Verification under TSO with an infinite Data Domain

Parosh Aziz Abdulla[1], Mohamed Faouzi Atig[1], Florian Furbach[2](✉), and Shashwat Garg[3]

[1] Uppsala University, Uppsala, Sweden
[2] Technical University of Denmark, Kongens Lyngby, Denmark
fwafu@dtu.dk
[3] Indian Institute of Technology Bombay, Mumbai, India

Abstract. We examine verification of concurrent programs under the total store ordering (TSO) semantics used by the x86 architecture. In our model, threads manipulate variables over infinite domains and they can check whether variables are related for a range of relations. We show that, in general, the control state reachability problem is undecidable. This result is derived through a reduction from the state reachability problem of lossy channel systems with data (which is known to be undecidable). In the light of this undecidability, we turn our attention to a more tractable variant of the reachability problem. Specifically, we study context bounded runs, which provide an under-approximation of the program behavior by limiting the possible interactions between processes. A run consists of a number of contexts, with each context representing a sequence of steps where a only single designated thread is active. We prove that the control state reachability problem under bounded context switching is PSPACE complete.

1 Introduction

Over the years, research on concurrent verification has been chiefly conducted under the premise that the threads run according to the classical Sequential Consistency (SC) semantics. Under SC, the threads operate on a set of shared variables through which they communicate *atomically*, i.e., read and write operations take effect immediately. In particular, a write operation is visible to all the threads as soon as the writer thread carries out its operation. Therefore, the threads always maintain a uniform view of the shared memory: they all see the latest value written on any given variable and we can interpret program runs as interleavings of sequential thread executions. Although SC has been immensely popular as an intuitive way of understanding the behaviours of concurrent threads, it is not realistic to assume computation platforms guarantee SC anymore. The reason is that, due to hardware and compiler optimizations, most modern platforms allow more relaxed program behaviours than those permitted 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., [33,32]), Cache protocols (e.g., [31,19]), language level concurrency (e.g., [24]), and distributed data stores (e.g., [17]). Program behaviours change dramatically when moving from the SC semantics to weaker semantics. Therefore, in recent years, research on the verification of concurrent programs under weak memory models have started to become popular. A classical example of weak memory models is the Total Store Ordering (TSO) semantics which is a formalization of the Intel x86 processor architecture [29]. The TSO semantics inserts an unbounded FIFO buffer, called the *store buffer*, between each thread and the main memory. When a thread performs a write instruction, the corresponding operation is appended to the end of the buffer, and hence it is not immediately visible to other threads. The write messages are non-deterministically propagated from the store buffer of a given thread to the shared memory. Verification of programs that contain data races needs to take the underlying memory model into account. This is crucial in hardware-close programming, especially in concurrent libraries or kernels. Such applications are inherently racy; exploiting racy WMM operations for efficiency is standard practice. Our work serves as a foundation for ensuring the correctness of such systems, which often rely on these intricate memory models to achieve optimal performance.

In a parallel development, significant research has been done on extending model checking frameworks to programs with infinite state spaces. There are two main reasons why a program might have an infinite state space. The first is that the program has unbounded control structures, which means it can have an unbounded number of threads. Examples include parameterized systems, in which correctness of the system is checked regardless of the number of threads, and programs that allow dynamic thread creation through spawning [11]. Secondly, the program may operate on unbounded data structures, such as clocks [12], stacks [16], and queues ([10,1]). These works, including their extensions, have been done under the SC assumption. Although recent works have started to explore parameterized verification for weak memory models [6,4,22], the verification of programs that operate on a shared unbounded data structure with weak memory semantics has remained unexplored until now.

In this paper, we combine infinite-state programs with weak memory models: we study the decidability and complexity of the reachability problem for programs operating on unbounded data structures under the TSO semantics. While the TSO semantics has been extensively studied (e.g., [15,5]), it has been assumed that the data domain is finite. This means that the possible values of a shared variable or a register are bounded. In contrast, our model allows for an infinite domain such as natural numbers $\mathbb{N}$ or real numbers $\mathbb{R}$. It contains register assignments, an operator that may assign an arbitrary value to a register, and a set of relations that act as guards. We focus on relations equality and "greater than" on totally ordered sets and combinations, negations and inversions of them. Our model finds practical utility in continuously running concurrent protocols. A prime example is the bakery ticket protocol used in various scenarios. It is presented in Section 4. Here, an unbounded number of requests occur, each assigned increasing numbers and the lowest-numbered request is serviced. This presents a scenario with inherent races that requires an infinite domain which our model can effectively capture. Note that our model is infinite in multiple dimensions: the threads are infinite-state as they operate on unbounded data domains, the store buffers are unbounded, and they carry write-messages over an unbounded domain.

In order to perform safety verification, we need to decide whether there is an execution that can reach some undesirable control state. We study the control state reachability problem and show that for many domains and relations, it is undecidable. Therefore, we propose an alternative approach by introducing an under-approximation schema using context-bounding [30,28,25,23,14]. Context-bounding has been proposed in [30] as a suitable approach for efficient bug detection in multithreaded programs. Indeed, for concurrent programs, a bounding concept that provides both good coverage and scalability must be based on aspects related to the interactions between concurrent components. It has been shown experimentally that concurrency bugs usually show up after a small number of context switches [28]. In this work, we study a context bounded analysis where only the active thread may perform an operation and update the memory. We show that in this case, the state reachability problem is not only decidable, but even PSPACE complete. To this end, we perform a two-step abstraction that employs insights about context bounded runs of TSO semantics as well as the structure of reachable configurations.

In the first step of our abstraction process, we refine the methods introduced by [14]. Their construction introduces a code-to-code translation that abstracts the buffer, simplifying the problem to state reachability under SC. Our approach leverages the fact that this abstraction does not explicitly depend on variable values. In our case, the abstraction step yields a register machine where the register values are integers or real numbers, and the transitions are conditioned by “gap-constraints" [9,18,27]. Gap constraints serve to identify, within each system configuration, (i) the variables with identical values and (ii) the gaps (differences) between variable values. Notably, these gaps can be arbitrarily large. The papers [9,18,27] analyze programs with gap constraints within the framework of well-structured systems [8,20]. As a result, they do not provide upper bounds on the complexity.

As another key contribution of this paper, we propose a method to achieve PSPACE completeness. The fundamental idea behind our algorithm is that for any system execution, there is an alternative execution with larger gaps among the variables. This implies that we do not need to explicitly track the gaps between variables, as is the case in [9,18,27]. Instead, we implement a second (precise) abstraction step, focusing solely on the order of variables. For any pair of variables x and y, we record whether $x = y$, $x < y$, or $x > y$.

2 Related Work

Not much current work considers the complexity and decidability of infinite-state state programs on weak memory models. Furthermore, most existing works consider parameterized verification rather than programs with infinite data domains. The paper [6] considers parameterized verification of programs running under TSO, and shows that the reachability problem is PSPACE complete. However, the work assumes that the threads are finite-state and, in particular, the threads do not manipulate unbounded data domains. The paper [22] shows PSPACE completeness when the underlying semantics is the Release-Acquire fragment of C11. The latter semantics gives rise to a different semantics compared to TSO. The paper also considers finite-state threads.

In [2], parameterized verification of programs running under TSO is considered. However, the paper applies the framework of well-structured systems where the buffers of the threads 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 paper [15] 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 this work cannot be applied in our setting.

The paper [4] considers parameterized verification under the TSO semantics when the individual threads are infinite-state. However, the authors study a *restricted* model, where it assumes that (i) all threads are identical and (ii) the threads do not use atomic operations. Generally, parameterized verification for the restricted model is easier than non-parameterized verification. For instance, in the case of TSO where the threads are finite-state, the restricted parameterized verification problem is in PSPACE [6] while the non-parameterized problem has a non-primitive recursive complexity [13].

The are many works on extending infinite-state systems with unbounded data domains. Well studied examples are Petri nets with data tokens [27], stacks with unbounded stack alphabets [7], and lossy channel systems with unbounded message alphabets [1]. All these works assume the SC semantics and are hence orthogonal to this work.

3 Total Store Order (TSO)

Let $\mathbb{B} = \{true, false\}$. Given a function $f : A \to B$ with $a \in A, b \in B$, $f[a \leftarrow b]$ is defined as follows: $f[a \leftarrow b](a) := b$, $f[a \leftarrow b](a') := f(a')$ for any $a' \in A$ with $a' \neq a$. We write $x \in w$ for letter $x \in \Sigma$ occurring in word $w \in \Sigma^*$ and $w' \leq w$ for $w' \in \Sigma^*$ being a subsequence of w.

Let x and y be two natural (real) numbers. Let $n \in \mathbb{N}$, we use $x <_n y$ (resp. $\leq_n y$) to denote that $x + n < y$ (resp. $x + n \leq y$). A data theory is defined by a pair $(\mathtt{D}, \mathsf{Rl})$ where $\mathtt{D}$ is an infinite data domain and $\mathsf{Rl} \subseteq \mathtt{D} \times \mathtt{D} \to \mathbb{B}$ is a finite set of relations over $\mathtt{D}$. In this paper, we restrict ourselves to the set of natural/real numbers as data domain, and the set of relations Rl to be a subset of $\mathsf{Rl}_{\leq n} = \{=, \neq, <, \leq, <_n, \leq_n | \ n \in \mathbb{N}\}$. We assume w.l.o.g. that $0 \in \mathtt{D}$.

