Context-Bounded Analysis of TSO Systems

Mohamed Faouzi Atig[1], Ahmed Bouajjani[2], and Gennaro Parlato[3]

[1] Uppsala University, Sweden
[2] LIAFA, Université Paris Diderot & Institut Universitaire de France, France
[3] School of Electronics and Computer Science, University of Southampton, UK

Abstract. We address the state reachability problem in concurrent programs running over the TSO weak memory model. This problem has been shown to be decidable with non-primitive recursive complexity in the case of finite-state threads. For recursive threads this problem is undecidable. The aim of this paper is to provide under-approximate analyses for TSO systems that are decidable and have better (elementary) complexity. We propose three bounding concepts for TSO behaviors that are inspired from the concept of bounding the number of context switches introduced by Qadeer and Rehof for the sequentially consistent (SC) model. We investigate the decidability and the complexity of the state reachability problems under these three bounding concepts for TSO, and provide reduction of these problems to known reachability problems of concurrent systems under the SC semantics.

1 Introduction

Sequential consistency is the standard interleaving model for shared memory concurrent programs, where computations of a concurrent programs are interleaved sequences of actions of the different threads, performed in the same order as they appear in the program. However, for performance reasons, modern multi-processors do not preserve in general the program order, that is, they may actually reorder actions executed by a same thread. This leads to so-called weak or relaxed memory models. One of such models is TSO (Total Store Order), which is adopted for instance in x86 machines [36]. In TSO, write operations can be delayed and overtaken by read operations. This corresponds to the use of FIFO store buffers, one per processor, where write operations wait until they are committed in the main memory. Writes are therefore not visible immediately, which may lead to undesirable behaviors since older values than expected may be read along program computations.

Actually, for data-race free programs it can be shown that weak memory models such as TSO induce the same semantics as SC, that is, all possible computations under TSO are also possible under SC [35,4,5,9,18,31,21]. However, data-race-freedom cannot be ensured in all situations. This is for instance the case for low level lock-free programs used in many concurrency libraries and other performance-critical system services. The design of such algorithms, which must be aware of the underlying memory model, is in general extremely difficult

S. Bensalem et al. (Eds.): FPS 2014 (Sifakis Festschrift), LNCS 8415, pp. 21–38, 2014.

due to the unintuitive and hard to predict effects of the weak memory models. Therefore, it is important to develop automatic verification techniques for programs running on such memory models.

In this paper, we focus on the TSO model and we address the state reachability problem, i.e., whether a state of the program (composed by the control locations of the threads and the memory state) is reachable from an initial state. This problem is of course relevant for checking (violations of) safety properties. To reason about programs running over TSO, we adopt an operational model based on parallel automata with unbounded FIFO queues representing the store buffers. The automata model the threads running on each of the processors. These automata are finite-state when programs do not have recursive procedure calls. For the case of recursive programs, threads are modeled using pushdown automata (automata with unbounded stacks). Note that our models have unbounded stacks and unbounded queues. In fact, although these structures are necessarily finite in actual machines, we may not assume any fixed bound on their size, so a finite-state model would not be sufficient to reason about the correctness of a general algorithm for all possible values of these bounds.

Even for finite-state processor threads, the decidability of the state reachability problem under TSO is not trivial due to the unboundedness of the queues. However, it has been shown that this problem is actually decidable, but unfortunately with very high complexity [11]. Indeed, the complexity of state reachability jumps from PSPACE for SC to non-primitive recursive for TSO. As for the case of recursive programs, it is easy to prove that the problem is undecidable as for SC. Therefore, it is important to investigate conditions under which the complexity of this problem becomes elementary, and for which decidability can be obtained even in the case of recursive programs. The approach we adopt in this paper for this purpose is based on the idea of bounding the number of context switches that has been used for the analysis of shared memory concurrent programs under SC [34].

An important issue is to define a suitable notion of context in the case of TSO systems that offers a good trade-off between coverage, decidability and complexity. The direct transposition of the definition for SC to this case consists in considering that a context is a computation segment where only one processor thread is active. This *processor-centric* definition does not restrict the behavior of the memory manager which can execute at any time write operations taken from any store buffer. A *memory-centric* definition, that is the dual of the previous one, considers that in a context only one store buffer is used for memory updates, without restricting the behaviors of the processor threads. Finally, a combination of the two previous definitions leads to a notion of context where only one processor thread is active, and only its store buffer can be used for memory updates. Notice that the three definitions above coincide with the one for SC when all write operations are immediately executed (i.e., the store buffers are of size 0).

We study the decidability and complexity of the analyses corresponding to these three definitions, named pc-CBA, mc-CBA, and pmc-CBA, for processor,

memory, processor-memory centric context-bounded analysis, respectively. In terms of behavior coverage, pc-CBA and mc-CBA are incomparable, and both of them subsume clearly pmc-CBA.

Actually, pmc-CBA coincides with the analysis that we have introduced and studied in [13]. Interestingly, this analysis can be reduced linearly to the context-bounded analysis for SC, and therefore both analysis have the same decidability and complexity characteristics. In addition to the fact that this analysis is decidable and has an elementary complexity (as opposed to the general TSO reachability analysis which is non-primitive recursive as mentioned above), a nice feature of this reduction is that the resulting analysis does not need explicit representation for the contents of the queues. It is possible to show that the content of the queue can be simulated in this case by adding a linear number of additional copies of the global variables. Also, our result allows to use for analyzing programs under TSO all the techniques and tools developed for SC context-bounded analysis, especially those based on code to code translations to sequential programs [28,26].

Then, the main contributions of this paper concern the decidability and complexity of the other two more powerful analyses pc-CBA and mc-CBA. First, we prove that in the case of finite-state processor threads, the pc-CBA is decidable with an elementary complexity. The complexity upper bound we have is polynomial in the size of the state space of the program (product of the thread automata and the memory state) and doubly exponential in the number of contexts. The proof is based on a reduction to the reachability problem of bounded-reverse-phase multiply pushdown automata (brp-MPDA). These models are multi-stack automata where all computations have a bounded number computation segments called reverse-phases, and within each of these segments only one stack can be used in a non-restricted way, while all the others can only be used for pop operations [32]. The name of reverse-phase is by opposition to the name of phase, used in a preceding work introducing bounded-phase multiply pushdown automata (bp-MPDA) [24], where again only one stack is unrestricted while the others can only be used for push operations. The decidability of the reachability problem in bp-MPDA and brp-MPDA has been established in [24] and [32], respectively.

