On Bounded Reachability Analysis of Shared Memory Systems*

Mohamed Faouzi Atig[1], Ahmed Bouajjani[2], K. Narayan Kumar[3], and Prakash Saivasan[3]

1 **Uppsala University, Sweden**
`mohamed_faouzi.atig@it.uu.se`
2 **LIAFA, Université Paris Diderot, France**
`abou@liafa.univ-paris-diderot.fr`
3 **Chennai Mathematical Institute, India**
`{kumar,saivasan}@cmi.ac.in`

Abstract

This paper addresses the reachability problem for pushdown systems communicating via shared memory. It is already known that this problem is undecidable. It turns out that undecidability holds even if the shared memory consists of a single boolean variable. We propose a restriction on the behaviours of such systems, called stage bound, towards decidability. A k stage bounded run can be split into a k *stages*, such that in each stage there is at most one process writing to the shared memory while any number of processes may read from it. We consider several versions of stage-bounded systems and establish decidability and complexity results.

1998 ACM Subject Classification D.2.4 Software/Program Verification

Keywords and phrases Reachability problem, Pushdown systems, Counter systems

Digital Object Identifier 10.4230/LIPIcs.FSTTCS.2014.611

1 Introduction

Shared memory concurrent programs are present at different levels of the software stack, from high level applications to low level software implementing system services on multicores. These programs are notoriously complex and hard to get right, which makes extremely important developing verification methods for checking their correctness. However, the design of automatic verification for these programs remains a highly challenging problem. First, it is well known that when threads can perform recursive procedure calls, the state reachability problem (which is relevant for checking safety properties) for these programs is undecidable, even when the manipulated data are finite. In the case where recursion is not allowed (or bounded), the problem is PSPACE-complete and the complexity grows exponentially in terms of the number of threads. Therefore, important issues are investigating the decidability of the state reachability problem under various assumptions on the behaviors of these programs, exploring how far the limits of decidability can be pushed, and understanding the trade-offs between behavior coverage, decidability, and complexity. This paper is a contribution addressing these issues.

To carry out our study, we adopt a formal model that is a network of processes with a shared store ranging over a finite domain, and we consider that processes can be pushdown

* This work was partially supported by the CNRS LIA Informel, the Infosys Foundation, TCS PhD Fellowship and the Uppsala Programming for Multicore Architectures Research Center (UPMARC).

© Mohamed Faouzi Atig, Ahmed Bouajjani, K. Narayan Kumar, and Prakash Saivasan;
licensed under Creative Commons License CC-BY

systems, or 1-counter systems (seen as pushdown systems with a single element stack alphabet), or simply finite-state systems. Each of these processes may perform reads and writes on the shared store.

First, we prove that in order to get decidability of the state reachability, restricting only the data domain of the shared store is not sufficient. Indeed, we show that two parallel 1-counter systems sharing only one bit are able to encode any 2-counter machine. This result implies that, to get decidability, it is necessary to restrict the way information flows through the shared memory.

Then, the idea we consider is the following: For each computation, consider a decomposition into what we call *stages*, where in each stage only one process is unrestricted while all the others are only allowed to read. Then, we only consider computations up to some fixed bound on the number of stages. Notice that this notion of bounding, called *stage-bounding*, does not restrict the way stacks and counters are accessed. It is rather imposing that writes by different processes to the memory cannot interleave in an unbounded manner (while reads are allowed to interleave unboundedly with any kind of operations from any process).

The notion of stage-bounding is somehow inspired by the notion of context-bounding introduced by Qadeer and Rehof in [13]. However, it is clear that stage-bounding is strictly more general than context-bounding in term of behavior coverage. This is due to the fact that operations (reads and writes) by different processes can alternate unboundedly within one single stage.

Interestingly, for networks of finite-state systems, the stage-bounded analysis is NP-complete (while the unbounded analysis is PSPACE-complete as mentioned earlier). So, stage-bounded analysis in this case has the same complexity as context-bounded analysis, while it allows for significantly more coverage. However, considering networks with two pushdown systems makes stage-bounded analysis much harder. We show that for systems with precisely two pushdown systems the complexity of stage-bounded analysis is (at least) non-primitive recursive. The decidability in this case is actually still an open problem, but we can prove that for two pushdown systems and one 1-counter system the state reachability problem under stage-bounding is undecidable.

On the other hand, we prove, and this is our main result, that for networks with at most one pushdown system and any number of 1-counter systems, stage-bounded analysis is decidable, and we show that it is in NEXPTIME while it is PSPACE-hard. We establish this decidability result by a non-trivial reduction to the state reachability problem for pushdown systems with reversal-bounded counters (i.e., counters where the number of ascending and descending phases is bounded) [11], which is quite surprising since the use of the counters is unrestricted in the original system. Detailed proofs are omitted here for want of space and may be found in the full version of this paper.

Related work: Several bounding concepts have been considered in the literature in the last few years such as context-bounding and phase-bounding [12]. Stage-bounded analysis strictly generalizes context-bounded analysis, while it is incomparable with phase-bounding which is based on restricting accesses to stacks (i.e., push and pop operations by different processes in each phase) rather than restricting accesses to the shared memory. Another work based on restricting the access to stacks is for instance [1]. Again, the results there are incomparable with those we present here.

In [2], acyclic networks of communicating pushdown systems are considered. While such an acyclic network can encode computations within one stage (since in a stage information flows unidirectionally from the writer to all other processes), it has been shown that switching once between acyclic communication topologies in a network is enough to get undecidability [3]. In contrast, our main result show a case where information flow can be redirected any finite number of times.

In [8], a context-bounded analysis is proposed for a model of multithreaded programs with counters based on multi-pushdown systems with reversal bounded counters. The results of that paper are incomparable with ours since they concern different models and different analyses, and they are established using different techniques, though both works show reductions to reachability in pushdown systems with reversal bounded counters.

In [7, 6], networks of pushdown systems with non-atomic writes are considered. Atomic read-writes cannot be implemented in that model, which means that only a weak form of synchronization is possible. It is shown that for a fixed number of processes the reachability problem is undecidable, while in the parametrized case the problem becomes decidable [7] and is PSPACE-complete [6]. In contrast, our results hold even for the case where atomic read-writes are allowed and show a decidable case for a fixed number of processes. The parametrized case in the context of our stage-bounded analysis is still open and cannot be reduced to the problem considered in [7, 6].