Transition Systems A labelled transition system is a tuple $\mathcal{TS} = (\Gamma, \mathcal{L}, \mathcal{T}, \gamma_{\mathsf{init}})$ that consists of a set of *configurations* Γ, a finite set of labels $\mathcal{L}$, a labelled transition relation $\mathcal{T} \subseteq \Gamma \times \mathcal{L} \times \Gamma$, and an initial configuration $\gamma_{\mathsf{init}} \in \Gamma$. We write $\gamma \xrightarrow{\ell} \gamma'$ for $\langle \gamma, \ell, \gamma' \rangle \in \mathcal{T}$. We say that $\pi = t_1 \dots t_n \in \mathcal{T}^*$ is a run of $\mathcal{TS}$ if there is a sequence of configurations $\gamma_1, \gamma_2, \dots, \gamma_{n+1}$ such that $t_i = \gamma_i \xrightarrow{\ell_i} \gamma_{i+1}$ for $i \leq n$ and $\gamma_1 = \gamma_{\mathsf{init}}$. The run π ends in configuration γ_{n+1}. We say that γ is reachable if there is a run π of $\mathcal{TS}$ that ends in γ.

Programs A concurrent program Prog consists of finite set of threads $\mathcal{T}$. Each thread $t \in \mathcal{T}$ is a finite state machine that works on its own set of local registers $\mathcal{R}_t$. The local registers of different threads are disjoint. Let $\mathcal{R} = \cup_{t \in \mathcal{T}} \mathcal{R}_t$. The threads communicate over a finite set of shared variables $\mathcal{X}$. The registers and the shared variables take their values from a data theory $(\mathtt{D}, \mathsf{Rl})$. Formally, a thread is a tuple $t = \langle \mathcal{Q}_t, \mathcal{R}_t, \Delta_t, q^t_{\mathsf{init}} \rangle$ where $\mathcal{Q}_t$ is a finite set of states of thread t, $q^t_{\mathsf{init}} \in \mathcal{Q}_t$ is the initial state of t, and $\Delta_t \subseteq \mathcal{Q}_t \times \mathsf{Op} \times \mathcal{Q}_t$ is a finite set of transitions that change the state and execute an operation $\mathsf{op} \in \mathsf{Op}$. Let $x \in \mathcal{X}, r_1, r_2 \in \mathcal{R}_t$. A transition $\delta \in \Delta_t$ is a tuple $\delta = \langle q, \mathsf{op}, q' \rangle$ where the operation $\mathsf{op} \in \mathsf{Op}$ has one of the following forms: (1) $r_1 := r_2$ assigns the value of register r_2 to register r_1, (2) $r_1 := \circledast$ non-deterministically assigns a value to register r_1, (3) $\mathsf{rl}(r_1, r_2)$ checks if the values of the two registers r_1 and r_2 satisfy the relation $\mathsf{rl} \in \mathsf{Rl}$, (4) $\mathsf{rd}(x, r_1)$ reads the value of shared variable x and stores it in register r_1, (5) $\mathsf{wt}(x, r_1)$ writes the value of register r_1 to shared variable x, and (6) $\mathsf{arw}(x, r_1, r_2)$ is the atomic read write operation which atomically executes a read followed by a write operation.

TSO Semantics The TSO memory model [33] is used by the x86 processor architecture. Each thread has its own FIFO write buffer. Write operations $\mathsf{wt}(x, r)$ in a thread t do not update the memory immediately; if $d \in \mathtt{D}$ is the value of r, then (x, d) is appended to the buffer of t. The buffer contents are updated to the shared memory non-deterministically. A read operation $\mathsf{rd}(x, r)$ in t accesses the latest write in the buffer of t. In case there is no such write, it accesses the shared memory. For the atomic read write operation $\mathsf{arw}(x, r_1, r_2)$ in thread t, the buffer of t must be empty (ϵ), and the value of x in the memory must be same as the value of r_1. Then x is set to the value of r_2.

Formally, the TSO memory model is a labelled transition system. A configuration γ is defined as a tuple $\gamma = \langle \mathsf{St}, \mathsf{RVal}, \mathsf{Buf}, \mathsf{Mem} \rangle$ where $\mathsf{St} : \mathcal{T} \to \bigcup_{t \in \mathcal{T}} \mathcal{Q}_t$ maps each thread to its current state, $\mathsf{RVal} : \mathcal{R} \to \mathtt{D}$ maps each register in a thread to its current value, $\mathsf{Buf} : \mathcal{T} \to (\mathcal{X} \times \mathtt{D})^*$ maps each thread buffer to its content, which is a sequence of writes. Finally, $\mathsf{Mem} : \mathcal{X} \to \mathtt{D}$ maps each shared variable to its current value in the memory. The initial configuration of Prog is defined by a tuple $\gamma_{\mathsf{init}} = \langle \mathsf{St}_{\mathsf{init}}, \mathsf{RVal}_{\mathsf{init}}, \mathsf{Buf}_{\mathsf{init}}, \mathsf{Mem}_{\mathsf{init}} \rangle$ where $\mathsf{St}_{\mathsf{init}}$ maps each thread t to its initial states q^t_{init}, $\mathsf{RVal}_{\mathsf{init}}$ and $\mathsf{Mem}_{\mathsf{init}}$ assign all registers

$$\frac{\langle q, r_1 := r_2, q'\rangle \in \Delta_t}{\langle \mathsf{St}, \mathsf{RVal}, \mathsf{Buf}, \mathsf{Mem}\rangle \xrightarrow{t, r_1 := r_2} \langle \mathsf{St}[t \leftarrow q'], \mathsf{RVal}[r_1 \leftarrow \mathsf{RVal}(r_2)], \mathsf{Buf}, \mathsf{Mem}\rangle} \text{ assign}$$

$$\frac{\langle q, r_1 := \circledast, q'\rangle \in \Delta_t \quad d \in \mathtt{D}}{\langle \mathsf{St}, \mathsf{RVal}, \mathsf{Buf}, \mathsf{Mem}\rangle \xrightarrow{t, r_1 := \circledast} \langle \mathsf{St}[t \leftarrow q'], \mathsf{RVal}[r_1 \leftarrow d], \mathsf{Buf}, \mathsf{Mem}\rangle} \text{ new value}$$

$$\frac{\langle q, \mathsf{rl}(r_1, r_2), q'\rangle \in \Delta_t \quad \mathsf{rl}(R(r_1), R(r_2))}{\langle \mathsf{St}, \mathsf{RVal}, \mathsf{Buf}, \mathsf{Mem}\rangle \xrightarrow{t, \mathsf{rl}(r_1, r_2)} \langle \mathsf{St}[t \leftarrow q'], \mathsf{RVal}, \mathsf{Buf}, \mathsf{Mem}\rangle} \text{ relation}$$

$$\frac{\langle q, \mathsf{wt}(x, r_1), q'\rangle \in \Delta_t}{\langle \mathsf{St}, \mathsf{RVal}, \mathsf{Buf}, \mathsf{Mem}\rangle \xrightarrow{t, \mathsf{wt}(x, r_1)} \langle \mathsf{St}[t \leftarrow q'], \mathsf{RVal}, \mathsf{Buf}[t \leftarrow (x, \mathsf{RVal}(r_1)).\mathsf{Buf}(t)], \mathsf{Mem}\rangle} \text{ write}$$

$$\frac{\langle q, \mathsf{rd}(x, r_1), q'\rangle \in \Delta_t \quad \nexists d \in \mathtt{D} : (x, d) \in \mathsf{Buf}(t)}{\langle \mathsf{St}, \mathsf{RVal}, \mathsf{Buf}, \mathsf{Mem}\rangle \xrightarrow{t, \mathsf{rd}(x, r_1)} \langle \mathsf{St}[t \leftarrow q'], \mathsf{RVal}[r_1 \leftarrow \mathsf{Mem}(x)], \mathsf{Buf}, \mathsf{Mem}\rangle} \text{ global read}$$

$$\frac{\langle q, \mathsf{rd}(x, r_1), q'\rangle \in \Delta_t \quad \mathsf{Buf}(t) = \alpha.(x, d).\beta \quad \alpha, \beta \in (\mathcal{X} \cdot \mathtt{D})^* \quad \nexists d' \in \mathtt{D} : (x, d') \in \alpha}{\langle \mathsf{St}, \mathsf{RVal}, \mathsf{Buf}, \mathsf{Mem}\rangle \xrightarrow{t, \mathsf{rd}(x, r_1)} \langle \mathsf{St}[t \leftarrow q'], \mathsf{RVal}[r_1 \leftarrow d], \mathsf{Buf}, \mathsf{Mem}\rangle} \text{ local read}$$

$$\frac{\langle q, \mathsf{arw}(x, r_1, r_2), q'\rangle \in \Delta_t \quad \mathsf{Buf}(t) = \epsilon \quad \mathsf{RVal}(r_1) = \mathsf{Mem}(x)}{\langle \mathsf{St}, \mathsf{RVal}, \mathsf{Buf}, \mathsf{Mem}\rangle \xrightarrow{t, \mathsf{arw}(x, r_1, r_2)} \langle \mathsf{St}[t \leftarrow q'], \mathsf{RVal}, \mathsf{Buf}, \mathsf{Mem}[x \leftarrow \mathsf{RVal}(r_2)]\rangle} \text{ atomic read write}$$

$$\frac{}{\langle \mathsf{St}, \mathsf{RVal}, \mathsf{Buf}[t \leftarrow \mathsf{Buf}(t).(x, d)], \mathsf{Mem}\rangle \xrightarrow{t, u} \langle \mathsf{St}, \mathsf{RVal}, \mathsf{Buf}, \mathsf{Mem}[x \leftarrow d]\rangle} \text{ memory update}$$

Fig. 1. The transition relation of TSO. We assume that $\mathsf{St}(t) = q$.

and shared variables the value 0, and $\mathsf{Buf}_{\mathsf{init}}$ initializes all thread buffers to the empty word ϵ. We formally define the labelled transition relation $\xrightarrow{\ell}$ on configurations in Figure 1 where the label ℓ is either of the form t, op (to denote a thread operation) or t, u (to denote an update operation) with $t \in \mathcal{T}$ is a thread and $\mathsf{op} \in \mathsf{Op}$ is an operation.