The reduction from TSO systems to brp-MPDA is far from being trivial. The difficulty is, for each context, in order to simulate with a stack the FIFO queue representing the store buffer of the active threads. A naive way to do it would use an unbounded number of reverse-phases (for stack rotations). We show, and this is the tricky part of the proof, that this is actually possible with only one stack rotation for each context, due to the particular semantics of the store buffers. For the case of recursive threads, we prove that however, the pc-CBA is surprisingly undecidable. Furthermore, we prove that the mc-CBA has the same decidability and complexity characteristics as the pc-CBA. The decidability is in this case obtained by a reduction to the bp-MPDA mentioned above, and the undecidability is established following the same lines as in the previous case.

Related work: Context-bounded analysis has been introduced in [34] as an under-approximate analysis for bug detection in multithreaded programs. It has been subsequently widely studied and extended in several works, e.g., in [28,25,26,14,16]. All these works consider the SC semantics. Our work extends this kind of analysis to programs running over weak memory models.

The decidability and the complexity of the state reachability problem for TSO (without restriction on the behaviors) and for other weak memory models (such as PSO) have been established in [11,12]. We are not aware of other work investigating the decidability and complexity results of the state reachability problem for weak memory models.

Testing and bounded model checking algorithms have been proposed for TSO in [19,20,7]. These methods cannot cover sets of behaviors for arbitrary sizes of the store buffers. Algorithmic methods based on abstractions or on bounding the size of store buffers are proposed in [23,6,2,1,3]. In [30], a regular model checking-based approach, using finite-state automata for representing sets of store buffer contents is proposed. The analysis delivers the precise set of reachable configurations when it terminates, but termination is not guaranteed in general.

Checking (trace-)robustness against TSO, i.e., whether all traces of a given program running over TSO are also traces of computations over SC, has been addressed in [33,8,17,15]. This problem has been shown to be decidable in [17] and to be polynomially reducible to state reachability for SC in [15]. Trace-robustness and (safety-)correctness for SC imply correctness for TSO, but the converse in not true.

2 Concurrent Pushdown Systems

In this section we define concurrent pushdown systems (CPDS) with two semantics: *Sequential Consistency* (SC) and *Total-Store-Order* (TSO). Moreover, we define a behaviour-language reachability problem for them.

2.1 Memory Model

A (shared) *memory model* is a tuple $M = (Var, D, \eta^0, T)$, where Var is a finite set of variable names, D is a finite domain of all variables in Var, $\eta^0 : Var \to D$ is an initial valuation, and T is a finite set of thread names. The set of *memory operations* M_{op} is defined as the smallest set containing the following: nop (*no-operation*), $\mathsf{r}(x, d)$ (*read*), $\mathsf{w}(x, d)$ (*write*), $\mathsf{arw}(x, d, d')$ (*atomic read-write*), for every $x \in Var$ and $d, d' \in D$.

We define the *action function* $act_M : M_{op} \to \{nop, read, write, atomicRW\}$ that maps each memory operation in its type. The *size* of a memory model M, denoted $|M|$, is $|M_{op}| + |D| + |Var|$.

Below, we give the SC and TSO semantics for a memory model.

Sequential Consistency (SC): An SC-*configuration* of a memory model M consists of a valuation map $\eta : Var \to D$. A configuration η is *initial* if $\eta = \eta^0$. Given two SC-configurations η and η' of M, there is an SC-*transition* from η to η' on an operation $op \in M_{op}$ performed by thread t, denoted $\eta \xrightarrow[\text{SC},M,t]{op} \eta'$, if one of the following holds:

[nop] $op = \mathsf{nop}$, and $\eta = \eta'$;
[read] $op = \mathsf{r}(x,d)$, $\eta(x) = d$, and $\eta = \eta'$;
[write] $op = \mathsf{w}(x,d)$, $\eta'(x) = d$, and $\eta'(y) = \eta(y)$ for every $y \in (Var \setminus \{x\})$;
[atomic-read-write] $op = \mathsf{arw}(x,d,d')$, and $\eta \xrightarrow[\text{SC},M,t]{\mathsf{r}(x,d)} \eta \xrightarrow[\text{SC},M,t]{\mathsf{w}(x,d')} \eta'$.

Total Store Order (TSO): In TSO, each thread $t \in T$ is equipped with a FIFO queue σ_t to store write operations performed by t. When t writes value d into variable x, the pair (x,d) is enqueued into σ_t. Write operations stored in queues will affect the content of the shared variables only later in time: a pair (x,d) is non-deterministically dequeued from one of the queues and only at that time d is written into x, hence visible to all the other threads. Conversely, when t reads from x, the value that t recovers is the last value that t has written into x, provided that this operation is still pending in σ_t; otherwise, the returned value for x is that stored in the memory.

Formally, a TSO-*configuration* of M is a tuple $C_M = \langle \eta, \{\sigma_t\}_{t \in T} \rangle$, where $\eta : Var \to D$ is a valuation map, and $\sigma_t \in (Var \times D)^*$ for every $t \in T$. C_M is *initial* if $\eta = \eta^0$ and $\sigma_t = \epsilon$ for every $t \in T$ (where ϵ denotes the empty word).

Let $C = \langle \eta, \{\sigma_t\}_{t \in T} \rangle$ and $C' = \langle \eta', \{\sigma'_t\}_{t \in T} \rangle$ be two TSO-configurations of M. There is a TSO-transition from C to C' on $op \in (M_{op} \cup \{mem\})$ performed by thread t, denoted $C \xrightarrow[\text{TSO},M,t]{op} C'$, if one of the following holds:

[nop] $op = \mathsf{nop}$, $\eta' = \eta$, and $\sigma'_h = \sigma_h$ for every $h \in T$;
[read] $op = \mathsf{r}(x,d)$, $C' = C$, and either $\sigma_t = \pi_1.(x,d).\pi_2$ for some $\pi_1 \in (\Sigma \setminus (\{x\} \times D))^*$, or $\sigma_t \in (\Sigma \setminus (\{x\} \times D))^*$ and $\eta(x) = d$;
[write] $op = \mathsf{w}(x,d)$, $\eta' = \eta$, $\sigma'_t = (x,d).\sigma_t$, and $\sigma'_h = \sigma_h$ for every $h \in (T \setminus \{t\})$;
[atomic-read-write] $op = \mathsf{arw}(x,d,d')$, $\sigma'_t = \sigma_t = \epsilon$, $\eta(x) = d$, $\eta'(x) = d'$, $\eta'(y) = \eta(y)$ for every $y \in (Var \setminus \{x\})$, and $\sigma'_h = \sigma_h$ for every $h \in T$;
[memory] $op = mem$, $\sigma_t = \sigma'_t.(x,d)$, $\eta'(x) = d$, $\eta'(y) = \eta(y)$ for every $y \in (Var \setminus \{x\})$, and $\sigma'_h = \sigma_h$ for every $h \in (T \setminus \{t\})$.