2 Preliminaries

Let Σ be a finite alphabet. We use Σ^* and Σ^+ to denote the set of all finite words and non-empty finite words respectively over Σ; and use ϵ to denote the empty word. We also write Σ_ϵ for $\Sigma \cup \{\epsilon\}$. We let $|w|$ denote the length of the word w. A language is a (possibly infinite) set of words. Consider a word $w = a_1 \cdots a_n$ over Σ. We define the reverse word of w as $w^R := a_n \cdots a_1$. We write $w(i)$ for a_i and $w[1..j]$ for $w(1) \cdots w(j)$. We use $w_1 \cdot w_2$ or simply $w_1 w_2$ to denote the concatenation of two given words w_1 and w_2.

We define $\preceq \Sigma^* \times \Sigma^*$ to be the *sub-word relation*: For every $u = a_1 \cdots a_n \in \Sigma^*$ and $v = b_1 \cdots b_m \in \Sigma^*$, $u \preceq v$ if and only if there are $i_1, \ldots, i_n \in \{1, \ldots, m\}$ such that $i_1 < i_2 < \cdots < i_n$ and for every $j : 1 \leq j \leq n$, $a_j = b_{i_j}$. For $w \in \Sigma^*$, $\Gamma \subseteq \Sigma$, we define $w|_\Gamma \in \Gamma^*$ for the projection of w on the Γ. Given a language $L \subseteq \Sigma^*$, the *upward closure* (resp. *downward closure*) of L (w.r.t. $\preceq$) is the set $L\uparrow$ (resp. $L\downarrow$) containing all the words $w \in \Sigma^*$ such that there is a word $u \in L$ and $u \preceq w$ (resp. $w \preceq u$). Given a word $w = a_1 a_2 \cdots a_n$, we define stuttering $St(w) = a_1^+ a_2^+ \cdots a_n^+$.

3 Shared-memory Concurrent Pushdown Systems

In this section we describe the SCPS model which consists of a set of pushdown systems that communicate through shared memory.

3.1 Pushdown Systems and Counter Systems

A *pushdown system (PDS)* is a tuple $(Q, \Gamma, \Sigma, \delta, s)$ where Q is the set of states, Γ is the stack alphabet, Σ is the tape alphabet, $s \in Q$ is the initial state and δ is the transition relation. We assume that Γ contains the special bottom of stack element $\perp$. The transition set δ is a subset of $Q \times \Gamma_\epsilon \times \Sigma \times \Gamma_\epsilon \times Q$ with the restriction that if $\tau = (q, \alpha, m, \beta, q') \in \delta$ then either $\alpha = \beta = \perp$ (*emptiness test*) or $\alpha, \beta \in \Gamma_\epsilon \setminus \{\perp\}$ and $|\alpha\beta| \leq 1$. When $\beta \neq \epsilon$ and $\alpha = \epsilon$ ($\alpha \neq \epsilon$ and $\beta = \epsilon$) we say τ is a *push* (resp. *pop*) transition.

The configuration of a PDS $A = (Q, \Gamma, \Sigma, \delta, s)$ is a pair (q, γ) with $q \in Q$ and $\gamma \in (\Gamma \setminus \{\perp\})^* \perp$. The *initial configuration* is the pair $(s, \perp)$. The transition relation $\xrightarrow{a}_A$, $a \in \Sigma$, on the set of configurations is defined as follows:

1. $(q, \alpha\gamma) \xrightarrow{a}_A (q', \gamma)$ if $(q, \alpha, a, \epsilon, q') \in \delta$. Pop move.
2. $(q, \gamma) \xrightarrow{a}_A (q', \beta\gamma)$ if $(q, \epsilon, a, \beta, q') \in \delta$. Push move.
3. $(q, \gamma) \xrightarrow{a}_A (q', \gamma)$ if $(q, \epsilon, a, \epsilon, q') \in \delta$. Internal move.
4. $(q, \perp) \xrightarrow{a}_A (q', \perp)$ if $(q, \perp, a, \perp, q') \in \delta$. Emptiness test.

We omit the A and write $\xrightarrow{a}$ when A is clear from the context. We write $(q, \gamma) \xrightarrow{w} (q', \gamma')$ for $w = a_1 \dots a_n \in \Sigma^*$ to mean that there is a sequence of transitions of the form $(q, \gamma) = (q_0, \gamma_0) \xrightarrow{a_1} (q_1, \gamma_1) \xrightarrow{a_2} \dots \xrightarrow{a_{n-1}} (q_{n-1}, \gamma_{n-1}) \xrightarrow{a_n} (q_n, \gamma_n) = (q', \gamma')$. Given a configuration c, we use $L(A, c)$ to denote the set of words w such that $(s, \perp) \xrightarrow{w} c$. Given two configurations c_1, c_2, we use $L(A, c_1, c_2)$ to denote the set of words w such that $c_1 \xrightarrow{w} c_2$.

A *counter system* (CS) is a pushdown system where $\Gamma = \{\alpha, \perp\}$. In this case we refer to the push and pop moves as *increment* and *decrement* and the emptiness test as *zero test*. Finally, if the stack alphabet $\Gamma = \{\perp\}$ the PDS is just a finite state system (FSS).

3.2 Concurrent Pushdown System with Shared Memory

We consider a set of pushdown systems communicating with each other via a shared memory. The contents of this memory is drawn from a finite set $\boldsymbol{M}$ and in each move one of the pushdown systems from the collection either writes a value from $\boldsymbol{M}$ into the shared memory or reads the current value in the shared memory.

Let $\mathcal{O}_{\boldsymbol{M}} = \{\boldsymbol{!m}, \boldsymbol{?m} \mid \boldsymbol{m} \in \boldsymbol{M}\}$ denote the tape alphabet, where $\boldsymbol{!m}$ denotes writing the value $\boldsymbol{m}$ to the shared memory while $\boldsymbol{?m}$ refers to reading the value $\boldsymbol{m}$ from the shared memory. The value $\boldsymbol{m_0} \in \boldsymbol{M}$ is the initial memory value. We shall write $\mathcal{R}_{\boldsymbol{M}}$ ($\mathcal{W}_{\boldsymbol{M}}$) for the set $\{\boldsymbol{?m} \mid \boldsymbol{m} \in \boldsymbol{M}\}$ ($\{\boldsymbol{!m} \mid \boldsymbol{m} \in \boldsymbol{M}\}$). A Shared-memory Concurrent Pushdown System (SCPS) over a set of memory values $\boldsymbol{M}$ is a tuple $(\mathcal{I}, \mathcal{P}, \boldsymbol{m_0})$ where $\mathcal{I}$ is a finite set of indices and $\mathcal{P} = \{P_i \mid i \in \mathcal{I}\}$ is an $\mathcal{I}$-indexed collection of pushdown systems $P_i = (Q_i, \Gamma_i, \mathcal{O}_{\boldsymbol{M}}, \delta_i, s_i)$.