The Reachability Problem ***Reach*** Given a concurrent program Progand a state $q_{final} \in \mathcal{Q}_t$ of thread t, Reach asks, if a configuration $\gamma = \langle \mathsf{St}, \mathsf{RVal}, \mathsf{Buf}, \mathsf{Mem}\rangle$ with $\mathsf{St}(t) = q_{final}$ is reachable by the transition system given by the TSO semantics of Prog. In this case, we say that the state q_{final} is reachable by Prog. We use Reach[D, Rl] to denote the reachability problem for a concurrent program with the data theory (D, Rl).

4 Lamport's Bakery Algorithm

To demonstrate the practical application of our model, we use it to model Lamport's Bakery Algorithm [26]. Created by Leslie Lamport in 1974, it is a cornerstone solution for achieving mutual exclusion in concurrent systems. Picture threads as patrons entering a bakery, each is handed a unique ticket upon arrival. These tickets, representing the order of entry, dictate the sequence for accessing critical sections. They ensure an orderly execution flow and preventing race conditions in a critical section.

Each thread is assigned a unique number that is larger then the numbers currently assigned to other threads. The thread possessing the lowest number is granted entry to the critical section. This thread may access the critical section an unbounded number of times. This means the assigned tickets keep increasing and thus an infinite domain is required. Note that the algorithm does not rely on precise tickets values, we only need to compare the tickets to each other. This makes the protocol well suited to our program model.

The protocol contains n threads where each thread $i \leq n$ is associated with two variables: The ticket number $ticket_i$ and the flag $chosen_i$ which signals whether the thread has chosen a ticket number. We assume r_{TRUE} and r_{FALSE} are initialized with different values that represent the boolean values of a flag and that $ticket_i$ is initially the same as r_{FALSE} for all $i \leq n$.

The algorithm for thread i is given in Algorithm 1. For the sake of simplicity and compactness we present the transition system as pseudocode. This is equivalent to a program definition since the code only accesses variables and registers using operations Op with relations $\mathsf{Rl}_<$. The remaining instructions only affect the finite control flow and can be expressed using transitions.

Algorithm 1 Lamport Bakery Protocol

```
 1: wt($chosen_i, r_{FALSE}$) {Begin choosing}
 2: $r_i := \circledast$ {Pick random ticket}
 3: for all $1 \leq j \leq n$ do
 4:    rd($ticket_j, r_j$)
 5:    if ($r_i < r_j$) then
 6:       goto line 1 {New ticket needed.}
 7:    end if
 8: end for
 9: wt($ticket_i, r_i$) {Ticket accepted}
10: wt($chosen_i, r_{TRUE}$) {Choosing finished}
11: for all $1 \leq j \leq n$ do
12:    rd($chosen_j, r_j$)
13:    if ($r_j \neq r_{TRUE}$) then
14:       goto line 12 {Thread $j$ is still choosing}
15:    end if
16:    rd($ticket_j, r_j$)
17:    if ($r_j \neq r_{FALSE}$ & $r_j < r_i$) then
18:       goto line 16 {Lower ticket $j$ found}
19:    end if
20: end for
21: CRITICAL Section
22: $r_i := r_{FALSE}$
23: goto line 1 {Back to NON-CRITICAL}
```

5 State Reachability for TSO with (Dis)-Equality Relation

We show that the reachability problem for concurrent programs under TSO is undecidable when $\{=, \neq\} \subseteq \mathsf{Rl}$. The proof is achieved through a reduction from the state reachability problem of Lossy Channel Systems with Data (DLCS) [1], which is already known to be undecidable. To simulate the lossy channel, we employ write buffers, as both are implemented as first-in-first-out queues. However, there are three main distinctions that must be considered: (i) write buffers do not contain letters, (ii) write buffers are not lossy, and (iii) the semantics of reads differ from receives.

We address these distinctions as follows: (i) We encode the letters as variables. (ii) We model writes being lost by avoiding to read them. (iii) To prevent buffer reads, we transfer the writes into a write buffer of a second thread with a different variable. We ensure that every write is accessed only once by overwriting them immediately with a different value.

Theorem 1. *Reach*$[\mathtt{D}, \mathsf{Rl}]$ *is undecidable for* $\{=, \neq\} \subseteq \mathsf{Rl}$.

The rest of this section is devoted to the proof of the above theorem. We first recall the definition of Lossy Channel Systems with Data (DLCS) [1]. Then, we present the reduction from state reachability problem of DLCS to $\mathtt{Reach}[\mathtt{D}, \mathsf{Rl}]$.

$$\frac{\langle q, x := y, q' \rangle \in \Delta_{\mathcal{L}}}{\langle q, \mathsf{XVal}, w \rangle \xrightarrow{x:=y} \langle q', \mathsf{XVal}[x \leftarrow \mathsf{XVal}(y)], w \rangle}\ \text{assign}$$

$$\frac{\langle q, x := \circledast, q' \rangle \in \Delta_{\mathcal{L}} \quad d \in \mathtt{D} \setminus \{\mathsf{XVal}(y) \mid y \in \mathcal{X}_{\mathcal{L}}\}}{\langle q, \mathsf{XVal}, w \rangle \xrightarrow{x:=\circledast} \langle q', \mathsf{XVal}[x \leftarrow d], w \rangle}\ \text{new value}$$

$$\frac{\langle q, x = y, q' \rangle \in \Delta_{\mathcal{L}} \quad \mathsf{XVal}(x) = \mathsf{XVal}(y)}{\langle q, \mathsf{XVal}, w \rangle \xrightarrow{x=y} \langle q', \mathsf{XVal}, w \rangle}\ \text{equality}$$

$$\frac{\langle q, x \neq y, q' \rangle \in \Delta_{\mathcal{L}} \quad \mathsf{XVal}(x) \neq \mathsf{XVal}(y)}{\langle q, \mathsf{XVal}, w \rangle \xrightarrow{x\neq y} \langle q', \mathsf{XVal}, w \rangle}\ \text{disequality}$$

$$\frac{\langle q, !\langle a, x \rangle, q' \rangle \in \Delta_{\mathcal{L}}}{\langle q, \mathsf{XVal}, w \rangle \xrightarrow{!\langle a,x \rangle} \langle q', \mathsf{XVal}, (a, \mathsf{XVal}(x)).w \rangle}\ \text{send}$$

$$\frac{\langle q, ?\langle a, x \rangle, q' \rangle \in \Delta_{\mathcal{L}}}{\langle q, \mathsf{XVal}, w.(a, d) \rangle \xrightarrow{?\langle a,x \rangle} \langle q', \mathsf{XVal}[x \leftarrow d], w \rangle}\ \text{receive}$$

$$\frac{w' \leq w}{\langle q, \mathsf{XVal}, w \rangle \xrightarrow{loss} \langle q, \mathsf{XVal}, w' \rangle}\ \text{lossiness}$$

Fig. 2. The transition relation of DLCS

Lossy Channel Systems with Data A DLCS $\mathcal{L} = \langle \mathcal{Q}_\mathcal{L}, \mathcal{X}_\mathcal{L}, \Sigma_\mathcal{L}, \Delta_\mathcal{L}, q_{\mathsf{init}} \rangle$ consists of a finite set of states $\mathcal{Q}_\mathcal{L}$, a finite number of variables $\mathcal{X}_\mathcal{L}$ ranging over an infinite domain $\mathtt{D}$, a finite channel alphabet $\Sigma_\mathcal{L}$, $q_{\mathsf{init}} \in \mathcal{Q}$ is the initial state, and a finite set of transitions $\Delta_\mathcal{L}$. The set $\Delta_\mathcal{L}$ of transitions is a subset of $\mathcal{Q}_\mathcal{L} \times \mathsf{Op}_\mathcal{L} \times \mathcal{Q}_\mathcal{L}$. Let $x, y \in \mathcal{X}_\mathcal{L}$. The set $\mathsf{Op}_\mathcal{L}$ consists of the following operations (1) $x := y$ which assigns the value of y to x, (2) $x := \circledast$, which assigns a fresh value from $\mathtt{D}$ that is different from the existing values of all variables[4], (3) $x = y$ ($x \neq y$) which compares the value of variables x and y, (4) $!\langle a, x\rangle$ which appends letter $a \in \Sigma_\mathcal{L}$ together with the value of x to the channel, (5) $?\langle a, x\rangle$ which deletes the head of the channel $\langle a, d\rangle$ and stores the value d in x, and (6) *loss* which removes elements in the channel.

A configuration γ of DLCS is defined by the tuple $\langle q, \mathsf{XVal}, w\rangle$ where $q \in \mathcal{Q}_\mathcal{L}$ is the current state, $\mathsf{XVal} : \mathcal{X}_\mathcal{L} \to \mathtt{D}$ is the current valuation of the variables, and $w \in (\Sigma \times \mathtt{D})^*$ is the content of the lossy channel. The system is lossy, which means any element in the channel may disappear anytime. The initial configuration γ_{init} of $\mathcal{L}$ is defined by $(q_{\mathsf{init}}, \mathsf{XVal}_{\mathsf{init}}, \epsilon)$ where $\mathsf{XVal}_{\mathsf{init}}(x) = 0$ for all $x \in \mathcal{X}_\mathcal{L}$. The transition relation of DLCS is given in Figure 2.

The state reachability problem for $\mathcal{L}$ asks whether, for a given final state $q_{\mathit{final}} \in \mathcal{Q}$, there is a reachable configuration γ of the form $\gamma = \langle q_{\mathit{final}}, \mathsf{XVal}, w\rangle$. In this case, we say that the state q_{final} is reachable by $\mathcal{L}$.