2.2 Concurrent Pushdown Systems

We start with pushdown systems which are meant to model a recursive thread.

Pushdown Systems: A *pushdown system* (PDS) is a tuple $A = (Q, q^0, \Gamma, \Delta)$ where Q is a finite set of control states, $q^0 \in Q$ is the initial state, Γ is a finite stack alphabet, and $\Delta = \Delta_{int} \cup \Delta_{push} \cup \Delta_{pop}$ is the set of A moves, with $\Delta_{int} \subseteq Q \times Q$, $\Delta_{push} \subseteq Q \times Q \times \Gamma$, and $\Delta_{pop} \subseteq Q \times \Gamma \times Q$.

A *configuration* of a PDS A is a pair in $Q \times \Gamma^*$. A configuration $\langle q, \gamma \rangle$ is *initial* if $q = q^0$ and $\gamma = \epsilon$. There is a transition from $\langle q, \gamma \rangle$ to a configuration $\langle q', \gamma' \rangle$ on $\delta \in \Delta$, denoted $\langle q, \gamma \rangle \xrightarrow[A]{\delta} \langle q', \gamma' \rangle$, if one of the following holds:

[internal move] $\delta = (q, q') \in \Delta_{int}$ and $\gamma' = \gamma$;
[push move] $\delta = (q, q', a) \in \Delta_{push}$ and $\gamma' = a.\gamma$;
[pop move] $\delta = (q, a, q') \in \Delta_{pop}$ and $\gamma = a.\gamma'$.

We define an action map $act_A : \Delta \to \{int, push, pop\}$ where $act_A(\delta) = a$ iff $\delta \in \Delta_a$. The *size* of a PDS $A = (Q, q^0, \Gamma, \Delta)$, denoted $|A|$, is $|Q| + |\Delta|$.

A PDS A is a *finite state system* (FSS) if $act_A(\delta) = int$, for every $\delta \in \Delta$.

Concurrent Pushdown Systems: A *concurrent pushdown system* (CPDS) is composed by a finite number of PDS–one per thread–which communicate through a memory model M according to the SC or the TSO semantics.

Syntax. A CPDS over a finite set of thread names T and memory model $M = (Var, D, \eta^0, T)$ is a set of tuples $A = \{(Q_t, q_t^0, \Gamma_t, \Delta_t^M)\}_{t \in T}$, where $A_t = (Q_t, q_t^0, \Gamma_t, \Delta_t)$ is a PDS (called the *thread* t of A), and $\Delta_t^M \subseteq (\Delta_t \times M_{op})$.

The *size* of a CPDS A with memory M is $|M| \cdot \prod_{t \in T} |A_t|$.

A CPDS A over T is a *concurrent finite state system* (CFSS) if for every $t \in T$, thread A_t of A is a FSS.

Semantics. For MEM $\in$ {SC,TSO}, a MEM-*configuration* of A is a pair $C = \langle \{C_t\}_{t \in T}, C_M \rangle$, where C_t is an A_t configuration and C_M is a MEM-configuration of M. Further, C is *initial* if for every $t \in T$, C_t is the initial configuration of A_t, and C_M is the initial MEM-configuration of M.

Define $Act_T = \{int, push, pop\}$ and $Act_M = \{nop, read, write, atomicRW\}$. Let $Act = (Act_T \times Act_M \times T) \cup (\{(nop, mem)\} \times T)$. There is a MEM-*transition* from $C = \langle \{C_t\}_{t \in T}, C_M \rangle$ to $C' = \langle \{C'_t\}_{t \in T}, C'_M \rangle$ on an action $(a, b, t) \in Act$, denoted $C \xrightarrow[\text{MEM},A]{(a,b,t)} C'$, if $C_h = C'_h$ for every $h \in (T \setminus \{t\})$, $C_M \xrightarrow[\text{MEM},M,t]{op} C'_M$, and one of the following holds:

[thread & memory transition] $(\delta, op) \in \Delta_t^M$ with $a = act_{A_t}(\delta)$ and $b = act_M(op)$, and $C_t \xrightarrow[A_t]{\delta} C'_t$;
[memory transition only] $a = nop$, $b = op = mem$, and $C_t = C'_t$.

2.3 Reachability Problem

A MEM-*run* of A is a sequence $\pi = C_0 \xrightarrow[\text{MEM},A]{(a_1,b_1,t_1)} C_1 \xrightarrow[\text{MEM},A]{(a_2,b_2,t_2)} \dots \xrightarrow[\text{MEM},A]{(a_n,b_n,t_n)} C_n$ for some $n \in \mathbb{N}$, where C_0 is the initial MEM-configuration of A. We define the *behaviour* of π as the sequence $beh(\pi) = (a_1, b_1, t_1)(a_2, b_2, t_2) \dots (a_n, b_n, t_n)$. For a behaviour language $B \subseteq Act^*$, a MEM-configuration C of A is B-*reachable* if there exists a MEM-run π of A such that $C = C_n$ and $beh(\pi) \in B$. We say that C is *reachable* in A if C is (Act^*)-reachable in A.

Reachability problems for CPDS. Given a CPDS A, a MEM-configuration C of A with MEM $\in \{\text{SC},\text{TSO}\}$, and a behaviour language $B \subseteq Act^*$, the *reachability problem* asks whether C is B-reachable in A.

It is well know that the reachability problem is undecidable for SC-configurations with behaviour language Act^*, as 2 stacks suffice to simulate Turing machines. Furthermore, since CPDS with TSO semantics can simulate CPDS with SC semantics, the reachability problem is also undecidable for TSO-configurations (and behaviour language Act^*). However, if we restrict to CFSS, the reachability problem is non-primitive recursive [11].

In the rest of the paper we consider several behaviour languages B in which we study the decidability and complexity of the reachability problem.

3 Processor-Centric Context-Bounded Analysis

In this section we consider *processor-centric context-bounded analysis* (pc-CBA) for CPDS with TSO semantics. A *pc-context* of a CPDS A is a contiguous part of an A run where only transitions from one thread and the memory are allowed. We study both the decidability and the complexity of the reachability problem for CPDS and CFSS under the TSO semantics up to a given number of pc-contexts. We show that the problem is undecidable for CPDS, and decidable with elementary complexity for CFSS.