A configuration of a SCPS $(\mathcal{I}, \mathcal{P}, \boldsymbol{m_0})$ over $\boldsymbol{M}$ is a triple $(\boldsymbol{q}, \boldsymbol{\gamma}, \boldsymbol{m})$ where $\boldsymbol{q}$ assigns an element of Q_i to each $i \in \mathcal{I}$, $\boldsymbol{m} \in \boldsymbol{M}$ is the contents of the shared memory and $\boldsymbol{\gamma}$ assigns an element of $((\Gamma_i \setminus \{\perp\})^* \cdot \{\perp\})$ to each $i \in \mathcal{I}$ such that $(\boldsymbol{q}(i), \boldsymbol{\gamma}(i))$ is a configuration of P_i. The initial configuration of the system is the triple $(\boldsymbol{s}, \boldsymbol{\perp}, \boldsymbol{m_0})$ where for each i, $(\boldsymbol{s}(i), \boldsymbol{\perp}(i))$ is the initial configuration of P_i.

The transition relation $\xrightarrow{op}_i$, $op \in \mathcal{O}_{\boldsymbol{M}}, i \in \mathcal{I}$, relating configurations of the SCPS is defined as follows: $(\boldsymbol{q}, \boldsymbol{\gamma}, \boldsymbol{m}) \xrightarrow{op}_i (\boldsymbol{q'}, \boldsymbol{\gamma'}, \boldsymbol{m'})$ iff $(\boldsymbol{q}(i), \boldsymbol{\gamma}(i)) \xrightarrow{op} (\boldsymbol{q'}(i), \boldsymbol{\gamma'}(i))$, $(\boldsymbol{q}(j), \boldsymbol{\gamma}(j)) = (\boldsymbol{q'}(j), \boldsymbol{\gamma'}(j))$ for $j \neq i$ and further one of the following holds
1. $op = \boldsymbol{?m}$ and $\boldsymbol{m'} = \boldsymbol{m}$ (a read operation)
2. $op = \boldsymbol{!m'}$ (a write operation)

We write $\xrightarrow{op}$ for $\biguplus_{i \in \mathcal{I}} \xrightarrow{op}_i$. This naturally extends to a relation $\xrightarrow{w}$ for $w \in \mathcal{O}^*_{\boldsymbol{M}}$. We write $(\boldsymbol{q}, \boldsymbol{\gamma}, \boldsymbol{m}) \longrightarrow (\boldsymbol{q'}, \boldsymbol{\gamma'}, \boldsymbol{m'})$ if there is some $w \in \mathcal{O}^*_{\boldsymbol{M}}$ such that $(\boldsymbol{q}, \boldsymbol{\gamma}, \boldsymbol{m}) \xrightarrow{w} (\boldsymbol{q'}, \boldsymbol{\gamma'}, \boldsymbol{m'})$

▶ **Remark 1.** *Communication via shared memory is* unreliable*. This is because, the reader may skip some of the values (lossiness) while reading some values multiple times (stuttering). It is easy to eliminate stuttering errors, unidirectionally, using a protocol that writes a delimiter between every adjacent pair of values. Eliminating lossiness would require acknowledgements from the reader, arranged using some a protocol (for eg. see Theorem 3).*

▶ **Remark 2.** *It is easy to extend the set of operations to include $\boldsymbol{\tau}$, indicating that the memory is not accessed, and $\boldsymbol{?m_1!m_2}$ indicating an* atomic *operation that reads the value $\boldsymbol{m_1}$ from the memory and replaces it with $\boldsymbol{m_2}$. None of the undecidability or lower-bounds proved in this paper require these instructions and the stage-bounded decidability arguments extend easily if they are included. Hence, they have been omitted here.*

4 The Reachability Problem for SCPS

A natural and important problem in verification is the control state *reachability problem*, which asks whether a particular (*bad*) control state can be reached via some execution of the system. Formally, given a SCPS $(\mathcal{I}, \mathcal{P}, \boldsymbol{m_0})$ and a configuration $(\boldsymbol{q}, \boldsymbol{\gamma}, \boldsymbol{m})$ determine whether $(\boldsymbol{s}, \bot, \boldsymbol{m_0}) \longrightarrow (\boldsymbol{q}, \boldsymbol{\gamma}, \boldsymbol{m})$. Unfortunately, this problem is undecidable.

▶ **Theorem 3.** *The reachability problem for SCPS is undecidable even when* $|\boldsymbol{M}| = 2$, $|\mathcal{I}| = 2$ *and both the pushdown systems in* $\mathcal{P}$ *are counter systems.*

Proof. (sketch) Fix a 2-counter machine A with two counters named 1 and 2. We construct a SCPS with two components, and we refer to them as the *master* and the *slave*. The master simulates the control state of A as well as the values of the counter 1. The job of the slave is to maintain the value of the counter 2. We show that it is possible for the master to communicate, unambiguously, a value from the set $\{1, 2, 3\}$ to the slave, standing for increment, decrement and test for zero respectively and also obtain a confirmation from the slave if it is able to complete the operation successfully. First we show how the master may communicate a single value from $\{1, 2, 3\}$ and then extend it to sequences of such values.

