Form Methods Syst Des
DOI 10.1007/s10703-014-0207-y

Budget-bounded model-checking pushdown systems

Parosh Aziz Abdulla · Mohamed Faouzi Atig · Othmane Rezine · Jari Stenman

Abstract We address the verification problem for concurrent programs modeled as multi-pushdown systems (MPDS). In general, MPDS are Turing powerful and hence come along with undecidability of all basic decision problems. Because of this, several subclasses of MPDS have been proposed and studied in the literature (Atig et al. in LNCS, Springer, Berlin, 2005; La Torre et al. in LICS, IEEE, 2007; Lange and Lei in Inf Didact 8, 2009; Qadeer and Rehof in TACAS, LNCS, Springer, Berlin, 2005). In this paper, we propose the class of bounded-budget MPDS, which are restricted in the sense that each stack can perform an unbounded number of context switches only if its depth is below a given bound, and a bounded number of context switches otherwise. We show that the reachability problem for this subclass is PSPACE-complete and that LTL-model-checking is EXPTIME-complete. Furthermore, we propose a code-to-code translation that inputs a concurrent program P and produces a sequential program P' such that running P under the budget-bounded restriction yields the same set of reachable states as running P'. Moreover, detecting (fair) non-terminating executions in P can be reduced to LTL-Model-Checking of P'. By leveraging standard sequential analysis tools, we have implemented a prototype tool and applied it on a set of benchmarks, showing the feasibility of our translation.

Keywords Concurrent pushdown systems · Verification problems · LTL-model-checking · Reachability problem · Recursive programs

P. A. Abdulla · M. F. Atig (✉) · O. Rezine · J. Stenman
Department of Information Technology, Uppsala University,
Box 337, 751 05 Uppsala, Sweden
e-mail: mohamed_faouzi.atig@it.uu.se

P. A. Abdulla
e-mail: parosh@it.uu.se

O. Rezine
e-mail: othmane.rezine@it.uu.se

J. Stenman
e-mail: jari.stenman@it.uu.se

Published online: 25 April 2014

1 Introduction

In the last few years, a lot of effort has been devoted to the verification problem for models of concurrent programs (see, e.g., [2,4,6,9,14,23,26,29,33,37,39]). Pushdown systems have been proposed as an adequate formalism to describe sequential programs with procedure calls. Therefore, it is natural to model recursive concurrent programs as Multi-PushDown Systems (MPDS for short). However, MPDS are in general Turing powerful, and hence all the basic decision problems are undecidable for them [38]. To overcome this barrier, several subclasses of multi-pushdown systems have been proposed and studied in the literature (e.g., [1,2,5,23,26,37]). The main goals of these works are (1) to explore the largest possible state space of the modeled concurrent program, and (2) to retain the decidability of some properties such as the reachability problem.

Context-bounding has been proposed in [37] as a suitable technique for the analysis of MPDS. The idea is to consider only runs of the system that can be divided into a given number of contexts, where in each context pop and push operations are exclusive to one stack. The state space which may be explored is still unbounded in the presence of recursive procedure calls, but the context-bounded reachability problem is NP-complete even in this case. Empirically, it has been shown that many concurrency errors, such as data races and atomicity violations, manifest themselves in executions with only a few contexts [33].

Another way to regain decidability is to consider depth-bounded verification for MPDS where the maximal possible depth (or size) of each stack is bounded by a given constant. In this case, the reachability problem becomes PSPACE-complete. However, since the explored state space is bounded, this approach is more suitable for detecting shallow bugs [11]. In fact, bounding the stack depth provides a completeness result for the case where the threads are modeled as finite-state systems (this is not the case for the context-bounded analysis).

In this paper, we generalize both context-bounded analysis and depth-bounded verification by introducing the class of *budget-bounded* MPDS. Intuitively, for each thread (or stack), we associate two natural numbers k, d such that each thread can perform at most k consecutive context switches unless its stack depth goes below the given bound d. More precisely, each thread is given a budget b of contexts. The thread then operates in two modes, I and II. In mode I, the stack depth of the thread is less than or equal to d, while in mode II it is strictly above d. The budget of the thread is unbounded in mode I, i.e., $b = \infty$. In other words, the thread is allowed to perform any number of context switches while it is in mode I. As soon as the stack depth of the thread grows above d, the thread enters mode II and its budget b is set to k. Each time the thread performs a context switch in mode II, its budget b is decremented by one. The thread leaves mode II in one of two ways: either it consumes all its budget (its budget b becomes negative) in which case the thread will be blocked; or the stack depth of the thread becomes d in which case it enters mode I and its budget is reset to unbounded ($b = \omega$) again.