Formally, let A be a CPDS over a set of thread names T and shared-memory M, and let k be a positive integer. For $t \in T$ we define L_t as the pc-context behaviour language $((Act_T \times Act_M \times \{t\}) \cup (\{(nop, mem\}) \times T))^*$ for thread t. A k *pc-context* behaviour language over T, denoted L_T^k, is the set of all words $w \in Act^*$ which can be factorized as $w_1 w_2 \dots w_k$, where for every $i \in [k]$, $w_i \in L_{t_i}$, for some thread $t_i \in T$. Given a TSO-configuration C of A, the k *pc-context reachability problem* is the problem of deciding whether C is L_T^k-reachable in A.

In the rest of the section we prove the following 2 theorems.

Theorem 1. *For any $k \in \mathbb{N}$ with $k \geq 5$, the k pc-context reachability problem for* CPDS *under* TSO *is undecidable.*

Theorem 2. *For any $k \in \mathbb{N}$, the k pc-context reachability problem for a* CFSS *A under* TSO *is solvable in double exponential time in the size of A and k.*

3.1 Proof of Theorem 1

The undecidability results is given by a reduction from the emptiness problem of the intersection of two context-free languages [22]: for any two PDA A_1 and A_2, we define a CPDS A that can reaches under TSO a special control state within 5 pc-contexts iff there is a word accepted by both A_1 and A_2.

A *pushdown automaton* (PDA) over a finite alphabet Σ is a tuple $D = (Q, q^0, \Gamma, \Delta_\Sigma, F)$, where $\Delta_\Sigma \subseteq \Delta \times \Sigma$, $E = (Q, q^0, \Gamma, \Delta)$ is a PDS, and $F \subseteq Q$. A word $w = a_1 a_2 \dots a_n \in \Sigma^*$ is *accepted* by B iff there is a sequence $C_0 \xrightarrow[E]{\delta_1} C_1 \xrightarrow[E]{\delta_2} \dots C_{n-1} \xrightarrow[E]{\delta_n} C_n$ such that C_0 is the initial configuration of E,

$(\delta_i, a_i) \in \Delta_\Sigma$ for every $i \in [n]$, and $C_n = \langle q_f, \gamma \rangle$ for some $q_f \in F$ and $\gamma \in \Gamma^*$. Define $L(B)$ to be the set of all words in Σ^* accepted by B.

Let A_1 and A_2 be two PDA over Σ. For simplicity's sake, we assume that $\epsilon \notin L(A_1) \cup L(A_2)$ and that in any word $w \in L(A_1) \cup L(A_2)$ there are no two consecutive identical symbols. We define the CPDS A with memory model M and four threads $T = \{t_1, t_2, t_3, t_4\}$ having the property that a configuration in which all threads are in the special control state, say @, is reachable iff there is a word $w \in L(A_1) \cap L(A_2)$; $M = (Var, \Sigma \cup \{\$\}, \eta^0, T)$ with $Var = \{x_1, x_2, x_3, x_4\}$, and $\eta^0(x_i) = \$$ for every $i \in [4]$.

Below we give a concise description of each thread t_i. We assume that all threads (1) never read or write \$ into a variable, and (2) never read consecutively the same symbol from the same variable.

- The description of t_1 is split in two stages. In the first stage, t_1 non deterministically generates a word $w_1 = a_1 a_2 \dots a_n \in \Sigma^+$, one symbol at a time. Each symbol is also pushed into t_1's stack and simultaneously written into x_1. After the first stage, t_1 has w_1^R stored in its own stack.
- Thread t_2, reading symbols from x_1, simulates the PDA A_1. Every symbol read from x_1 is also written into variable x_2. Nondeterministically, t_1 stops the simulation whenever A_1 reaches a final state and enters the special control state @. Let w_2 be the word composed by the sequence of symbols read by t_2 from x_1. Note that, w_2 is a sub-word of w_1 ($w_2 \subseteq w_1$).
- Thread t_3 acts the same as t_2 except that it simulates A_2 and reads from variable x_2 and writes into x_3. Let w_3 be the word read by t_3 from x_2. It is easy to see that $w_3 \subseteq w_2$.
- Thread t_4 reads a word w_4 from x_3 and rewrites w_4^R into x_4 using its stack, and finally enters the control state @. Again, $w_4 \subseteq w_3$.
- In the second stage, t_1 checks whether it can read w_1^R from x_4, where w_1^R is the content of its stack. If this is the case, t_1 enters the control state @.

From above, it is easy to see that when all threads are in the state @ the following property holds: $w_4 \subseteq w_3 \subseteq w_2 \subseteq w_1$ and $w_1 = w_4$; which is true iff $w_1 = w_2 = w_3 = w_4$. Furthermore, $w_2 = w_3$ is also accepted by both A_1 and A_2. Thus, $L(A_1) \cap L(A_2) \neq \emptyset$ iff A reaches in 5 pc-contexts a configuration where all threads are in the control state @, and this concludes the proof.

3.2 Proof of Theorem 2

The proof is given by a reduction to the reachability problem for CPDS under SC semantics constrained to the *bounded-reverse-phase* behaviour language. A bounded-reverse-phase language is defined as follows. For a thread $t \in T$, define $L_t = ((Act_T \times Act_M \times \{t\}) \cup (Act_T \setminus \{push\} \times Act_M \times T))^*$. A word in L_t describes CPDS sub-runs in which only thread t is allowed to take all its transitions, while the other threads are forbidden to use push transitions. For $h \in \mathbb{N}$, a *h-reverse-phase* word w is such that $w \in Act^*$ and can be factorized as $w_1 w_2 \dots w_h$, where for every $i \in [h]$, $w_i \in L_{t_i}$ for some $t_i \in T$. A h-reverse-phase behaviour language

is the set of all *k-reverse-phase* words. For any given $h \in \mathbb{N}$, the k-reverse-phase reachability problem for SC is decidable in double exponential time as shown below.

Theorem 3. *For any $k \in \mathbb{N}$, the k-reverse-phase reachability problem for a* CPDS *A under SC is solvable in double-exponential time in k and $|A|$, where $|A|$ is the size of A.*

Proof. The upper-bound can be shown by a straightforward reduction to the emptiness problem of k-reverse-phase multi-pushdown automata (introduced in [32]) where there is a shared control-state between all the stacks. The latter problem is known to be solvable in double-exponential time in k and exponential time in the size of the model [32,27]. Then there is a trivial reduction from the k-reverse-phase reachability problem for a CPDS A under SC to the emptiness problem of a k-reverse-phase multi-pushdown automaton B by converting A into an automaton without variables and process states (this can be done by encoding the variable valuation and process states in the shared state of B). This will result in an exponential blow-up and so the k-reverse-phase reachability problem for A can be solved in double-exponential-time in k and $|A|$ (since the size of B is exponential in A). □

The reduction is as follows. Let T be the set of thread names of A. We define a CPDS D that non-deterministically *simulates* A along any bounded pc-context using the SC semantics. More specifically, D simulates consecutively each pc-context of A using 2-reverse-phases. Below we only describe the simulation of a single pc-context.