Theorem 2 ([1]). *The state reachability problem for DLCS is undecidable.*

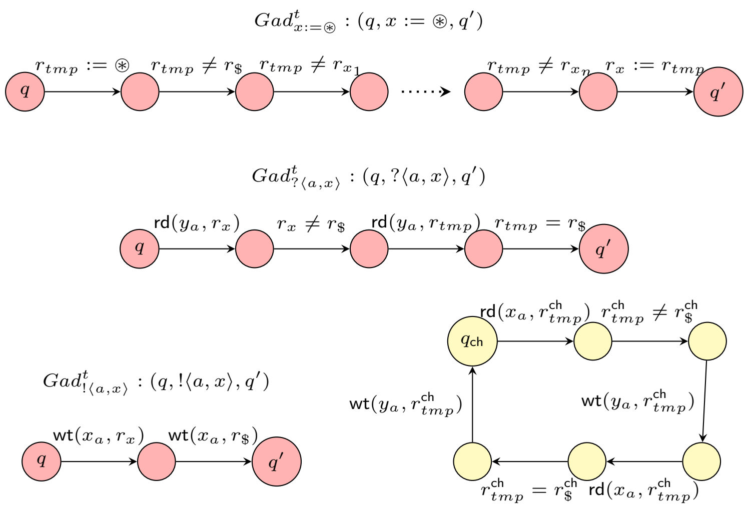
$Gad^t_{x:=\circledast} : (q, x := \circledast, q')$
$r_{tmp} := \circledast$
$r_{tmp} \neq r_\$$
$r_{tmp} \neq r_{x_1}$
$r_{tmp} \neq r_{x_n}$
$r_x := r_{tmp}$
q
q'
$Gad^t_{?\langle a,x\rangle} : (q, ?\langle a, x\rangle, q')$
$\mathsf{rd}(y_a, r_x)$
$r_x \neq r_\$$
$\mathsf{rd}(y_a, r_{tmp})$
$r_{tmp} = r_\$$
$Gad^t_{!\langle a,x\rangle} : (q, !\langle a, x\rangle, q')$
$\mathsf{wt}(x_a, r_x)$
$\mathsf{wt}(x_a, r_\$)$
q_{ch}
$\mathsf{rd}(x_a, r^{\mathsf{ch}}_{tmp})$
$r^{\mathsf{ch}}_{tmp} \neq r^{\mathsf{ch}}_\$$
$\mathsf{wt}(y_a, r^{\mathsf{ch}}_{tmp})$
$\mathsf{wt}(y_a, r^{\mathsf{ch}}_{tmp})$
$r^{\mathsf{ch}}_{tmp} = r^{\mathsf{ch}}_\$$
$\mathsf{rd}(x_a, r^{\mathsf{ch}}_{tmp})$

Fig. 3. $\mathsf{Prog}(\mathcal{L})$ with threads t (pink states) and t_{ch} (yellow states).

[4] This differs from the $\circledast$ in TSO where the value $d \in \mathtt{D}$ assigned by the operation $x := \circledast$ can be anything.

Reduction from DLCS reachability Given a DLCS $\mathcal{L} = \langle \mathcal{Q}_{\mathcal{L}}, \mathcal{X}_{\mathcal{L}}, \Sigma_{\mathcal{L}}, \Delta_{\mathcal{L}}, q_{\mathsf{init}} \rangle$ over data domain $\mathtt{D}$ with $\mathcal{X}_{\mathcal{L}} = \{x_1 \ldots x_n\}$, we reduce the state reachability of $\mathcal{L}$ to the reachability problem $\mathtt{Reach}[\mathtt{D}, \{=, \neq\}]$ of a concurrent program $\mathsf{Prog}(\mathcal{L})$, with two threads t, t_{ch}. The thread t simulates the operations of $\mathcal{L}$, while thread t_{ch} simulates the lossy channel of $\mathcal{L}$ using its write buffer. Let $\mathcal{R}_t = \{r_{\$}, r_{tmp}\} \cup \{r_x \mid x \in \mathcal{X}_{\mathcal{L}}\}$, $\mathcal{R}_{t_{\mathsf{ch}}} = \{r^{\mathsf{ch}}_{\$}, r^{\mathsf{ch}}_{tmp}\}$ be the local registers of threads t and t_{ch}. Corresponding to each $x \in \mathcal{X}_{\mathcal{L}}$, we have the register r_x in thread t, which stores the current values of x. Registers r_{tmp} and r^{ch}_{tmp} are used to temporarily store certain values. The shared variables of $\mathsf{Prog}(\mathcal{L})$ are $\mathcal{X} = \{x_a, y_a \mid a \in \Sigma_{\mathcal{L}}\}$, they help in simulating the behavior of the lossy channel of $\mathcal{L}$.

Simulating the DLCS. The transitions of $\mathsf{Prog}(\mathcal{L})$ are sketched in Figure 3. The initialization of the program is omitted in the figure and goes as follows. The thread t_{ch} starts by assigning a non-deterministic value (say \$) to the register $r^{\mathsf{ch}}_{\$}$ (i.e., $r^{\mathsf{ch}}_{\$} := \circledast$), then checks that the new value \$ is different from 0 (i.e., by checking that $r^{\mathsf{ch}}_{\$} \neq r^{\mathsf{ch}}_{tmp}$), and finally performs an atomic read write operation $\mathsf{arw}(x, r^{\mathsf{ch}}_{tmp}, r^{\mathsf{ch}}_{\$})$ on each variable $x \in \mathcal{X}$. The thread t starts by reading the value of each shared variable $x \in \mathcal{X}$ (i.e., performing $\mathsf{rd}(x, r_{\$})$) and checks if its value is different from 0 (i.e., $r_{\$} \neq r_{tmp}$). At the end of this initialization phase, all the shared variables have the new value \$, the registers r_{tmp} and r^{ch}_{tmp} have the value 0 and the registers $r_{\$}$ and $r^{\mathsf{ch}}_{\$}$ have the value \$. The current state of thread t is the initial state q_{init} of $\mathcal{L}$ while the thread t_{ch} is in a state q_{ch}.

Every transition $\langle q, x := y, q' \rangle \in \Delta_{\mathcal{L}}$ is simulated in $\mathsf{Prog}(\mathcal{L})$ by threat t with a gadget—a sequence of transitions that starts in q and ends in q'. The transitions $(q, x := y, q')$, $(q, x = y, q')$ and $(q, x \neq y, q')$ in the DLCS are simulated by the thread t as gadgets with single transitions $(q, r_x := r_y, q'), (q, r_x = r_y, q')$ and $(q, r_x \neq r_y, q')$, respectively. We omit their description in Figure 3.

To simulate $x := \circledast$, we load the new value in register r_{tmp} and ensure that it is different from the values in registers $r_{\$}$ and $r_{x_1} \ldots r_{x_n}$. This is depicted by the gadget $Gad^t_{x:=\circledast}$ in thread t. The send operation $!\langle a, x \rangle$ in the DLCS is simulated by the gadget $Gad^t_{!\langle a,x \rangle}$. In the DLCS, the send appends the letter a and the value of x to the channel. This is simulated by the write $\mathsf{wt}(x_a, r_x)$, thereby appending $(x_a, val(r_x))$ to the buffer of t. To simulate reads of the DLCS, we first make note of a crucial difference in the way reads happen in DLCS and TSO. In DLCS, a read happens from the head of the channel, and the head is deleted immediately after the read. In TSO however, we can read from the latest write in the shared memory multiple times. In order to simulate the "read once" policy of the DLCS, we follow each $\mathsf{wt}(x_a, r_x)$ with another write $\mathsf{wt}(x_a, r_{\$})$.

Thread t_{ch} is a loop from the state q_{ch} which continuously reads from x_a a value from a simulated send followed by the separator \$. It copies these values to y_a using local register r^{ch}_{tmp}. The first time it reads from x_a, it reads the value d of x from a simulated send $!\langle a, x \rangle$. It ensures that this is not the \$ symbol ($r^{\mathsf{ch}}_{tmp} \neq r^{\mathsf{ch}}_{\$}$), and writes this value from r^{ch}_{tmp} into variable y_a, thus appending (y_a, d) in the buffer of t_{ch}. It then reads again the value of x_a into r^{ch}_{tmp}. This time, it makes sure to read \$ with the check $r^{\mathsf{ch}}_{tmp} = r^{\mathsf{ch}}_{\$}$. The receive $?\langle a, x \rangle$ of the DLCS is simulated by $Gad^t_{?\langle a,x \rangle}$. First, we read from y_a and store it in r_x, ensuring this value d is not \$. Then, we read \$ from y_a. This ensures that the earlier value d is overwritten in the memory and is not read twice.

A loss in the channel of the DLCS results in losing some messages $\langle a, d\rangle$. This is accounted for in $\mathsf{Prog}_{\mathcal{L}}$ in two ways. Thread t_{ch} may not pass on a value written from x_a to y_a since the loop may not execute for every value. Thread t may not read a value written by t_{ch} in y_a since it was already overwritten by some later writes.

Lemma 1. *The state q_{final} is reachable by $\mathcal{L}$ if and only if q_{final} is reachable by* $\mathsf{Prog}(\mathcal{L})$*.*

The formal proof is given in Appendix A of the full version [3]. Theorem 1 extends to any set of relations that we can use to simulate equality and disequality. For instance $\leq, \not\leq \in \mathsf{Rl}$.