We consider two problems for budget-bounded MPDS. First, we show that the reachability problem is PSPACE-complete. The lower bound is proved by a straightforward reduction from the non-emptiness test of the intersection of a finite set of regular languages (which is PSPACE-complete). To prove the upper-bound, we show that it is possible to reduce, in polynomial time, the reachability problem for a budget-bounded MPDS to the non-emptiness test for the synchronous product of a finite set of depth-bounded pushdown automata (which is PSPACE-complete). Second, we consider the LTL model checking problem. We show that this problem is EXPTIME-complete for budget-bounded MPDS.

In the second part of the paper, we investigate the issue of defining a code-to-code translation that inputs a concurrent program P and produces a sequential program P' such that

running P under the budget-bounded restriction yields the same set of reachable states as running P'. In other words, we have reduced the problem of verifying (an under-approximation of) the concurrent P to that of verifying a sequential program P'. In fact, the only source of abstraction in our translation is the fact that we limit the behavior of P when its stack depth exceeds the given limit. In particular, our translation preserves the data domains of the original program in the sense that P and P' have the same features (e.g., recursive procedure calls, type of data structures). We show that the translation can be performed using additional copies of the shared variables and local variables. More importantly, the fact that P' is a sequential program means that our translation allows us to use existing analysis and verification tools designed for sequential programs in order to perform the same kind of analysis and verification for concurrent programs under the budget-bounded restriction. To show its use in practice, we have implemented our approach and applied it on several examples, using the three back end tools MOPED [16], ESBMC [12], and CBMC [11]. We compare our results to the ones obtained using concurrent verification tools, namely ESBMC [12] and POIROT [28]. In our experiments, bugs (i.e. violations of state invariants) appear for small bounds.

Furthermore, we describe how our bounded-budget translation can be used to check *liveness* properties. Checking liveness properties is not relevant for context-bounded MPDS. The reason is that context-bounding does not give a chance for every thread to execute infinitely often, since bounding the number of context-switches necessarily implies that, eventually, all threads but one remain switched out forever. We apply an extension of our translation to check termination under fairness for three examples programs taken from [3]. For this, we use SPIN as a backend tool.

1.1 Related work

Our model is inspired by the work of Finkel and Sangnier [17], where they propose an extension of reversal-bounded counter machines, restricting each counter to a finite number of alternations between the increasing and decreasing modes when its value goes beyond a given bound.

As mentioned earlier, several decidable classes of multi-pushdown systems have been proposed [2,23,26,37]. The closest model to multi-pushdown systems with budgets is Scope-bounded Multistack PushDown Systems (SMPDS for short) [26] where each symbol in a stack can be popped only if it has been pushed within a bounded number of context switches. We can show that the reachability problem for SMPDS can be reduced to its corresponding one for bounded-budget MPDS where the value of the stack depth is 0. This can be done by assuming that a stack symbol that will never be popped in the context of [26] will not be pushed into the stack. Thus, each symbol that is pushed into the stack should be removed within k context-switches. On the other hand, simulating bounded-budget MPDS by SMPDS does not seem to be straightforward without an exponential explosion (to encode the content of each stack up to the stack depth bound).

Our code-to-code translation follows the line of research on compositional reductions from concurrent to sequential programs [14,24,25,30]. Recently, La Torre and Parlato have proposed in [27] a sequentialization for SMPDS where for each SMPDS, they construct an equivalent single-stack pushdown system that faithfully simulates the behavior of each thread. However, the proposed sequentialization has not been implemented and so we were not able to compare it with our translation. Moreover, the two translation schemes were developed independently and simultaneously [34].

This work is an extension of the published paper [1].

2 Preliminaries

In this section, we fix some basic definitions and notations. We assume here that the reader is familiar with language and automata theory in general. For more details, the reader is referred to specialized books such as [13], [18], [19], and [21].

Notations. Let $\mathbb{N}$ denote the set of natural numbers, and $\mathbb{N}_\omega$ denote the set $\mathbb{N} \cup \{\omega\}$ (ω represents the first limit ordinal). For every $i, j \in \mathbb{N}_\omega$ such that $i \leq j$, we use $[i..j]$ to denote the set $\{k \in \mathbb{N}_\omega \mid i \leq k \leq j\}$.

Let Σ be a finite alphabet. We denote by Σ^* (resp. Σ^ω) the set of all finite (resp. infinite) words (over Σ, and by ϵ the empty word. A language is a (possibly infinite) set of words. Let u be a word over Σ. The length of u is denoted by $|u|$ (we assume that $|\epsilon| = 0$ and $|u| = \omega$ if $u \in \Sigma^\omega$). For words $u_1, u_2 \in \Sigma^*$, we use $u_1 \cdot u_2$ (or simply $u_1 u_2$) to denote the concatenation of u_1 and u_2. For a natural number $k \in \mathbb{N}$, we use $\Sigma^{\leq k}$ to denote the set of all words of length at most k.

