory Model: x86-TSO. In *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings (Lecture Notes in Computer Science)*, Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel (Eds.), Vol. 5674. Springer, 391–407.
- Alberto Ros and Stefanos Kaxiras. 2016. Racer: TSO consistency via race detection. In *49th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2016, Taipei, Taiwan, October 15-19, 2016*. IEEE Computer Society, 33:1–33:13.
- Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. 2011. Understanding POWER multiprocessors. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, Mary W. Hall and David A. Padua (Eds.). ACM, 175–186.
- Walter J. Savitch. 1970. Relationships Between Nondeterministic and Deterministic Tape Complexities. *J. Comput. Syst. Sci.* 4, 2 (1970), 177–192.
- Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. 2010. x86-TSO: a rigorous and usable programmer’s model for x86 multiprocessors. *Commun. ACM* 53, 7 (2010), 89–97.
- D. Weaver and T. Germond (Eds.). 1994. *The SPARC Architecture Manual Version 9*. PTR Prentice Hall.