Invariant. At the beginning and the end of the simulation of each pc-context of A, D encodes the configuration of A as follows. D has all threads of A, where for every thread $t \in T$, t encodes the configuration of the thread with the same name in A along with its FIFO queue. More specifically, the control state of t in A is stored in the control state of t in D, and since t does not use its stack at all–as A is a CFSS–the stack of t in D is used to store the FIFO queue σ_t in A with the head pair on the top of the stack. Moreover, the valuation of the shared variables in A is encoded in the shared variables of D. The shared variables of D also include an auxiliary variable used to keep track on whether the automaton is in a pc-context simulation phase. During the simulation of a pc-context an auxiliary thread $s \notin T$ is used. We guarantee that the stack of s is empty whenever D is not in a simulation phase.

Below we describe the 3 steps for the simulation of a pc-context. During the description we also convey a correctness showing that the invariant above holds after the simulation of a pc-context, provided it holds at the very beginning of that pc-context simulation.

Pre-simulation. D non-deterministically selects a thread $t \in T$ that is allowed to progress in the pc-context under simulation. Then, it reverses the content of the stack of t into the stack of s. Note that, the last pair written in the queue of t (in A) is now on the top of the stack of s.

As D copies the stack content, it also computes two pieces of information that are stored in the control state of s.

The first piece of information consists in collecting for each shared variable $x \in Var$ the value corresponding to the last write pair for x, if any, that still resides in the queue. We compute this information to avoid inspecting the stack of s to simulate read operations from t.

The second piece of information η is used to simulate memory operation concerning thread t again to avoid accessing the stack of s. It consists in a sequence of write pairs whose length is bounded by the number of variables of Var. This sequence is defined by the map *lastseq*. For $\sigma = (x_1, d_1) \dots (x_n, d_n) \in (Var \times D)^*$, $lastseq(\sigma)$ is the subsequence of σ in which we remove all pairs (x_j, d_j) such that $x_j = x_i$ and $j < i$. For example, for $\sigma = (y,5)(z,2)(y,4)(x,2)(z,3)(x,1)$, $lastseq(\sigma) = (y,4)(z,3)(x,1)$. η is defined as follows. The queue content γ^R of t in A, where γ is the stack content of t in D at the beginning of the simulation, can be split in two subsequences $\gamma_1\gamma_2$, where γ_2 is the portion of the queue that is dequeued by means of memory operations of t by the end of the simulation of the current pc-context. This partition is not known at the beginning of the simulation, and is non-deterministically guessed by D. We define η as the sequence $lastseq(\gamma_2)$. Again, s uses η to simulate the memory operation from t without using the stack of s. The idea is that only the elements in η are relevant for the simulation as the remaining write pairs will be overwritten by pairs in η by the end of the simulation hence non visible to the other threads.

Since we do not remove elements from the queue (stack of s) during the simulation, we eliminate them only at the end of the simulation when we copy the queue content from the stack of s to that of t. Thus, when the content of the stack of t is reversed into the stack of s at the beginning of the simulation, D non-deterministically guesses the intermediate point between γ_1 and γ_2 and inserts in the stack a separation symbol \$ to remember which part must be discarded. As a remark, it may happen that all pairs in the queue may be used to update the memory in the current pc-round and thus no \$ is inserted in this phase. If this is the case, we need to update the sequence η to keep it consistent as we simulate write operations.

Simulation. After the pre-simulation step, D non-deterministically simulates a sequence of A transitions that may include moves from t and memory transitions of all threads.

A write operation performed by t in A is simulated by pushing the corresponding write pair (x, d) onto the stack of s. Simultaniously, s updates its control to keep track of the last written value for x. Finally, if \$ has not been pushed in the stack yet this pair is also used to update the sequence η by concatenating (x, d) to η and then removing any other existing pair in η for x. After than, s may non-deterministically decide to push \$ onto the stack of s.

A read transition performed by t in A, say on variable x, is simulated by using the last written value for x stored in the control of s, if any, otherwise the value of x in the shared-memory is used. Note that, when we read a value for which we keep track of its value in the control state of s, it may be the case that such

a write pair has already been used to update the shared-memory and we should use this value instead. However, if this is the case these two values coincide as the shared memory cannot be overwritten by any other thread as they are idle in the current pc-context.

A memory transition from the queue of t', for $t' \in T \setminus \{t\}$, is simulated by popping the pair from the stack of t', which contains the head pair of the queue of t', and then by updating the shared-memory accordingly.

To simulate memory transitions from t's queue, we use the sequence η, stored in the control state of s. We remove the leftmost pair from η and update the shared-memory according to it. It is easy to see, that some write pairs are not simulated at all, in particular we do not simulate all write operations that are not captured by η. However, in terms of correctness this is not an issue as all these values will be overwritten in the shared memory by the end of the simulation of the current pc-context by some pair in η.

Restoring the encoding of the reached A configuration. The simulation of the pc-context can non-deterministically end, provided that the sequence η is empty. We restore the configuration of t by copying back the control state of s into t and the content of the stack of s into the stack of t up to the symbol \$ (while discarding the remaining stack content of s).

2 Phase for each pc-context. From the above description it is easy to see that the number of reverse-phases needed to simulate one pc-context are 2: one is required in the first macro step to copy the stack content from t to s, in the second macro step we only pushes on s's stack and hence do not need any extra reverse-phase, and the last step consumes another reverse-phase for the copy of s's stack into the one of t.

4 Memory-Centric Context-Bounded Analysis

In this section, we consider memory-centric context-bounded analysis (mc-CBA) for CPDS and CFSS under TSO semantics. A *mc-context* of a CPDS A is a contiguous part of an A run where only memory transitions concerning the queue of one thread can be performed and no restriction are posed on the actions of all the threads. We study both the decidability and the complexity of the reachability problem for CPDS and CFSS under the TSO semantics up to a given number of mc-contexts. We show that the problem is undecidable for CPDS, and decidable with elementary complexity for CFSS.