Let L be a set of (possibly infinite) words over Σ and let $w \in \Sigma^*$ be a word. We define $w \cdot L = \{w \cdot u \mid u \in L\}$. We define the *shuffle* operator $\sqcup\!\sqcup$ over two words inductively as $\sqcup\!\sqcup(\epsilon, w) = \sqcup\!\sqcup(w, \epsilon) = \{w\}$ and $\sqcup\!\sqcup(a \cdot u', b \cdot v') = a \cdot (\sqcup\!\sqcup(u', b \cdot v') \cup b \cdot (\sqcup\!\sqcup(a \cdot u', v')$. Given two sets (possibly infinite) of words L_1 and L_2, we define their shuffle as $\sqcup\!\sqcup(L_1, L_2) = \bigcup_{u \in L_1, v \in L_2} \sqcup\!\sqcup(u, v)$. The shuffle operator for multiple sets can be extended analogously.

Finite-State Automata A *finite-state automaton* is a tuple $\mathcal{A} = (Q, \Sigma, \Delta, I, F)$ where: (1) Q is the finite non-empty set of states, (2) Σ is the input alphabet, (3) $\Delta \subseteq (Q \times (\Sigma \cup \{\epsilon\}) \times Q)$ is the transition relation, (4) $I \subseteq Q$ is the set of initial states, and (5) $F \subseteq Q$ is the set of final states. The language of finite words accepted (or recognized) by $\mathcal{A}$ is denoted by $L(\mathcal{A})$. We may also interpret the set F as a Büchi acceptance condition, and we denote by $L^\omega(\mathcal{A})$ the language of infinite words accepted by $\mathcal{A}$ The size of $\mathcal{A}$ is defined by $|\mathcal{A}| = (|Q| + |\Sigma|)$.

Pushdown Automata. A pushdown automaton is defined as a tuple $\mathcal{P} = (Q, \Sigma, \Gamma, \Delta, I, F)$ where: (1) Q is a finite non-empty set of states, (2) Σ is an input alphabet, (3) Γ is a stack alphabet, (4) Δ is a finite set of transition rules of the form $(q, u) \xrightarrow{a} (q', u')$ where $q, q' \in Q, a \in \Sigma \cup \{\epsilon\}, u, u' \in \Gamma^*$ such that $|u| + |u'| \leq 1$, (5) $I \subseteq Q$ is a set of initial states, and (6) $F \subseteq Q$ is a set of final states. The size of $\mathcal{P}$ is defined by $|\mathcal{P}| = |Q| + |\Sigma| + |\Gamma|$.

A *configuration* of $\mathcal{P}$ is a tuple (q, w) where $q \in Q$ is the current state, and $w \in \Gamma^*$ is the stack content. We define the binary relation $\rightarrow_{\mathcal{P}}$ between configurations as follows: $(q, u \cdot w) \xrightarrow{a}_{\mathcal{P}} (q', u' \cdot w)$ iff $(q, u) \xrightarrow{a} (q', u')$.

We use the transition relation $\rightarrow^*_{\mathcal{P}}$ to denote the transitive closure of $\rightarrow_{\mathcal{P}}$: Let $\Sigma \in \Sigma^*$ be an input word and c and c' two configurations of $\mathcal{P}$, we have $c \xrightarrow{\Sigma}{}^*_{\mathcal{P}} c'$ if and only if one of the following cases holds: (1) $\sigma = \epsilon$ and $c = c'$, or (2) there are a sequence of configurations $c_0, c_1, \ldots, c_n$ of $\mathcal{P}$ and a sequence of input symbols $a_1, a_2, \ldots, a_n \in \Sigma \cup \{\epsilon\}$ such that: (i) $c = c_0$ and $c' = c_n$, (ii) $\sigma = a_1 a_2 \cdots a_n$, and (iii) $c_{j-1} \xrightarrow{a_j}_{\mathcal{P}} c_j$ for all $j \in [1..n]$.

The language $L(\mathcal{P})$ accepted by $\mathcal{P}$ is defined by the set of finite words $\sigma \in \Sigma^*$ such that $(q_{\mathsf{init}}, \epsilon) \xrightarrow{\sigma}{}^*_{\mathcal{P}} (q_{\mathsf{final}}, \epsilon)$ for some $q_{\mathsf{init}} \in I$ and $q_{\mathsf{final}} \in F$. Similar to the case of finite-state automata, even for pushdown automata we can interpret the acceptance set F as a Büchi acceptance condition and we denote by $L^\omega(\mathcal{P})$ the language of infinite words accepted by $\mathcal{P}$.