Assume that the memory contains the value 0. To communicate the value $i \in \{1, 2, 3\}$ the master carries out the sequence of operations $(\mathbf{!1?0})^i.(\mathbf{?1!0})^i$ on the memory. The slave guesses the value j being sent and executes a sequence of the form $(\mathbf{?1!0})^j.(\mathbf{!1?0})^j$. There are three possibilities and we analyze each of them:

1. $i = j$. In this case there is exactly one successful interleaving of the two sequences and it is of the form $(\mathbf{!1}_m\mathbf{?1}_s\mathbf{!0}_s\mathbf{?0}_m)^i.(\mathbf{!1}_s\mathbf{?1}_m\mathbf{!0}_m\mathbf{?0}_s)^i$ (where, the component involved in the memory operation is marked as a subscript). Further it leaves the memory with the value 0.
2. $i < j$. In this case, the interleaved runs reaches a deadlock after a sequence of the form $(\mathbf{!1}_m\mathbf{?1}_s\mathbf{!0}_s\mathbf{?0}_m)^i$ where both components wait for the other one to write the value 1 to proceed further.
3. $i > j$. In this case, the interleaved runs reaches a deadlock after a sequence of the form $(\mathbf{!1}_m\mathbf{?1}_s\mathbf{!0}_s\mathbf{?0}_m)^i(\mathbf{!1}_m\mathbf{!1}_s + \mathbf{!1}_s\mathbf{!1}_m)$ and both components wait for the other one to write the value 0 to proceed further.

Since all unsuccessful runs deadlock, it follows that the protocol can be repeated for any sequence of values and the system will either deadlock or succeed in communicating the sequence correctly to the slave. Finally, handling the confirmation from the slave to the master is also easy. After guessing the next operation the slave attempts to carry out the operation and only on success does it enter the protocol described above. The details are easy to formalize. ◀

5 Stage-bounded Computations

We introduce hereafter the concept of stage-bounding. We divide a run into segments, called stages, where in each stage at most one component is allowed to write on the shared memory while there is no restriction on the number of readers. We emphasize that there is no restriction placed on the number of writes or the number of context switches between the different components nor is there any restriction on the accesses to stacks during a stage. We then place an a priori bound on the number of stages in the run. Formally

▶ **Definition 4.** Let $\rho = \boldsymbol{c_0} \xrightarrow{op_1}_{p_1} \boldsymbol{c_1} \xrightarrow{op_2}_{p_2} \dots \boldsymbol{c_{n-1}} \xrightarrow{op_n}_{p_n} \boldsymbol{c_n}$ be a run of the SCPS $(\mathcal{I}, \mathcal{P}, \boldsymbol{m})$. We say that ρ is a p-run if for all $1 \leq i \leq n$, $p_i = p$ whenever $op_i \in \mathcal{W}_{\boldsymbol{M}}$. That is, all the write transitions are contributed by the same process p.

We say that ρ is a 1-stage run if it is a p-run for some $p \in \mathcal{I}$ and a run ρ is a k-stage run if we may write $\rho = \boldsymbol{c_0} \xrightarrow{w_1} \boldsymbol{c_1} \xrightarrow{w_2} \ldots \boldsymbol{c_{k-1}} \xrightarrow{w_k} \boldsymbol{c_k}$ such that each $\boldsymbol{c_{i-1}} \xrightarrow{w_i} \boldsymbol{c_i}$ is a 1-stage run for each $1 \leq i \leq n$.

Stage-bounded Reachability Problem: Given a SCPS $(\mathcal{I}, \mathcal{P}, \boldsymbol{m_0})$, an integer k and a configuration $(\boldsymbol{q}, \boldsymbol{\gamma}, \boldsymbol{m})$ determine whether there is a k-stage run $(\boldsymbol{s}, \boldsymbol{\perp}, \boldsymbol{m_0}) \longrightarrow (\boldsymbol{q}, \boldsymbol{\gamma}, \boldsymbol{m})$.

▶ **Remark 5.** *Stage-bounding restricts the ability to eliminate lossiness in shared-memory communication via acknowledgements (see Remark 1). This makes the undecidability of stage-bounded reachability non-trivial, in particular the proof of Theorem 7, and it is also crucial for Theorem 8.*

5.1 Stage bounded reachability for Communicating FSS

We next show that stage-bounding is relevant even when all components of the SCPS are finite-state. In this case stage bounded reachability problem is indeed easier than the unrestricted reachability problem.

▶ **Theorem 6.** *The reachability problem for an SCPS where every component is a FSS is* PSPACE-COMPLETE *while the stage bounded reachability problem for SCPS where every component is a FSS is* NP-COMPLETE.

Proof. (sketch) When there is no bound on the number of stages, it is easy to see that an SCPS with n FSS components is equivalent to the product (intersection) of n FSS and hence the reachability problem is PSPACE-COMPLETE.

To solve the stage bounded reachability problem, we show that it suffices to consider runs where in each stage every one of the readers participates in at most $|A_i|$ transitions, where A_i is the ith automaton. We then use this to show that in addition we may restrict to runs where in each stage the writer participates in at most $O((\sum_i |A_i|)^2)$ transitions. This immediately yields a polynomial bound on the length of stage-bounded computations to be explored to solve the reachability problem and hence a decision procedure in NP. ◀

5.2 Undecidability of Bounded-Stage Reachability

Unfortunately, stage bounding does not lead to decidability in the general case. We can indeed prove that SCPS with two pushdown systems and one 1-counter system are able to encode the computation of any Turing machine.

▶ **Theorem 7.** *The* 3*-stage reachability problem for SCPS consisting of two pushdown systems and one counter system is undecidable.*

Proof. We will reduce the halting problem for Turing machines to the stage-bounded reachability problem in a SCPS with two pushdown systems and one counter. We refer to the two pushdowns as the *generator* and the *replayer.* If somehow a writer and a reader could follow a protocol that ensures that every letter that is written is read exactly once then the undecidability would follow quite easily without the counter. However, doing this using shared memory in a stage bounded manner is tricky and details are as follows. In what follows we assume that stuttering errors are eliminated using a suitable delimiter (see Remark 1).

The simulation of a (potential) accepting run of the TM is carried out in 4 steps which use 3 stages in all. We fix a suitable encoding of the configurations as a word over some alphabet Γ and assume that this alphabet does not contain the symbol $\#$. In the first step, the generator writes down a (initial) configuration C_1 of the TM in its stack followed by the $\#$ symbol. While doing so, it uses the shared memory to send a value, say \$, to the counter for each letter in C_1. The counter counts the number of such values. Since stuttering has been eliminated, the value of the counter c_1 is $\leq |C_1|$ at the end of this step.