6 Context Bounded Analysis

In the light of this undecidability, we turn our attention to a variant of the reachability problem which is tractable. We study context bounded runs, an under-approximation of the program behavior that limits the possible interactions between processes. A run consists of a number of *contexts*. A context is a sequence of steps where only a certain fixed thread t is *active*. We say that $\pi \in \mathsf{CB}(k)$ if and only if there is a partitioning $\pi = \pi_1 \ldots \pi_k$ such that for all contexts $i \leq k$ there is an active thread $t_i \in \mathcal{T}$ such that only the active thread updates the memory and performs operations: If $\gamma \xrightarrow{\ell} \gamma' \in \pi_i$, then $\ell \in \{t_i\} \times (\mathsf{Op} \cup \{u\})$.

In the following, we show PSPACE completeness of $\mathsf{CB}(k)$-$\mathtt{Reach}[\mathtt{D}, \mathsf{Rl}_{\leq n}]$ for relations such as (dis) equality, "greater than" or even "greater by at least n" for $n \in \mathbb{N}$ (see Theorem 4). Our approach begins with a proof of PSPACE hardness through a reduction from the non-emptiness problem of the intersection of regular languages [21].

Next, we demonstrate PSPACE membership by reducing the problem to state reachability of a finite transition system which we solve in polynomial space. This reduction faces challenges from two main sources, namely, (i) the unbounded size of the write buffers, and (ii) the infinite data domain $\mathtt{D}$. In this section, we show how to construct a finite transition system while preserving state reachability in two key steps.

Following [14], we first perform a buffer abstraction. An in-depth analysis of the TSO semantics within context bounded runs reveals a critical insight: Even though the buffer may contain an unbounded number of writes, only a bounded number of these writes can be read later on. This allows us to non-deterministically identify and store the necessary writes using variables.

Finally, we implement a domain abstraction. A popular approach is to abstract the values into equivalence classes based on the supported relations. This reveals our next challenge: (iii) the set of relations $\mathsf{Rl}_{\leq n}$ is infinite. We conduct an analysis of the reachable configurations and discover the following: If a configuration is reachable, then any configuration that is the same except with greater distances between differing values is reachable as well. It follows that, for control state reachability, the abstraction does not require the precise distances between variables; their relative order is sufficient.

6.1 Lower-bound

We establish PSPACE hardness by polynomially reducing the problem of checking non-emptiness of the intersection of regular languages to $\mathsf{CB}(k)$-$\mathtt{Reach}[\mathsf{D}, \mathsf{Rl}_{\leq n}]$. Given a set of finite automata $\mathcal{A}_1 \ldots \mathcal{A}_n$ with $\mathcal{A}_i = \langle \mathcal{Q}_i, \Delta_i, q_i^{\mathsf{init}} \mathcal{Q}_i^F \rangle$, where $\Delta_i \subseteq \mathcal{Q}_i \times \Sigma \times \mathcal{Q}_i$, $q_i^{\mathsf{init}} \in \mathcal{Q}_i$, and $\mathcal{Q}_i^F \subseteq \mathcal{Q}_i$ for $i \leq n$, the problem asks whether there is a word $w \in \Sigma^*$ that is accepted by each automaton $\mathcal{A}_i$ with $i \leq n$. This is known to be PSPACE hard[21].

We construct a program $\mathsf{Prog}(\mathcal{A}_1 \ldots \mathcal{A}_n)$ that consists of a single thread and reaches a state q_{final} if and only if there is such a word. The idea of the construction is that we assign each state $q_i \in \mathcal{Q}_i$ a unique value stored in a register r_{q_i} and we store the value of the current state of each automaton $\mathcal{A}_i$ in a register r_i. To begin, we ensure that the current states are the initial ones. This means $r_i = r_{q_i^{\mathsf{init}}}$ holds for each $i \leq n$. Then, we choose a letter $a \in \Sigma$ and simulate some transition $q_i \xrightarrow{a} q_i' \in \Delta_i$ for each automaton. This is done by ensuring that the current state is q_i with $r_i = r_{q_i}$ and then updating the current state with $r_i := r_{q_i'}$. We repeat this step until each current state is a final state. At this point, we know we have simulated runs for each automaton that accept the same word and we reach q_{final}.

The formal definition of the construction as well as the proof of correctness is given in Appendix B of [3]. This is a polynomial reduction of non-emptiness of the intersection of regular languages to $\mathsf{CB}(k)$-$\mathtt{Reach}[\mathsf{D}, \mathsf{Rl}_{\leq n}]$. Observe that we only need test for equality and disequality. The disequalitiy checks are necessary to ensure that each register r_{q_i} has been assigned a different value.

Theorem 3. *$\mathsf{CB}(k)$-$\mathtt{Reach}[\mathsf{D}, \mathsf{Rl}_{\leq n}]$ is* PSPACE *hard.*

6.2 PSPACE Upper-bound

Assume that we are given a program Prog and a context bound k. As an intermediary step towards finite state space we construct a finite state machine $\mathtt{AB}(\mathsf{Prog}, k)$ with variables, over the infinite data domain D. The name $\mathtt{AB}$ stands for *abstract buffer* as it abstracts from the unbounded write buffers using a finite number of variables. We show that $\mathtt{AB}(\mathsf{Prog}, k)$ is state reachability equivalent with the TSO semantics of Prog bound by $\mathsf{CB}(k)$.

While abstracting away the buffers, the main challenge is to simulate read operations. Recall from Section 3 that each read operation in a thread accesses either a write from its own buffer or from the shared memory. A buffer read always reads from the threads latest write on the same variable. Since only the active thread may interact with the memory during the context, we can assume w.l.o.g. that all memory updates occur at the end of a context. This means a memory read accesses the last write on the same variable that updated the memory in an earlier context, and hence we do not need to store the whole buffer content. For *memory reads*, we need the latest writes leaving the buffer at the end of each context for each variable. For *buffer reads*, we only require the latest writes on each variable that are issued by each thread.

Construction of the abstract machine The abstract machine $\mathsf{AB}(\mathsf{Prog}, k)$ is defined by the tuple $\langle \mathcal{Q}_{\mathsf{AB}}, \mathcal{X}_{\mathsf{AB}}, \Delta_{\mathsf{AB}}, q^{\mathsf{AB}}_{\mathsf{init}} \rangle$ where $\mathcal{Q}_{\mathsf{AB}}$ is the finite set of states, $\mathcal{X}_{\mathsf{AB}}$ is the finite set of variables, Δ_{AB} the transition relation, and $q^{\mathsf{AB}}_{\mathsf{init}}$ is the initial state. A control state $q_{\mathsf{AB}} \in \mathcal{Q}_{\mathsf{AB}}$ is a tuple $(\mathsf{St}, act, j, c, u)$ where: (i) the current state of every thread is stored using function $\mathsf{St} : \mathcal{T} \to \mathcal{Q}$; (ii) function $act : \{1 \ldots k\} \to \mathcal{T}$ assigns to each context an active thread; (iii) the current context is stored in variable $j \in \{1 \ldots k\}$; (iv) the function $c : \mathcal{X} \times \mathcal{T} \to \{0, 1 \ldots k\}$ assigns to each variable $x \in \mathcal{X}$ and thread $t \in \mathcal{T}$, the (future) context j' in which the latest write on x will leave the write buffer of t. This determines when t can access the shared memory on that variable again; and (v) function $u : \{1 \ldots k\} \to 2^{\mathcal{X}}$ assigns each context j the set of variables that are updated during j. Additionally, we will introduce some helper states with the transitions relation. We omit them from the definition of $\mathcal{Q}_{\mathsf{AB}}$. The initial state $q^{\mathsf{AB}}_{\mathsf{init}}$ is such a helper state.

The set of variables $\mathcal{X}_{\mathsf{AB}}$ contains: (i) the set of variables $\mathcal{X}$ in Prog, (ii) the set of registers $\mathcal{R}$, (iii) for each each context $j \leq k$ and each variable $x \in \mathcal{X}$, we introduce a variable x_j, which stores the value of the last write on x that leaves the write buffer in context j, (iv) for each thread t and each variable $x \in \mathcal{X}$, we introduce a variable x_t which stores the value of the newest write of t on x that is still in the buffer of t. Notice that this is the write that t accesses when reading x (if such a write exists).

We define the transition relation Δ_{AB} in Figure 4. Let $c_{\mathsf{init}}(x, t) = 0$ for all $x \in \mathcal{X}$ and $t \in \mathcal{T}$, and $u_{\mathsf{init}}(i) = \emptyset$ for all $i \in \{1, \ldots, k\}$. The outgoing transitions of state $q^{\mathsf{AB}}_{\mathsf{init}}$ are the outgoing transitions of $(\mathsf{St}_{\mathsf{init}}, act, 0, c_{\mathsf{init}}, u_{\mathsf{init}})$ for every possible function act. This means the construction guesses a function act and behaves as if the other elements in the tuple have the initial values. Local transitions are adapted in a straightforward manner. A read on x from the buffer occurs if there is a write on x in the buffer. This means the latest write on x leaves the buffer in a context $c(x, t)$ after (or in) the current context j. In such a case, we access x_t which holds the latest write on x in the buffer of t. If there is no such write on x in the buffer, i.e. $c(x, t) < j$ holds, then the read fetches the value of x from the shared memory.