Formally, let A be a CPDS over a set of thread names T and shared-memory M, and let k be a positive integer. For $t \in T$, we define L_t as the mc-context behaviour language $((Act_T \times Act_M \times \{T\}) \cup (\{(nop, mem\}) \times \{t\}))^*$ for thread t. A k *mc-context* behaviour language over T, denoted L_T^k, is the set of all words $w \in Act^*$ which can be factorized as $w_1 w_2 \dots w_k$, where for every $i \in [k]$, $w_i \in L_{t_i}$, for some thread $t_i \in T$. Given a TSO-configuration C of A, the k *mc-context reachability problem* is the problem of deciding whether C is L_T^k-reachable in A.

In the rest of the section we prove the following 2 theorems.

Theorem 4. *For any $k \in \mathbb{N}$ with $k \geq 5$, the k mc-context reachability problem for* Cpds *under* Tso *is undecidable.*

Theorem 5. *For any $k \in \mathbb{N}$, the k mc-context reachability problem for a* Cfss *A under* Tso *is solvable in double exponential time in the size of A and k.*

4.1 Proof of Theorem 4

We exploit the construction given in the proof of Theorem 1 to prove the undecidability of the problem.

We show that the Cpds constructed to decide the intersection of the languages accepted by the pushdown automata A_1 and A_2 has also a 5 mc-context run to witness the existence of a common word accepted by both A_1 and A_2, if any. Thread t_1 runs first, until it finishes its first context. Then, synchronously, we interleave the memory transitions on t_1's queue with the transitions of thread t_2 so that it reads the entire words w from x_1 and writes it into its queue on the variable x_2. The same is done for thread t_2 and t_3, and then for t_3 and t_4. Finally actions by t_1 are synchronised with the memory transitions from t_4's queue. It is direct to see that such a schedule leads to a 5 mc-context run, and this concludes the proof.

4.2 Proof of Theorem 5

The proof is given by a reduction to the reachability problem for Cpds under SC semantics constrained to the *bounded-phase* behaviour language. A *phase* captures the dual notion of a reverse-phase, as it represents a contiguous segment of any run in which only one thread can use its stack with no restrictions, instead all the other threads can only push in their own stack. Formally, a bounded-phase language is defined as follows: For a thread $t \in T$, define $L'_t = ((Act_T \times Act_M \times \{t\}) \cup (Act_T \setminus \{pop\} \times Act_M \times T))^*$. A word in L'_t describes Cpds sub-runs in which only thread t is allowed to take all its transitions, while the other threads are forbidden to use pop transitions. For $h \in \mathbb{N}$, a *h-phase* word w is such that $w \in Act^*$ and can be factorized as $w_1 w_2 \dots w_h$, where for every $i \in [h]$, $w_i \in L'_{t_i}$ for some $t_i \in T$. A h-phase behaviour language is the set of all *k-phase* words. For any given $h \in \mathbb{N}$, the k-phase reachability problem for Sc is decidable in double exponential time as for the case of k-reverse-phase reachability problem for Sc (see Theorem 3).

Theorem 6. *For any $k \in \mathbb{N}$, the k-phase reachability problem for a* Cpds *A under SC is solvable in time double-exponential time in k and $|A|$, where $|A|$ is the size of A.*

Proof. The upper-bound can be shown by a straightforward reduction to the emptiness problem of k-phase multi-pushdown automata (introduced in [24]) where there is a shared control-state between all the stacks. The latter problem

is known to be solvable in double-exponential time in k and exponential time in the size of the model [24,32,10]. Then there is a trivial reduction from the k-phase reachability problem for a CPDS A under SC to the emptiness problem of a k-phase multi-pushdown automaton B by converting A into an automaton without variables and process states (this can be done by encoding the variable valuation and process states in the shared state of B). This will result in an exponential blow-up and so the k-phase reachability problem for A can be solved in double-exponential-time in k and $|A|$ (since the size of B is exponential in A).

Before giving the reduction to the bounded-phase reachability problem for CPDS under SC semantics, we show that any mc-phase can be rewritten such that: (1) In the first part of the run, only one thread $t \in T$ is allowed to perform actions and no memory transitions are not allowed for all the threads, (2) and in the second part, only memory transitions concerning the queue of t are allowed and no restrictions are posed on the actions of all threads except the thread t (which is not allowed to perform any action). Formally, for $t \in T$ we define B_t as a restricted mc-context behaviour language $\big((Act_T \times Act_M \times \{t\})^* \cdot ((Act_T \times Act_M \times (T \setminus \{t\})) \cup (\{(nop, mem\}) \times \{t\}))^*\big)$ for thread t. A *k restricted mc-context* behaviour language over T, denoted B_T^k, is the set of all words $w \in Act^*$ which can be factorized as $w_1 w_2 \dots w_k$, where for every $i \in [k]$, $w_i \in L_{t_i}$, for some thread $t_i \in T$. Given a TSO-configuration C of A, the *k restricted mc-context reachability problem* is the problem of deciding whether C is B_T^k-reachable in A.

Let us assume that in a mc-context, we are only performing memory transitions concerning the queue of one process $t \in T$. Then it is easy to see that the execution of t can never be affected by anyone else (since they don't update the memory). Other threads might effect t's execution if they were able to change the configuration of the shared-memory. However, this is not the case as only memory transitions can occur from t's queue. Instead, memory transitions from t do change the state of the shared-memory, but as we now argue, it cannot deviate the course of t's execution. Recall that the behaviour of t depends on: (1) the value of a variable in the memory if there is no pending write for this variable in the queue of t, and (2) the last write operations that are still reside in the queue of t. This means that performing (or not) memory transitions from the queue of t will not affect the behaviour of t. This implies that any mc-context can be reordered such that in we execute first the sequence of actions of the process t and then we execute the sequence of memory transitions and actions of all the other threads. This leads to the fact that the k mc-context reachability problem for a *Tso*-configuration C of A can be reduced to the k restricted mc-context reachability problem for C (which stated by the following lemma):

Lemma 1. *Given a* TSO*-configuration C of A, C is B_T^k-reachable in A iff C is L_T^k-reachable in A.*

Next, we show that it is possible to reduce the k restricted mc-context reachability problem for a CFSS under TSO to the k-phase reachability problem for a CPDS under SC. The reduction we propose is similar in spirit to the one for the processor-centric case (see Proof of Theorem 2), and here we only sketch the

differences. We define a CPDS D that simulates every restricted mc-context of A with 3 phases. The set of thread names of B is $T \cup \{s\}$, where $s \notin T$ is an auxiliary thread which is employed for the simulation. The invariant we maintain is the following: when the simulation starts and ends thread s is in an *idle state* meaning that it is in a special control state, say @, and its stack is empty. Furthermore, every other B threads $t \in T$ encodes in its configuration the one of t in A: in its control state it is encoded the control state of t in A and the finite sequence $last(\sigma_t)$ where σ_t is the content of t's queue in A, and in stores in its stack σ_t with the head write pair placed on top of the stack.