In step 2, the generator guesses a sequence of configurations C_2,C_3,... C_n ending in an accepting configuration, writes them down, separated by $\#$s, in its stack. It also writes the same sequence to the memory, as it is generated, which in turn is read by the replayer and copied on to its stack. At the end of step 2, the contents of the generator's stack is $C_n^R \# C_{n-1}^R \# \ldots C_1^R$ while that of the replayer is $y = D_m^R \# D_{n-1}^R \# \ldots D_1^R$, $m \leq n-1$ and y is a subword of $C_n^R \# C_{n-1}^R \# \ldots C_2^R$. It indicates the end of this stage by writing some suitable value to the memory which signals the end of this stage to the replayer and the counter. In all we have used one stage so far.

In step 3, the counter sends its value c_1 to the generator using the shared memory by writing c_1 copies of some fixed value ending with some special value to indicate the completion of this sequence. The generator removes one non-$\#$ symbol from his stack for each such value. At the end of this sequence of operations if the top of stack is not a $\#$ the generator will reject this run. Thus, a successful completion of this step will mean that $|C_n| \leq c_1$ and thus, $|C_n| \leq |C_1|$. At the end of this step, the contents of the generator's stack is $C_{n-1}^R \# C_{n-2}^R \# \ldots C_1^R$ and the counter is empty. This constitutes the second stage.

In the last step, the replayer removes the contents of its stack one element at a time and writes the removed value to the shared memory for the generator to read. It writes a special end marker at the end of the sequence and enters an accepting state. The sequence read by the generator would therefore be of the form $z = E_p^R \# E_{p-1}^R \# \ldots E_1^R$ (followed by the end marker) where $p \leq m \leq n-1$. Clearly z is a subword of y. The generator, as it reads E_p^R removes symbols from its stack verifying that C_{n-1} may be reached in one step from the configuration E_p (we write $E_p \Rightarrow C_{n-1}$ to indicate this), entering a reject state if either this is false or if they are not of the same length. It then repeats this procedure for E_{p-1} and C_{n-2} and so on. It enters an accepting state only if it empties its stack at the end of the entire sequence.

Observe that if the generator reaches its accepting state then p has to be $n-1$, $|E_{n-1}| = |C_{n-1}|$, ..., $|E_1| = |C_1|$ and $E_{n-1} \Rightarrow C_{n-1}$, ..., $E_1 \Rightarrow C_1$. Further, since z is a subword of y, y is a subword of $C_n^R \# C_{n-1}^R \# \ldots C_2^R$ and $p = n-1$, we have $E_i \preceq D_i \preceq C_{i+1}$ for all $1 \leq i \leq n-1$. Thus,

$$|C_1| = |E_1| \leq |C_2| = |E_2| \leq \ldots \leq |C_{n-1}| = E_{n-1} \leq |C_n|\,.$$

But $|C_n| \leq |C_1|$ and thus,

$$|C_1| = |E_1| = |C_2| = |E_2| \ldots = |C_{n-1}| = |E_{n-1}| = |C_n|\,.$$

Therefore $E_1 = C_2$, $E_2 = C_3$, ..., $E_{n-1} = C_n$ and the result follows. ◀

6 Decidability for single pushdown plus counters

We present in this section our main result:

▶ **Theorem 8.** *The stage bounded reachability problem for SCPS with at most one pushdown system is in* NEXPTIME.

Basically, we show that each counter system can be simulated by an exponential sized bounded-reversal counter system thus reducing the problem to reachability in a *pushdown automaton*[1] (PDA) with reversal bounded counters (which is known to be in NP).

The proof proceeds in three steps. The first step is applicable to any SCPS. In this step, we eliminate the shared memory, decouple the different pushdown systems as a collection of pushdown automata (PDA) and reduce the reachability problem for the SCPS to the emptiness of the intersection of these PDAs. (This problem, in general, is undecidable, but we will be able to restrict ourselves to the case where the PDAs are of a restricted variety.) In a shared memory system, the sequence of values written by the writer in a stage is not transmitted with precision to the reader as the reader may miss some values while reading others multiple times and this is what permits the decoupling.

We fix an SCPS $S = (\mathcal{I}, \mathcal{P}, \boldsymbol{m_0})$ over the set of memory values $\boldsymbol{M}$ where $\mathcal{P} = \{P_i \mid i \in \mathcal{I}\}$ is an $\mathcal{I}$ indexed collection of pushdown systems $P_i = (Q_i, \Gamma_i, \mathcal{O}_{\boldsymbol{M}}, \delta_i, s_i)$, for the rest of this section. For the moment, consider one stage runs where $p \in \mathcal{I}$ identifies the writer. Suppose we are interested in the existence of one stage runs starting at the configuration $((s_i)_{i\in\mathcal{I}}, (\rho_i)_{i\in\mathcal{I}}, \boldsymbol{m})$ and ending at some configuration $((q_i)_{i\in\mathcal{I}}, (\gamma_i)_{i\in\mathcal{I}}, \boldsymbol{m'})$. Now, consider the languages L_i, $i \in \mathcal{I}$, $i \neq p$, defined as

$$L_i = \{\boldsymbol{m_1 m_2 \ldots m_n} \mid \boldsymbol{?m_1?m_2}\ldots\boldsymbol{?m_n} \in L(P_i, (s_i, \rho_i), (q_i, \gamma_i))\}$$

and L_p given by

$$\{\boldsymbol{m.m_1 \ldots m_n.m'} \mid \boldsymbol{?m}^*\boldsymbol{!m_1?m_1}^*\boldsymbol{!m_2}\ldots\boldsymbol{!m_n?m_n}^*\boldsymbol{!m'?m'}^* \in L(P_p, (s_p, \rho_p), (q_p, \gamma_p))\}\,.$$

Then, the existence of a one stage run from $((s_i)_{i\in\mathcal{I}}, (\rho_i)_{i\in\mathcal{I}}, \boldsymbol{m})$ to $((q_i)_{i\in\mathcal{I}}, (\gamma_i)_{i\in\mathcal{I}}, \boldsymbol{m'})$ (with p as the writer) is equivalent to the non-emptiness of

$$St(L_p)\downarrow \cap \bigcap_{i\neq p} L_i \uparrow\ .$$

Moreover, the languages $St(L_p)\downarrow$ and $L_i\uparrow$ are easily realized as the languages of PDAs A_p and A_i constructed from the PDSs P_p and P_i respectively. These automata maintain the stack and control state of the PDS they simulate as well. Observe that the language accepted by these PDAs are either upward or downward closed.