A write operation on x overwrites the latest entry in the write buffer on that variable x_t and determines a future (or current) context j' with $j' \geq j$ in which it leaves the buffer. This is recorded in the variable $x_{j'}$ and x is added to the set $u(j')$ which holds the variables that are updated in context j'. Note that j' cannot be smaller then any other context in which a write on a variable y leaves the buffer of t. This information is obtained from the function c. Also, j' must be a context in which t is active.

$$\frac{\langle q'_{\mathtt{AB}}, op, q''_{\mathtt{AB}}\rangle \in \Delta_{\mathtt{AB}} \quad q'_{\mathtt{AB}} = (\mathsf{St}_{\mathsf{init}}, act, 0, c_{\mathsf{init}}, u_{\mathsf{init}})}{\langle q^{\mathtt{AB}}_{\mathsf{init}}, op, q''_{\mathtt{AB}}\rangle} \text{ init} \qquad \frac{op \in \{r_1 := r_2, r_1 := \circledast, \mathsf{rl}(r_1, r_2)\}}{\langle q_{\mathtt{AB}}, op, q_{\mathtt{AB}}\,[\mathsf{St}(t) \leftarrow q_b]\rangle} \text{ local}$$

$$\frac{op = \mathsf{rd}(x, r_1) \quad c(x,t) \geq j}{\langle q_{\mathtt{AB}}, r_1 := x_t, q_{\mathtt{AB}}\,[\mathsf{St}(t) \leftarrow q_b]\rangle} \text{ buffer read} \qquad \frac{op = \mathsf{rd}(x, r_1) \quad c(x,t) < j}{\langle q_{\mathtt{AB}}, r_1 := x, q_{\mathtt{AB}}\,[\mathsf{St}(t) \leftarrow q_b]\rangle} \text{ memory read}$$

$$\frac{op = \mathsf{wt}(x, r_1) \quad j' \geq j \quad act(j') = t \quad j' \geq max\{c((y,t)) \mid y \in \mathcal{X}\}}{\langle q_{\mathtt{AB}}, x_t := r_1, q_\delta\rangle, \langle q_\delta, x_{j'} := r_1, q_{\mathtt{AB}}[\mathsf{St}(t) \leftarrow q_b,\ c(x,t) \leftarrow j', u(j') \leftarrow u(j') \cup \{x\}]\rangle} \text{ write}$$

$$\frac{op = \mathsf{arw}(x, r_1, r_2) \quad j = c(x,t) = max\{c((y,t)) \mid y \in \mathcal{X}\}}{\langle q_{\mathtt{AB}}, x_t = r_1, q_{\delta,1}\rangle\langle q_{\delta,1}, x_j := r_2, q_{\delta,2}\rangle, \langle q_{\delta,2}, x_t := r_2, q_{\mathtt{AB}}[\mathsf{St}(t) \leftarrow q_b, u(j) \leftarrow u(j) \cup \{x\}]\rangle} \text{ buffer arw}$$

$$\frac{op = \mathsf{arw}(x, r_1, r_2) \quad j > c(x,t) \quad j \geq max\{c((y,t)) \mid y \in \mathcal{X}\}}{\langle q_{\mathtt{AB}}, x = r_1, q_\delta\rangle, \langle q_\delta, x := r_2, q_{\mathtt{AB}}[\mathsf{St}(t) \leftarrow q_b]\rangle} \text{ memory arw}$$

$$\frac{q_{\mathtt{AB}} \in \mathcal{Q}_{\mathtt{AB}} \qquad j < k \quad u(j) = \{x^1, \ldots, x^n\}}{\langle q_{\mathtt{AB}}, x^1 := x^1_j, q_{new,1}\rangle \cdots \langle q_{new,n-1}, x^n := x^n_j, q_{\mathtt{AB}}\,[j \leftarrow j+1]\rangle} \text{ context switch}$$

Fig. 4. The transition relation $\Delta_{\mathtt{AB}}$ of $\mathtt{AB}(\mathsf{Prog}, k)$. Let $\delta = \langle q_a, op, q_b\rangle \in \Delta_t$ and $q_{\mathtt{AB}} = (\mathsf{St}, act, j, c, u)$ with $\mathsf{St}(t) = q_a$ and $act(j) = t$.

At any time, the run can switch from a context j with $j < k$ to $j + 1$. Let $u(j) = \{x^1 \ldots x^n\}$. These are the variables that are updated during context j. The values of the last updates on these variables in the context, stored in $x^1_j \ldots x^n_j$, are written to the corresponding variables in the shared memory. Since $\mathtt{AB}(\mathsf{Prog}, k)$ only performs memory updates at the end of a context, an atomic read write $\mathsf{arw}(x, r_1, r_2)$ requires that the current buffer content leaves the buffer in the current context. This is ensured by using the condition $j \geq max\{c((y,t)) \mid y \in \mathcal{X}\}$. If there is a write on x in the buffer of t, then $j = c(x,t)$. This is covered by the *buffer arw* rule in Figure 4. Here, the current value of x is stored in x_t, so we first check that it equals r_1 and update x_t as well as x_j with r_2. If $j > c(x,t)$ holds, then there is no write on x in the buffer of t (*memory arw* rule) and we compare the value of x in the shared memory with r_1 and update it to r_2.

A configuration $\gamma = (q_{\mathtt{AB}}, \mathsf{Mem})$ in the induced LTS of $\mathtt{AB}(\mathsf{Prog}, k)$ consists of a state $q_{\mathtt{AB}} \in \mathcal{Q}_{\mathtt{AB}}$ along with a variable assignment Mem. Let $\gamma_{\mathsf{init}} = (q^{\mathtt{AB}}_{\mathsf{init}}, \mathsf{Mem}_{\mathsf{init}})$ be the initial configuration of $\mathtt{AB}(\mathsf{Prog}, k)$. Given the transitions $\Delta_{\mathtt{AB}}$, we can define the transitions in the induced LTS in a straightforward manner. A state $q_{\mathit{final}} \in \mathcal{Q}_t$ of thread t is said to be reachable by $\mathtt{AB}(\mathsf{Prog}, k)$ if and only if there is a reachable configuration of the form $((\mathsf{St}, act, j, c, u), \mathsf{Mem})$ such that $\mathsf{St}(t) = q_{\mathit{final}}$ holds.

Lemma 2. *A state of* Prog *is reachable under TSO by a run* $\pi \in \mathit{CB}(k)$ *if and only if it is reachable by* $\mathtt{AB}(\mathsf{Prog}, k)$*.*

The proof of Lemma 2 is given in Appendix C of [3]. Next, we abstract away the infinite data domain from $\mathtt{AB}(\mathsf{Prog}, k)$. We remove this last source of infinity by constructing a finite state machine $\mathsf{Rl}_{<}{-}\mathtt{AB}(\mathsf{Prog}, k)$ from $\mathtt{AB}(\mathsf{Prog}, k)$.

$$\frac{\langle q_{\mathtt{AB}}, x := x', q'_{\mathtt{AB}}\rangle \in \Delta_{\mathtt{AB}} \quad x {=_{\mathsf{Rl}'}} x' \quad \forall \mathsf{rl} \in \mathsf{Rl}_<, \forall z, y \in \mathcal{X}_{\mathtt{AB}} \setminus \{x\} : \mathsf{rl}_{\mathsf{Rl}}(y,z) \Leftrightarrow \mathsf{rl}_{\mathsf{Rl}'}(y,z)}{\langle(q_{\mathtt{AB}}, \mathsf{Rl}), x := x', (q'_{\mathtt{AB}}, \mathsf{Rl}')\rangle \in \Delta} \text{ assign}$$

$$\frac{\langle q_{\mathtt{AB}}, x := \circledast, q'_{\mathtt{AB}}\rangle \in \Delta_{\mathtt{AB}} \quad \forall \mathsf{rl} \in \mathsf{Rl}_<, \forall z, y \in \mathcal{X}_{\mathtt{AB}} \setminus \{x\} : \mathsf{rl}_{\mathsf{Rl}}(y,z) \Leftrightarrow \mathsf{rl}_{\mathsf{Rl}'}(y,z)}{\langle(q_{\mathtt{AB}}, \mathsf{Rl}), x := \circledast, (q'_{\mathtt{AB}}, \mathsf{Rl}')\rangle \in \Delta} \text{ new value}$$

$$\frac{\langle q_{\mathtt{AB}}, \mathsf{rl}''(x,y), q'_{\mathtt{AB}}\rangle \in \Delta_{\mathtt{AB}} \quad \mathsf{rl}'' \in \mathsf{Rl}_< \quad \mathsf{Rl} = \mathsf{Rl}' \quad \mathsf{rl}''_{\mathsf{Rl}}(x,y)}{\langle(q_{\mathtt{AB}}, \mathsf{Rl}), \mathsf{rl}''(x,y), (q'_{\mathtt{AB}}, \mathsf{Rl}')\rangle \in \Delta} \ \mathsf{Rl}_< \text{ relation}$$

$$\frac{\langle q_{\mathtt{AB}}, \mathsf{rl}''(x,y), q'_{\mathtt{AB}}\rangle \in \Delta_{\mathtt{AB}} \quad \mathsf{rl}'' \notin \mathsf{Rl}_< \quad \mathsf{Rl} = \mathsf{Rl}' \quad x <_{\mathsf{Rl}} y}{\langle(q_{\mathtt{AB}}, \mathsf{Rl}), \mathsf{rl}''(x,y), (q'_{\mathtt{AB}}, \mathsf{Rl}')\rangle \in \Delta} \ \mathsf{Rl}_{\leq n} \text{ relation}$$

Fig. 5. The transition relation of $\mathsf{Rl}_<{-}\mathtt{AB}(\mathsf{Prog}, k)$. Sets Rl and Rl' satisfy (i) equality is an equivalence relation; (ii) disequality holds iff equality does not hold; (iii) " $<$ " is a total order on variables that are not equal.