The simulation goes as follows. Initially s guesses the thread t from which memory transitions can be executed. The content of t's stack is transferred into s's stack, where now the tail of t's queue is stored on the top of s's stack. Also the control state of t as well as $last(\sigma_t)$ is copied into the control state of s.

Now, we first simulate the moves of t and only after the moves of the remaining threads along with the memory transitions concerning t's queue (in order to respect the definition of restricted mc-context). The simulation of t is as follows. For write operations we update $last(\sigma_t)$ as described in Section 3.2, and push the produced pair on the stack of s. Read operations, instead, will consult $last(\sigma_t)$ to get the value of the read variable, if any, otherwise it recovers the value from the memory.

In the second stage of the simulation we restore back into t's stack the content of s's stack as well as the control state. The sequence *last* will not be copied as it will change after memory operations will be performed. Such sequence is reconstructed at the end of the simulation.

We now simulate all the other threads and memory updates in arbitrary order. Memory transitions are simulated as expected by popping pairs from t's stack and updating the memory accordingly. Transitions of other threads, say $\hat{t}$ are simulated straightforwardly by using $last(\sigma_{\hat{t}})$ and the shared memory in a similar as we have done for t.

Non deterministically the simulation ends and the invariant is reestablished by computing $last(\sigma_t)$. For such a purpose we need to inspect entirely t's stack. Thus we copy it back and forth to the s's stack by paying one more phase. Finally s enters into the special control state @ and the simulation ends.

By using the same argument as in Section 3.2 we can show that the above construction of B allows to reduce in polytime the k restricted mc-context reachability problem for CFSS under TSO to the 3k-phase reachability problem for CPDSunder SC. Thus, from Lemma 1 and Theorem 6 we can state the main result of the section.

Theorem 7. *For any positive integer k, the k mc-context reachability problem for a* CFSS *A under* TSO *is solvable in double exponential time in $|A|$ and k.*

5 Process-Memory Centric Context-Bounded Analysis

In this section, we consider process-memory centric context-bounded analysis (pmc-CBA) for CPDS with TSO semantics. A context, in this case called *pmc-context*, of a CPDS A is a contiguous part of an A computation where only one

processor thread is active, and only its store buffer can be used for memory updates. We consider here the reachability problem for CPDS up to a bounded number of pmc-contexts. We recall that the pmc-CBA for CPDS (resp.TSO-CFSS) with TSO semantics is reducible to the standard context-bounded analysis for CPDS (resp. CFSS) with SC semantics which is known to be decidable [34].

Next, we formally define the bounded pmc-reachability problem for TSO-CPDSs. Let A be a CPDS over thread names T and shared-memory model M, and let k be a positive integer. The *pmc-context language* L_t of a thread $t \in T$ is the set $\big(((Act_T \times Act_M) \cup \{(nop, mem)\}) \times \{t\}\big)^*$. A k pmc-context behavior language over T, denoted L_T^k, is the set of all words $w \in Act_A^*$ which can be factorized as $w_1 w_2 \cdots w_k$, where for every $i \in [k]$, $w_i \in L_{t_i}$, for some thread $t_i \in T$.

Given a MEM $\in$ {SC,TSO} and MEM-configuration C of A, the *k pmc-context reachability problem* is the problem of deciding whether C is L_T^k-reachable in A.

Theorem 8 ([13]). *For any $k \in \mathbb{N}$, the k pmc-context reachability problem for* CPDS *(resp.* CFSS*) under* TSO *is reducible to the k- pmc-context reachability problem for* CPDS *(resp.* CFSS*) under* SC *semantics.*

Moreover, we have:

Theorem 9. *For any $k \in \mathbb{N}$, the k- pmc-context reachability problem for* CPDS *(resp.* CFSS*) under* SC *semantics is solvable in nondeterministic exponential time in k and $|A|$, where $|A|$ is the size of A.*

Proof. The upper-bound can be shown by a straightforward reduction to the reachability problem of k-context multi-pushdown systems (introduced in [34]) where there is a shared control-state between all the stacks. The latter problem is known to be solvable in non-deterministic polynomial time in k and the size of the system [29]. It is easy to see that there is a trivial reduction from the k-pmc-context reachability problem for a CPDS A under SC to k-context multi-pushdown system B by encoding all the process states and the valuation of the memory into one single state. This will result in an exponential blow-up and so the k-pmc-context reachability problem for A can be solved in nondeterministic exponential-time in k and $|A|$ (since the size of B is exponential in A).

As an immediate corollary of Theorem 8 and Theorem 9, we obtain:

Theorem 10. *For any $k \in \mathbb{N}$, the k pmc-context reachability problem for* CPDS *(resp.* CFSS*) A under* TSO *is decidable and can be solved in nondeterministic exponential-time in k and $|A|$.*

6 Conclusion

We have considered three different notions of context-bounded analysis for TSO computations, depending on whether a processor, or a memory, or a processor and memory centric view is adopted. We have shown that each of these three

notions allows to cut-off drastically the complexity of checking state reachability w.r.t. the unrestricted case, although of course the analysis is under-approximate. The work we present in this paper allows to improve our understanding of the trade-offs between expressiveness, decidability, and complexity of checking state reachability under TSO semantics.

While pmc-CBA was already introduced in our previous work [13], this work introduces two other natural and more general concepts of pc-CBA and pm-CBA for which the complexity of the TSO state reachability problem is still elementary. In terms of coverage, pc-CBA and mc-CBA are incomparable while both of them are strictly more general than pmc-CBA. Indeed, these two analyses allow to capture with a given bound on the pc/mc context switches sets of behaviors that would need an unbounded number of pmc context switches. However, this increase in power comes with a price. First, while pcm-CBA is decidable even for recursive programs (pushdown threads), both pc-CBA and mc-CBA are undecidable in this case. For programs without recursive procedures, pmc-CBA is in NEXPTIME while both pc-CBA and mc-CBA are in 2EXPTIME.

An interesting question left for future work is whether the analyses presented here for TSO can be extended to other weak memory models.