We are however interested in k stage runs where the identity of the writer (and hence the closures to be applied) changes with the stage. Further, across the stage boundaries, we have to preserve the control state and stacks of each component as well as the content of the shared memory.

It is useful to work with a fixed sequence τ of length k over $\mathcal{I}$ identifying the writers in the k stages. Let $\tau := p_1, p_2, \ldots, p_k$, $p_i \in \mathcal{I}$ be such a sequence. Let $(\boldsymbol{s}, \perp, \boldsymbol{m_0})$ and $(\boldsymbol{q}, \boldsymbol{\gamma}, \boldsymbol{m})$ be the initial and target configurations of the SCPS and we wish to determine if there is a k stage run consistent with τ that goes from the initial to the target configuration. In this case, the pushdown automaton A_i^τ plays the role played by the automaton A_i in the one stage setting. It simulates P_i and its runs break up into k parts, where in the jth part it applies either a stuttering downward closure or upward closure to the behaviour of P_i depending on whether $j = \tau(j)$ or not. Notice that A_i^τ maintains the control state and stack of P_i. A_i^τ also makes explicit the boundary points between stage i and stage $i+1$ by using a letter of

[1] We plan to use "automata" instead of systems when they are used as language generators and to avoid ambiguity with the components of the SCPS.

the form $(\boldsymbol{m}, i)$ (instead of just $\boldsymbol{m}$). These marker letters allow us to *synchronize* the stage boundaries of the different A_i^τ's. Further, these markers are also used to ensure that the contents are of the memory are correctly transferred across stages. We formalize these ideas below.

We use $\boldsymbol{M}_i$ (resp. $\boldsymbol{M}_\tau$) to denote $\boldsymbol{M}^* \cdot (\boldsymbol{M} \times \{i\})$ for all $i \in [1..k]$ (resp. $\bigcup_{i\in[1..k]} \boldsymbol{M}_i \cup \boldsymbol{M}$).

▶ **Lemma 9.** *For every $p \in \mathcal{I}$, we can construct, in polynomial time in $|S|$, a PDA A_p^τ over the stack alphabet Γ_p, with a target configuration c_p such that*

- *if $w \in L(A_p^\tau, c_p)$ then $w \in \boldsymbol{M}_1 \cdot \boldsymbol{M}_2 \cdots \boldsymbol{M}_k$. (unambiguous breakup)*
- *if $w \in L(A_p^\tau, c_p)$ and $w = w_1 w_2 \ldots w_k$ with $w_i \in \boldsymbol{M}_i$ for all $1 \le i \le k$ then $w'_1.w'_2.\ldots.w'_k \in L(A_p^\tau, c_p)$ for all $w'_1, \ldots w'_k$ such that, for all i, $w'_i \in \boldsymbol{M}_i$ and either $p = \tau(i)$ and $w'_i \in St(w_i)\downarrow$ or $p \neq \tau(i)$ and $w'_i \in (w_i \uparrow \cap \boldsymbol{M}_i)$. (closure)*
- *There is a k stage run from $(\boldsymbol{s}, \perp, \boldsymbol{m_0})$ to $(\boldsymbol{q}, \boldsymbol{\gamma}, \boldsymbol{m})$ with $\tau(i)$ as the writer in the ith stage iff $\bigcap_{p\in\mathcal{I}} L(A_p^\tau, c_p) \neq \emptyset$. (decoupling)*

In the second step we exploit the fact that the language of each A_p^τ is a finite unambiguous concatenation of languages that are upward or downward closed. Towards this we first state two propositions which explain the importance of closures.

▶ **Proposition 10** (Downward closure of CFLs [5]). *Given a pushdown automaton P and two configurations c_i, c_f, we can construct, in time and space at most exponential in size of P, c_i and c_f, a FSA A with two configurations c'_i and c'_f such that $L(A, c'_i, c'_f) = L(P, c_i, c_f)\downarrow$.*

▶ **Proposition 11** (Upward closure of CFLs). *Given a pushdown automaton P and two configurations c_i, c_f, we can construct, in time and space at most exponential in size of P, c_i and c_f, a FSA A with two configurations c'_i and c'_f such that $L(A, c'_i, c'_f) = L(P, c_i, c_f)\uparrow$.*

This means that, if we are dealing with a single stage then we may replace the PDA A_i, $i \in \mathcal{I}$, described earlier, by exponential sized finite automata B_i, $i \in \mathcal{I}$ (for all i, including the writer p). Thus we have reduced the problem to the emptiness of the intersection for FAs. However the k stage case is somewhat more complex. This is because, as A_i^τ switches from one stage to the next, it has to preserve the configuration of P_i (i.e. the contents of the stack) as well as the contents of the memory. While this is trivial when A_i^τ is a pushdown, it is not possible to do this using finite number of states. However, all is not lost as we may convert A_i^τ into a $2k$-*turn PDA* B_i^τ. A run of pushdown automaton is said to be 1-*turn* if the stack height of the sequence of configurations is either uniformly non-increasing (does not involve a push move) or non-decreasing (does not involve a pop move). A k-*turn* run is concatenation of k sequences of 1-*turn* runs. A k-*turn* PDA is one which only allows at most k-*turn* runs (see [9]).

We explain the ideas behind the construction of B_i^τ now. Let us fix a pushdown automaton A. For any $\gamma \in \Gamma^*\perp$ we say that a run χ from (q, ρ) to (q', ρ') is a γ-run if γ is the longest suffix of ρ that appears as a suffix of the contents of the stack in every configuration along the run χ. Observe that, this implies that γ must be a suffix of ρ and ρ' and further there is a configuration in χ whose stack content is exactly γ. (Observe that every run from (q, ρ) to (q', ρ') is a γ-run for some (unique) suffix γ of ρ). We write $L_\gamma(c, c')$ to refer to the set of words accepted on γ-runs from c to c'. Let $c = (q, \rho)$ to $c' = (q', \rho')$. Then, $L(c, c')$, the set of words accepted on runs from c to c' is

$$\{x.y \mid x \in L_\gamma(c, (q'', \gamma)), y \in L_\gamma((q'', \gamma), c'), \gamma \text{ a suffix of } \rho, q'' \in Q\}.$$