Domain Abstraction We use domain abstraction to solve $\mathtt{CB}(k)$-$\mathtt{Reach}[\mathtt{D}, \mathsf{Rl}_{\leq n}]$ by reducing state reachability of $\mathtt{AB}(\mathsf{Prog}, k)$ to reachability of a finite state machine. We introduce the set of relations $\mathsf{Rl}_< = \{=, \neq, <\}$. To abstract away the infinite data domain, we abstract from the exact values of the variables. Instead of storing actual values, we store which relations from $\mathsf{Rl}_<$ holds between which pairs of variables, which is finite information. This way, we reduce the infinite domain $\mathtt{D}$ to the finite Boolean domain $\mathbb{B}$. For example, $(q_{\mathtt{AB}}, x = y)$ is an abstraction of a configuration $(q_{\mathtt{AB}}, \mathsf{Mem}(x) = 1, \mathsf{Mem}(y) = 1)$. Given a variable assignment Mem and a relation rl, we define $\mathsf{rl}_{\mathsf{Mem}}(x,y) := \mathsf{rl}(\mathsf{Mem}(x), \mathsf{Mem}(y))$. Any variable assignment Mem induces a set of relations $\mathsf{Rl}_{\mathsf{Mem}} = \{\mathsf{rl}_{\mathsf{Mem}} \mid \mathsf{rl} \in \mathsf{Rl}_<\}$ over the variables $\mathcal{X}_{\mathtt{AB}}$. When considering multiple sets of relations we denote a relation $\mathsf{rl} \in \mathsf{Rl}$ as $\mathsf{rl}_{\mathsf{Rl}}$. For a variable assignment Mem, we say set of relations Rl over variables is consistent with Mem if $\mathsf{Rl} = \mathsf{Rl}_{\mathsf{Mem}}$.

Given $\mathtt{AB}(\mathsf{Prog}, k) = \langle \mathcal{Q}_{\mathtt{AB}}, \mathcal{X}_{\mathtt{AB}}, \Delta_{\mathtt{AB}}, q^{\mathtt{AB}}_{\mathsf{init}}\rangle$, we now construct the finite state machine $\mathsf{Rl}_<{-}\mathtt{AB}(\mathsf{Prog}, k) = \langle \mathcal{Q}, \Delta, q_{\mathsf{init}}\rangle$ as follows: $\mathcal{Q} := \mathcal{Q}_{\mathtt{AB}} \times \{\mathsf{rl}_{\mathcal{X}_{\mathtt{AB}}} : \mathcal{X}_{\mathtt{AB}} \times \mathcal{X}_{\mathtt{AB}} \to \mathbb{B} \mid \mathsf{rl} \in \mathsf{Rl}_<\}$. We abstract from a variable assignment by storing in the states which relations are satisfied. The initial state is $q_{\mathsf{init}} = (q^{\mathtt{AB}}_{\mathsf{init}}, \mathsf{Rl}_{\mathsf{Mem}_{\mathsf{init}}})$. We define the transitions of $\mathsf{Rl}_<{-}\mathtt{AB}(\mathsf{Prog}, k)$ in Figure 5. We construct the transitions such that they abstract from the transitions of the LTS induced by the semantics of $\mathtt{AB}(\mathsf{Prog}, k)$. Where the semantics on transitions of $\mathtt{AB}(\mathsf{Prog}, k)$ require that certain values in the configurations before and after the operation are the same, the transitions of $\mathsf{Rl}_<{-}\mathtt{AB}(\mathsf{Prog}, k)$ only require that the relations between variables before and after the relation are the same. For instance, the assign rule for operation $x := x'$ requires that Rl and Rl' are the same for all variables except x and $x =_{\mathsf{Rl}'} x'$ must hold after the operation. Conditions (i)-(iii) in Figure 5 reflect the properties of $\mathsf{Rl}_<$ on values. They ensure that Rl and Rl' have consistent variable assignments. Note that for any operation $<_n$ (or $\leq_n$), we soften the condition to $x <_{\mathsf{Rl}} y$. We will show that this still results in an abstraction precise enough to be state reachability equivalent.

Since $\mathsf{Rl}_<{-}\mathtt{AB}(\mathsf{Prog}, k)$ is a finite state machine, it induces the obvious LTS where a configuration consists of a state. The following lemma shows that the construction is indeed an abstraction of $\mathtt{AB}(\mathsf{Prog}, k)$. We assume Prog uses $\mathsf{Rl}_{\leq n}$.

Lemma 3. *If $q_{\mathtt{AB}}$ is reachable by $\mathtt{AB}(\mathsf{Prog}, k)$, then a state $(q_{\mathtt{AB}}, \mathsf{Rl})$ is reachable by $\mathsf{Rl}_{<}-\mathtt{AB}(\mathsf{Prog}, k)$.*

Proof. Assume $\langle (q_{\mathtt{AB}}, \mathsf{Mem}) \xrightarrow{op} (q'_{\mathtt{AB}}, \mathsf{Mem}')\rangle$. We argue that $\langle (q_{\mathtt{AB}}, \mathsf{Rl}_{\mathsf{Mem}}), op, (q'_{\mathtt{AB}}, \mathsf{Rl}_{\mathsf{Mem}'})\rangle \in \Delta$ holds as well. The lemma follows immediately. We show this for operation $x := \circledast$. For all other operations, the proof is analogue and we omit it.

It follows from the semantics of $x := \circledast$, that $\mathsf{Mem}(y) = \mathsf{Mem}'(y)$ for any $y \in \mathcal{X}_{\mathtt{AB}} \setminus \{x\}$ holds. This means $\mathsf{Rl}_{\mathsf{Mem}}$ and $\mathsf{Rl}_{\mathsf{Mem}'}$ satisfy the new value rule. The equality relations in $\mathsf{Rl}_{\mathsf{Mem}}$ and $\mathsf{Rl}_{\mathsf{Mem}'}$ are consistent with the equality relations on values of Mem and Mem'. The equality relation given by the values is an equivalence relation and thus Condition (i) is satisfied. Similarly, Condition (ii) is satisfied since values are obviously not equal if and only if they are not related by equality. Condition (iii) is satisfied since relation $<$ on values forms a total order. All conditions are satisfied. This means $\langle (q_{\mathtt{AB}}, \mathsf{Rl}_{\mathsf{Mem}}), x := \circledast, (q'_{\mathtt{AB}}, \mathsf{Rl}_{\mathsf{Mem}'})\rangle \in \Delta$.

Lemma 4. *If a state $(q_{\mathtt{AB}}, \mathsf{Rl})$ is reachable by $\mathsf{Rl}_{<}-\mathtt{AB}(\mathsf{Prog}, k)$, then $q_{\mathtt{AB}}$ is reachable by $\mathtt{AB}(\mathsf{Prog}, k)$.*

We prove this by performing an induction over runs of $\mathsf{Rl}_{<}-\mathtt{AB}(\mathsf{Prog}, k)$ and constructing equivalent runs of $\mathtt{AB}(\mathsf{Prog}, k)$. In order to do this, we construct configurations with consistent variable assignments. The main challenge is that these variable assignments may not have large enough distances between the values. Take the operation $x <_n y$, for instance. Here, $\mathsf{Rl}_{<}-\mathtt{AB}(\mathsf{Prog}, k)$ only requires $x < y$. Note that any value other than 0 was created by an $x := \circledast$ operation. We can modify a run so that some of these operations assign larger values. This way, we can increase the distances of variable assignments of reachable configurations without changing their consistency with respect to relations. The formal proof of this is given in Appendix E of [3].

Theorem 4. *$\mathit{CB}(k)$-$\mathtt{Reach}[\mathsf{D}, \mathsf{Rl}_{\leq n}]$ is* PSPACE *complete.*

Proof. While $\mathsf{Rl}_{\leq n}$ is an infinite set, $\mathsf{Rl}_{<}$ has only 3 relations. This means $\mathsf{Rl}_{<}-\mathtt{AB}(\mathsf{Prog}, k)$ is a finite transition system where state reachability is decidable. According to Lemma 2, Lemma 3 and Lemma 4, deciding state reachability of $\mathsf{Rl}_{<}-\mathtt{AB}(\mathsf{Prog}, k)$ is equivalent to solving $\mathsf{CB}(k)$-$\mathtt{Reach}[\mathsf{Rl}_{\leq n}]$.

We non-deterministically solve the state reachability of $\mathsf{Rl}_{<}-\mathtt{AB}(\mathsf{Prog}, k)$ by guessing a run that is length-bounded by the size of the state space and checking whether it reaches q_{final}. We store the current state $((\mathsf{St}, \mathit{act}, j, c, u), \mathsf{Rl})$ together with a binary encoding of the current length of the run. Note that the state only requires polynomial space. The number of states of $\mathsf{Rl}_{<}-\mathtt{AB}(\mathsf{Prog}, k)$ is exponential in the program size as well as k, which means the binary encoding also requires polynomial space.

We extend the run by choosing to either perform a context switch or an operation. We begin with the initial state $q^{\mathtt{AB}}_{\mathsf{init}}$, which is a special case since we first need to guess a function *act* according to the init rule in Figure 4. To perform an operation, we look at the current state of the active thread $\mathsf{St}(act(j))$, pick an outgoing transition from the program, and update the state according to the corresponding rules given in Figure 4 and Figure 5.

We illustrate this on the new-value operation. Assume we pick the outgoing transition $\langle q_a, x := \circledast, q_b\rangle \in \Delta_{act(j)}$. In this case, we update the state according to the local rule in Figure 4. Then we update the set Rl according to the new-value rule in Figure 5. We leave all relations that do not include x unchanged, and we non-deterministically choose x to be either equal to some variable, or to be between two other adjacent variables, or to be the largest or smallest variable. We update the relations to x accordingly. For any other operation, the changes to Rl are uniquely determined. For writes, we additionally need to non-deterministically pick some future context j' of the update according to the write rule in Figure 4. In the case of a context switch, we perform a series of variable assignments according to the context switch rule.

Note that we do not explicitly construct the entire $\mathsf{Rl}_{<}-\mathtt{AB}(\mathsf{Prog}, k)$ transition system; the program and the rules given in Figure 4 and Figure 5 are sufficient to guess a run. Each step can be performed in polynomial space. Once $\mathsf{St}(act(j)) = q_{final}$ holds, we know q_{final} is reachable. The complexity of this process is in PSPACE. According to Theorem 3, the problem is PSPACE hard as well.