References

1. Abdulla, P.A., Atig, M.F., Chen, Y.-F., Leonardsson, C., Rezine, A.: Automatic fence insertion in integer programs via predicate abstraction. In: Miné, A., Schmidt, D. (eds.) SAS 2012. LNCS, vol. 7460, pp. 164–180. Springer, Heidelberg (2012)
2. Abdulla, P.A., Atig, M.F., Chen, Y.-F., Leonardsson, C., Rezine, A.: Counter-example guided fence insertion under TSO. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 204–219. Springer, Heidelberg (2012)
3. Abdulla, P.A., Atig, M.F., Chen, Y.-F., Leonardsson, C., Rezine, A.: Memorax, a precise and sound tool for automatic fence insertion under TSO. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 530–536. Springer, Heidelberg (2013)
4. Adve, S.V., Hill, M.D.: A unified formalization of four shared-memory models. IEEE Trans. Parallel Distrib. Syst. 4(6), 613–624 (1993)
5. Ahamad, M., Neiger, G., Burns, J.E., Kohli, P., Hutto, P.W.: Causal memory: Definitions, implementation, and programming. Distributed Computing 9(1), 37–49 (1995)
6. Alglave, J., Kroening, D., Nimal, V., Tautschnig, M.: Software verification for weak memory via program transformation. In: Felleisen, M., Gardner, P. (eds.) ESOP 2013. LNCS, vol. 7792, pp. 512–532. Springer, Heidelberg (2013)
7. Alglave, J., Kroening, D., Tautschnig, M.: Partial orders for efficient bounded model checking of concurrent software. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 141–157. Springer, Heidelberg (2013)
8. Alglave, J., Maranget, L.: Stability in weak memory models. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 50–66. Springer, Heidelberg (2011)
9. Aspinall, D., Ševčík, J.: Formalising java's data race free guarantee. In: Schneider, K., Brandt, J. (eds.) TPHOLs 2007. LNCS, vol. 4732, pp. 22–37. Springer, Heidelberg (2007)

10. Atig, M.F., Bollig, B., Habermehl, P.: Emptiness of multi-pushdown automata is 2ETIME-complete. In: Ito, M., Toyama, M. (eds.) DLT 2008. LNCS, vol. 5257, pp. 121–133. Springer, Heidelberg (2008)
11. Atig, M.F., Bouajjani, A., Burckhardt, S., Musuvathi, M.: On the verification problem for weak memory models. In: Hermenegildo, M.V., Palsberg, J. (eds.) POPL, pp. 7–18. ACM (2010)
12. Atig, M.F., Bouajjani, A., Burckhardt, S., Musuvathi, M.: What's decidable about weak memory models? In: Seidl, H. (ed.) ESOP 2012. LNCS, vol. 7211, pp. 26–46. Springer, Heidelberg (2012)
13. Atig, M.F., Bouajjani, A., Parlato, G.: Getting rid of store-buffers in TSO analysis. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 99–115. Springer, Heidelberg (2011)
14. Atig, M.F., Bouajjani, A., Qadeer, S.: Context-bounded analysis for concurrent programs with dynamic creation of threads. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 107–123. Springer, Heidelberg (2009)
15. Bouajjani, A., Derevenetc, E., Meyer, R.: Checking and enforcing robustness against TSO. In: Felleisen, M., Gardner, P. (eds.) ESOP 2013. LNCS, vol. 7792, pp. 533–553. Springer, Heidelberg (2013)
16. Bouajjani, A., Emmi, M., Parlato, G.: On sequentializing concurrent programs. In: Yahav, E. (ed.) SAS 2011. LNCS, vol. 6887, pp. 129–145. Springer, Heidelberg (2011)
17. Bouajjani, A., Meyer, R., Möhlmann, E.: Deciding robustness against total store ordering. In: Aceto, L., Henzinger, M., Sgall, J. (eds.) ICALP 2011, Part II. LNCS, vol. 6756, pp. 428–440. Springer, Heidelberg (2011)
18. Boudol, G., Petri, G.: Relaxed memory models: an operational approach. In: Shao, Z., Pierce, B.C. (eds.) POPL, pp. 392–403. ACM (2009)
19. Burckhardt, S., Musuvathi, M.: Effective program verification for relaxed memory models. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 107–120. Springer, Heidelberg (2008)
20. Burnim, J., Sen, K., Stergiou, C.: Testing concurrent programs on relaxed memory models. In: ISSTA, pp. 122–132. ACM (2011)
21. Friedman, R.: Consistency Conditions for Distributed Shared Memories. Phd. thesis, Technion: Israel Institute of Technology (1994)
22. Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to automata theory, languages, and computation - international edition, 2nd edn. Addison-Wesley (2003)
23. Kuperstein, M., Vechev, M.T., Yahav, E.: Partial-coherence abstractions for relaxed memory models. In: PLDI, pp. 187–198. ACM (2011)
24. La Torre, S., Madhusudan, P., Parlato, G.: A robust class of context-sensitive languages. In: LICS, pp. 161–170. IEEE Computer Society (2007)
25. La Torre, S., Madhusudan, P., Parlato, G.: Analyzing recursive programs using a fixed-point calculus. In: PLDI, pp. 211–222. ACM (2009)
26. La Torre, S., Madhusudan, P., Parlato, G.: Reducing context-bounded concurrent reachability to sequential reachability. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 477–492. Springer, Heidelberg (2009)
27. La Torre, S., Napoli, M., Parlato, G.: On the complement of multi-stack visibly pushdown languages. Technical report (2014)
28. Lal, A., Reps, T.: Reducing concurrent analysis under a context bound to sequential analysis. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 37–51. Springer, Heidelberg (2008)

29. Lal, A., Touili, T., Kidd, N., Reps, T.: Interprocedural analysis of concurrent programs under a context bound. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 282–298. Springer, Heidelberg (2008)
30. Linden, A., Wolper, P.: An automata-based symbolic approach for verifying programs on relaxed memory models. In: van de Pol, J., Weber, M. (eds.) SPIN 2010. LNCS, vol. 6349, pp. 212–226. Springer, Heidelberg (2010)
31. Luchango, V.: Memory Consistency Models for High Performance Distributed Computing. Phd. thesis, Massachusetts Institute of Technology (2001)
32. Madhusudan, P., Parlato, G.: The tree width of auxiliary storage. In: Ball, T., Sagiv, M. (eds.) POPL, pp. 283–294. ACM (2011)
33. Owens, S.: Reasoning about the implementation of concurrency abstractions on x86-TSO. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 478–503. Springer, Heidelberg (2010)
34. Qadeer, S., Rehof, J.: Context-bounded model checking of concurrent software. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 93–107. Springer, Heidelberg (2005)
35. Saraswat, V.A., Jagadeesan, R., Michael, M.M., von Praun, C.: A theory of memory models. In: Yelick, K.A., Mellor-Crummey, J.M. (eds.) PPOPP, pp. 161–172. ACM (2007)
36. 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)