We write $Cl(L)$ to refer to the upward or downward closure of L when the identity of the closure does not matter. Thus $Cl(L(c, c'))$ is

$$\{x.y \mid x \in Cl(L_\gamma(c, (q'', \gamma))), y \in Cl(L_\gamma((q'', \gamma), c')), \gamma \text{ a suffix of } \rho, q'' \in Q\}.$$

For each $\alpha \in \Gamma$ and $q_1, q_2 \in Q$, we let:

$$L^-_\alpha(q_1, q_2) = \{w \mid (q_1, \alpha\bot) \xrightarrow{w} (q_2, \bot) \text{ without using emptiness tests } \}$$
$$L^+_\alpha(q_1, q_2) = \{w \mid (q_1, \bot) \xrightarrow{w} (q_2, \alpha\bot) \text{ without using emptiness tests } \}$$
$$L^\bot(q_1, q_2) = \{w \mid (q_1, \bot) \xrightarrow{w} (q_2, \bot)\}$$

We can see that the language $L_\gamma(c, (q'', \gamma))$ (resp. $L_\gamma((q'', \gamma), c')$) can be rewritten as a concatenation of the following languages $L^-_{\alpha_1}(q, q_1) \cdot L^-_{\alpha_2}(q_1, q_2) \cdots L^-_{\alpha_\ell}(q_{\ell-1}, q'') \cdot L$ with $\rho = \alpha_1\alpha_2 \cdots \alpha_\ell\gamma$ (resp. $L \cdot L^+_{\alpha_1}(q'', q_1) \cdot L^+_{\alpha_2}(q_1, q_2) \cdots L^+_{\alpha_\ell}(q_{\ell-1}, q')$ with $\rho' = \alpha_\ell\alpha_{\ell-1} \cdots \alpha_1\gamma$) and $L = \{\epsilon\}$ if $\gamma \neq \bot$ and $L = L^\bot(q'', q'')$ otherwise.

Hence, any word $w \in Cl(L(c, c'))$ can be rewritten as the concatenation of three words (i.e., $w = w_1w_2w_3$). The first word w_1 is in $Cl(L^-_{\alpha_1}(q, q_1)) \cdot Cl(L^-_{\alpha_2}(q_1, q_2)) \cdots Cl(L^-_{\alpha_\ell}(q_{\ell-1}, q''))$ for some letters $\alpha_1\alpha_2 \cdots \alpha_\ell$ and stack content γ such that $\rho = \alpha_1\alpha_2 \cdots \alpha_\ell\gamma$. The second word w_2 is in $Cl(L^\bot(q'', q''))$ if $\gamma = \bot$, and in $\{\epsilon\}$ otherwise. The last word w_3 is in $Cl(L^+_{\alpha'_1}(q'', q'_1)) \cdot Cl(L^+_{\alpha'_2}(q'_1, q'_2)) \cdots Cl(L^+_{\alpha'_m}(q'_{m-1}, q'))$ for some letters $\alpha'_1\alpha'_2 \cdots \alpha'_m$ such that $\rho' = \alpha'_m\alpha'_{m-1} \cdots \alpha'_1\gamma$.

Furthermore, the languages $L^-_\alpha(q_1, q_2)$, $L^+_\alpha(q_1, q_2)$ and $L^\bot(q_1, q_2)$ are context-free and their upward and downward closures are effectively regular (see Propositions above) and so let $B^-_\alpha(q_1, q_2)$, $B^+_\alpha(q_1, q_2)$ and $B^\bot(q_1, q_2)$ be finite automata recognizing $Cl(L^-_\alpha(q_1, q_2))$, $Cl(L^+_\alpha(q_1, q_2))$ and $Cl(L^\bot(q_1, q_2))$ respectively.

Now, we describe the PDA B: It uses the same stack alphabet as A and maintains the *current* state of A as part of its local state. Its run consists of 3 phases. The first phase consists in generating the first word w_1 by repeating the following a number of times: it guesses a triple (q_1, α, q_2), verifies that q_1 is the current state of A, pops the top of stack and verifies that it is indeed α, simulates a run of $B^-_\alpha(q_1, q_2)$ and then changes the current state of A to q_2. In the optional second phase, it will generate the second word w_2 by verifying that the stack is empty, guessing a pair (q_1, q_2), checking that q_1 is the current state of A, simulating a run of $B^\bot(q_1, q_2)$ and then changing the current state of A to q_2. The third stage consists of generating the word w_3 by repeating the following a number of times: it guesses triples of the form (q_1, α, q_2), verifies that q_1 is the current state of A, pushes α on to the stack, simulates a run of $B^+_\alpha(q_1, q_2)$ and then changes current state of A to q_2. Observe that these runs involve only two turns one during the pop phase and one during the push phase. The language of B while starting at the stack configuration γ with q as the current state of P and ending with stack configuration γ' and q' as the current state of P is the language $Cl(L(c, c'))$. But, if $L(c, c')$ is already closed then $Cl(L(c, c')) = L(c, c')$. Thus, in this case B simulates a (arbitrary) run of A from c to c' using a single turn run and further maintains the configuration reached by A at the end of this run.

Since the language of A^τ_p in each stage is either upward or downward closed, by concatenating k appropriately chosen copies of the automata B (with correct closures in each stage depending on the sequence τ) we construct a $2k$-*turn* PDA B^τ_p that has the same language as A^τ_p as stated by the following Lemma

▶ **Lemma 12.** *For every $p \in \mathcal{I}$, it is possible to construct, in exponential time in the size of A^τ_p, a $2k$ turn PDA B^τ_p and a configuration c'_p, such that $L(B^\tau_p, c'_p) = L(A^\tau_p, c_p)$.*

Unfortunately, the emptiness of the intersection of even two 2-*turn* PDAs is undecidable, as can be seen from an easy reduction from the Post's correspondence problem (PCP). The situation is quite different when the PDAs are counters. In fact, we can show:

▶ **Lemma 13.** *Let k be a natural number. Let A_1 be a $2k$ turn PDA and $A_2, \ldots, A_n$ a sequence of $2k$-turn counter automata. Let c_i be a configurations of A_i for all $i : 1 \leq i \leq n$. Then, the problem of checking whether $L(A_1, c_1) \cap \cdots \cap L(A_n, c_n)$ is not empty can be decided in nondeterministic time that is polynomial in the size of A_i and k, and exponential in n.*

The proof of Lemma 13 is done by a reduction to the state reachability problem for pushdown systems with reversal-bounded counters where each counter is allowed a bounded number of alternation between *modes*. (The latter problem is known to be NP-COMPLETE [10].) A counter *mode* is a run of the system where the performed sequence of transitions on this counter consists, apart from internal transitions, only of increment transitions or only of decrement transitions or only of zero test transitions. A pushdown system with k reversal-bounded counters is pushdown system augmented with counters where any run decomposes into at most k modes for each counter. Then, the reduction consists simply in constructing a pushdown system S augmented with counters that simulates the synchronous product of $A_1, \ldots, A_n$ while observing that any run of a $2k$-*turn* counter automaton performs at most $3k$ modes in S (Note that zero tests are counted as a reversal (*mode*) but are not counted as a turn). These arguments can be formalized to obtain a complete proof.