7 Conclusion

We examined safety verification of concurrent programs running under TSO that operate on variables ranging over an infinite domain. We have shown that this is undecidable even if the program can only check the variables for equality and non-equality. We studied a context bounded variant of the problem as well. Here, we solved the problem for programs using relations in $\mathsf{Rl}_{\leq n}$ and showed that it is PSPACE complete.

As future work, we plan to examine more expressive under-approximations of the program behaviour than the presented context bounded analysis and how these under-approximations affect decidability and complexity of the problem. We also intend to explore the problem for additional relations and/or operations a program may perform.

References

1. Abdulla, P.A., Aiswarya, C., Atig, M.F.: Data communicating processes with unreliable channels. In: LICS. pp. 166–175. ACM (2016). `https://doi.org/10.1145/2933575.2934535`, `https://doi.org/10.1145/2933575.2934535`
2. Abdulla, P.A., Atig, M.F., Bouajjani, A., Ngo, T.P.: A load-buffer semantics for total store ordering. LMCS **14**(1) (2018)
3. Abdulla, P.A., Atig, M.F., Furbach, F., Garg, S.: Verification under TSO with an infinite data domain (2024)

4. Abdulla, P.A., Atig, M.F., Furbach, F., Godbole, A.A., Hendi, Y.G., Krishna, S.N., Spengler, S.: Parameterized verification under TSO with data types. In: TACAS 2023. LNCS, vol. 13993, pp. 588–606. Springer (2023). https://doi.org/10.1007/978-3-031-30823-9_30, https://doi.org/10.1007/978-3-031-30823-9_30
5. Abdulla, P.A., Atig, M.F., Phong, N.T.: The best of both worlds: Trading efficiency and optimality in fence insertion for TSO. In: ESOP 2015. LNCS, vol. 9032, pp. 308–332. Springer (2015). https://doi.org/10.1007/978-3-662-46669-8_13, https://doi.org/10.1007/978-3-662-46669-8_13
6. Abdulla, P.A., Atig, M.F., Rezvan, R.: Parameterized verification under TSO is PSPACE-complete. Proc. ACM Program. Lang. **4**(POPL), 26:1–26:29 (2020). https://doi.org/10.1145/3371094, https://doi.org/10.1145/3371094
7. Abdulla, P.A., Atig, M.F., Stenman, J.: Dense-timed pushdown automata. In: LICS. pp. 35–44. IEEE Computer Society (2012). https://doi.org/10.1109/LICS.2012.15, https://doi.org/10.1109/LICS.2012.15
8. Abdulla, P.A., Cerans, K., Jonsson, B., Tsay, Y.: Algorithmic analysis of programs with well quasi-ordered domains. Inf. Comput. **160**(1-2), 109–127 (2000). https://doi.org/10.1006/inco.1999.2843, https://doi.org/10.1006/inco.1999.2843
9. Abdulla, P.A., Delzanno, G.: On the coverability problem for constrained multiset rewriting. In: Proc. AVIS'06, The fifth Int. Workshop on on Automated Verification of Infinite-State Systems (2006)
10. Abdulla, P.A., Jonsson, B.: Verifying programs with unreliable channels. In: LICS. pp. 160–170. IEEE Computer Society (1993). https://doi.org/10.1109/LICS.1993.287591, https://doi.org/10.1109/LICS.1993.287591
11. Abdulla, P.A., Sistla, A.P., Talupur, M.: Model checking parameterized systems. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking, pp. 685–725. Springer (2018). https://doi.org/10.1007/978-3-319-10575-8_21, https://doi.org/10.1007/978-3-319-10575-8_21
12. Alur, R., Dill, D.L.: A theory of timed automata. Theor. Comput. Sci. **126**(2), 183–235 (1994). https://doi.org/10.1016/0304-3975(94)90010-8, https://doi.org/10.1016/0304-3975(94)90010-8
13. Atig, M.F., Bouajjani, A., Burckhardt, S., Musuvathi, M.: On the verification problem for weak memory models. In: SIGPLAN-SIGACT. pp. 7–18. ACM (2010). https://doi.org/10.1145/1706299.1706303, https://doi.org/10.1145/1706299.1706303
14. Atig, M.F., Bouajjani, A., Parlato, G.: Getting rid of store-buffers in TSO analysis. In: CAV. LNCS, vol. 6806, pp. 99–115. Springer (2011). https://doi.org/10.1007/978-3-642-22110-1_9, https://doi.org/10.1007/978-3-642-22110-1_9
15. Bouajjani, A., Derevenetc, E., Meyer, R.: Checking and enforcing robustness against TSO. In: ESOP 2013. LNCS, vol. 7792, pp. 533–553. Springer (2013). https://doi.org/10.1007/978-3-642-37036-6_29, https://doi.org/10.1007/978-3-642-37036-6_29
16. Bouajjani, A., Esparza, J., Maler, O.: Reachability analysis of pushdown automata: Application to model-checking. In: CONCUR. LNCS, vol. 1243, pp. 135–150. Springer (1997). https://doi.org/10.1007/3-540-63141-0_10, https://doi.org/10.1007/3-540-63141-0_10
17. Burckhardt, S.: Principles of eventual consistency. FTPL **1**(1-2), 1–150 (2014). https://doi.org/10.1561/2500000011, https://doi.org/10.1561/2500000011
18. Cerans, K.: Deciding properties of integral relational automata. In: ICALP94 Proceedings. LNCS, vol. 820, pp. 35–46. Springer (1994). https://doi.org/10.1007/3-540-58201-0_56, https://doi.org/10.1007/3-540-58201-0_56

19. Elver, M., Nagarajan, V.: TSO-CC: consistency directed cache coherence for TSO. In: HPCA. pp. 165–176. IEEE Computer Society (2014). https://doi.org/10.1109/HPCA.2014.6835927, https://doi.org/10.1109/HPCA.2014.6835927
20. Finkel, A., Schnoebelen, P.: Well-structured transition systems everywhere! Theor. Comput. Sci. **256**(1-2), 63–92 (2001). https://doi.org/10.1016/S0304-3975(00)00102-X, https://doi.org/10.1016/S0304-3975(00)00102-X
21. Kozen, D.: Lower bounds for natural proof systems. In: 18th Annual Symposium on Foundations of Computer Science (SFCS 1977). pp. 254–266 (1977). https://doi.org/10.1109/SFCS.1977.16
22. Krishna, S.N., Godbole, A., Meyer, R., Chakraborty, S.: Parameterized verification under release acquire is PSPACE-complete. In: Milani, A., Woelfel, P. (eds.) PODC. pp. 482–492. ACM (2022). https://doi.org/10.1145/3519270.3538445, https://doi.org/10.1145/3519270.3538445
23. La Torre, S., Madhusudan, P., Parlato, G.: Reducing context-bounded concurrent reachability to sequential reachability. In: CAV. LNCS, vol. 5643, pp. 477–492. Springer (2009). https://doi.org/10.1007/978-3-642-02658-4_36, https://doi.org/10.1007/978-3-642-02658-4_36
24. Lahav, O., Giannarakis, N., Vafeiadis, V.: Taming release-acquire consistency. In: SIGPLAN-SIGACT. pp. 649–662. ACM (2016). https://doi.org/10.1145/2837614.2837643, https://doi.org/10.1145/2837614.2837643
25. Lal, A., Reps, T.W.: Reducing concurrent analysis under a context bound to sequential analysis. FMSD **35**(1), 73–97 (2009). https://doi.org/10.1007/s10703-009-0078-9, https://doi.org/10.1007/s10703-009-0078-9
26. Lamport, L.: A new solution of dijkstra's concurrent programming problem. Commun. ACM **17**(8), 453–455 (aug 1974). https://doi.org/10.1145/361082.361093, https://doi.org/10.1145/361082.361093
27. Lazic, R., Newcomb, T.C., Ouaknine, J., Roscoe, A.W., Worrell, J.: Nets with tokens which carry data. Fundam. Informaticae **88**(3), 251–274 (2008), http://content.iospress.com/articles/fundamenta-informaticae/fi88-3-03
28. Musuvathi, M., Qadeer, S.: Iterative context bounding for systematic testing of multithreaded programs. In: PLDI. pp. 446–455. ACM (2007). https://doi.org/10.1145/1250734.1250785, https://doi.org/10.1145/1250734.1250785
29. Owens, S., Sarkar, S., Sewell, P.: A better x86 memory model: x86-TSO. In: TPHOLs. LNCS, vol. 5674, pp. 391–407. Springer (2009). https://doi.org/10.1007/978-3-642-03359-9_27, https://doi.org/10.1007/978-3-642-03359-9_27
30. Qadeer, S., Rehof, J.: Context-bounded model checking of concurrent software. In: TACAS. LNCS, vol. 3440, pp. 93–107. Springer (2005)
31. Ros, A., Kaxiras, S.: Racer: TSO consistency via race detection. In: MICRO. IEEE Computer Society (2016). https://doi.org/10.1109/MICRO.2016.7783736, https://doi.org/10.1109/MICRO.2016.7783736
32. Sarkar, S., Sewell, P., Alglave, J., Maranget, L., Williams, D.: Understanding POWER multiprocessors. In: ACM SIGPLAN, PLDI. pp. 175–186. ACM (2011). https://doi.org/10.1145/1993498.1993520, https://doi.org/10.1145/1993498.1993520
33. Sewell, P., Sarkar, S., Owens, S., Nardelli, F.Z., Myreen, M.O.: x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. Commun. ACM **53**(7), 89–97 (2010). https://doi.org/10.1145/1785414.1785443, https://doi.org/10.1145/1785414.1785443