Thus we have reduced the k stage bounded reachability problem of an SCPS consisting of one PDS and $(n-1)$ counter systems to exponentially[2] many instances of the emptiness problem for the intersection of a polynomial sized $2k$-turn PDA and $(n-1)$ $2k$-turn counter automata of exponential size (Lemma 9 and Lemma 12). We have also shown that each instance of the latter problem can be decided in NEXPTIME (Lemma 13). Combining these results yields to a NEXPTIME upper-bound and a proof of Theorem 8.

7 Lower Bounds for the Stage-bounded Reachability Problem

We have so far shown that the stage bounded reachability problem for systems with at least two pushdowns and one counter is undecidable while it can be decided in NEXPTIME for systems containing at most one pushdown. This problem remains open for SCPS with exactly two pushdowns. In the following, we show that even if it is decidable its complexity cannot be primitive recursive.

The *regular post embedding* problem is the following: Let Σ and Γ be two alphabets. Given two functions $f : \Sigma \to \Gamma^+$ and $g : \Sigma \to \Gamma^+$, extended homomorphically to Σ^+, and a regular language $R \subseteq \Sigma^+$, does there exist a $w \in R$ such that $f(w) \preceq g(w)$? As shown in [4], this problem is decidable but cannot be solved by any algorithm with primitive recursive complexity. We reduce the regular post embedding problem to the stage-bounded reachability problem for SCPS with two pushdowns to obtain the following theorem

▶ **Theorem 14.** *The 2-stage bounded reachability problem for SCPS with two pushdowns cannot be solved by any algorithm whose complexity is primitive recursive.*

The emptiness problem for the intersection of a collection of n finite automata is known to be PSPACE complete and we reduce this problem to the n-stage bounded reachability problem for SCPS with n counters to obtain the following theorem

▶ **Theorem 15.** *The stage-bounded reachability problem for SCPS consisting only of counter systems is* PSPACE-HARD.

[2] The exponential blow-up comes from the guess of the sequence of writers.

8 Conclusion

We have introduced a new concept for bounding the analysis of shared memory concurrent systems. This concept is based on bounding the number of switches between writes by different processes to the shared memory, without restricting the way reads can be performed by any of the processes in the system.

Stage-bounding allows to improve significantly the behaviors coverage w.r.t. context-bounding. We have shown that for networks of finite-state systems, the complexity of stage-bounded analysis is NP-complete, as for context-bounding. In practice, this implies that this analysis can be implemented using a complete bounded model-checking with a polynomial bound. In the case of networks of infinite-state systems, we have mainly shown that the state reachability problem in networks of counter systems with a shared store is decidable under stage-bounding, and that the same still holds for networks with one additional pushdown.

Several questions remain open. One of them is closing the gap between the known upper and lower bounds on the complexity. Also, the case two pushdown systems is open, although we know that even if it is decidable, it would be with a high complexity. Finally, an interesting open question is whether it is possible to generalize our decidability result to dynamic or parametrized networks of counter systems, by considering that each component in the network is allowed to be the single writer in a bounded number of stages.

References

1 Mohamed Faouzi Atig. Model-checking of ordered multi-pushdown automata. *Logical Methods in Computer Science*, 8(3), 2012.

2 Mohamed Faouzi Atig, Ahmed Bouajjani, and Tayssir Touili. On the reachability analysis of acyclic networks of pushdown systems. In *CONCUR*, volume 5201 of *Lecture Notes in Computer Science*, pages 356–371. Springer, 2008.

3 Mohamed Faouzi Atig and Tayssir Touili. Verifying parallel programs with dynamic communication structures. In *CIAA*, volume 5642 of *Lecture Notes in Computer Science*, pages 145–154. Springer, 2009.

4 Pierre Chambart and Ph. Schnoebelen. Post embedding problem is not primitive recursive, with applications to channel systems. In *FSTTCS*, volume 4855 of *Lecture Notes in Computer Science*, pages 265–276. Springer, 2007.

5 Bruno Courcelle. On constructing obstruction sets of words. *Bulletin of the EATCS*, 44:178–186, 1991.

6 Javier Esparza, Pierre Ganty, and Rupak Majumdar. Parameterized verification of asynchronous shared-memory systems. In *CAV*, volume 8044 of *Lecture Notes in Computer Science*, pages 124–140. Springer, 2013.

7 Matthew Hague. Parameterised pushdown systems with non-atomic writes. In *FSTTCS*, volume 13 of *LIPIcs*, pages 457–468. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2011.

8 Matthew Hague and Anthony Widjaja Lin. Synchronisation- and reversal-bounded analysis of multithreaded programs with counters. In *CAV*, volume 7358 of *Lecture Notes in Computer Science*, pages 260–276. Springer, 2012.

9 J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley series in computer science. Addison-Wesley, 1979.

10 Rodney R. Howell and Louis E. Rosier. An analysis of the nonemptiness problem for classes of reversal-bounded multicounter machines. *J. Comput. Syst. Sci.*, 34(1):55–74, 1987.

11 Oscar H. Ibarra. Reversal-bounded multicounter machines and their decision problems. *J. ACM*, 25(1):116–133, 1978.

12 Salvatore La Torre, Parthasarathy Madhusudan, and Gennaro Parlato. A robust class of context-sensitive languages. In *LICS*, pages 161–170. IEEE Computer Society, 2007.

13 Shaz Qadeer and Jakob Rehof. Context-bounded model checking of concurrent software. In *TACAS*, volume 3440 of *Lecture Notes in Computer Science*, pages 93–107. Springer, 2005.