Let $d \in \mathbb{N}$. We define the transition relation $\rightarrow_{>d}$ between configurations of $\mathcal{P}$ as follows: $(q, w) \xrightarrow{a}_{>d} (q', w')$ if and only if $(q, w) \xrightarrow{a}_{\mathcal{P}} (q'', w')$, $|w'| > d$ and $|w| > d$. Intuitively, the transition relation $\rightarrow_{>d}$ can be performed only if the stack depth of the initial and target

configuration is at least $d+1$. The transition relation $\rightarrow^*_{>d}$ is the reflexive transitive closure of $\rightarrow_{>d}$.

Similarly, we can define the transition relation $\rightarrow_{\leq d}$ between configurations of $\mathcal{P}$ as follows: $(q,w)\xrightarrow{a}_{\leq d}(q',w')$ if and only if $(q,w)\xrightarrow{a}_{\mathcal{P}}(q',w')$, $|w'|\leq d$ and $|w|\leq d$. Intuitively, the transition relation $\rightarrow_{\leq d}$ can only be performed when the stack depth of both the initial and target configurations are at most d. The transition relation $\rightarrow^*_{\leq d}$ is the reflexive transitive closure of $\rightarrow_{\leq d}$.

Given $d,k\in\mathbb{N}$ and an input word $\sigma\in\Sigma^*$, we define the relation $\xrightarrow{\sigma}_{(k,d)}$ between configurations of depth d as follows: $(q,w)\xrightarrow{\sigma}_{(k,d)}(q',w')$ if and only if there are two configurations (q_1,w_1) and (q_2,w_2), two input symbol $a_1,a_2\in\Sigma\cup\{\epsilon\}$ and a word $\sigma'\in\Sigma^*$ such that: (1) $(q,w)\xrightarrow{a_1}_{\mathcal{P}}(q_1,w_1)\xrightarrow{\sigma'}{}^*_{>d}(q_2,w_2)\xrightarrow{a_2}(q',w')$, (2) $\sigma=a_1\cdot\sigma'\cdot a_2$, (3) $|\sigma|\leq k$, (4) $w'=w$, and (5) $|w|=d$. This means that the pushdown automaton can only read k consecutive input symbols without its stack depth going below the bound d.

Let $L_{(k,d)}(\mathcal{P})$ denote the set of finite words $\sigma\in\Sigma^*$ such that there are a sequence of configurations $c_0,c_1,\ldots,c_n$ and a sequence of input words $\sigma_1,\sigma_2,\ldots,\sigma_n$ where (1) c_0 is of the form (q_0,ϵ) with $q_0\in I$, (2) c_n is of the form (q_n,ϵ) with $q_n\in F$, (3) $\sigma=\sigma_1\sigma_2\cdots\sigma_n$, and (4) for every $i\in[1..n]$, we have $c_{i-1}\xrightarrow{\sigma_i}_{(k,d)}c_i$ or $c_{i-1}\xrightarrow{\sigma_i}{}^*_{\leq d}c_i$ holds. We call $L_{(k,d)}(\mathcal{P})$ the (k,d)-bounded language of $\mathcal{P}$.

We define the language $L^{\omega}_{(k,d)}(\mathcal{P})$ to be the set of infinite words $\sigma\in\Sigma^{\omega}$ such that there are an infinite sequence of configurations $c_0,c_1,\ldots$ and an infinite sequence of finite input words $\sigma_1,\sigma_2,\ldots$ where: (1) c_0 is of the form (q_0,ϵ) with $q_0\in I$, (2) c_i is of the form (q_i,w_i) for all $i\in\mathbb{N}$, (3) there is an infinite sequence of indices $i_1,i_2\ldots$ such that $q_{i_j}\in F$ for all $j\geq 1$, (4) $\sigma=\sigma_1\sigma_2\cdots$, and (4) for every $i\in[1..n]$, we have $c_{i-1}\xrightarrow{\sigma_i}_{(k,d)}c_i$ or $c_{i-1}\xrightarrow{\sigma_i}{}^*_{\leq d}c_i$ holds.

We also define the language $L_{(-1,d)}(\mathcal{P})$ to be the set of words $\sigma\in\Sigma^*$ such that $(q_{\mathsf{init}},\epsilon)\xrightarrow{\sigma}{}^*_{\leq d}(q_{\mathsf{final}},\epsilon)$ where $q_{\mathsf{init}}\in I$ and $q_{\mathsf{final}}\in F$. Intuitively, the set $L_{(-1,d)}(\mathcal{P})$ (or simply $L_d(\mathcal{P})$ when it is clear from the context) contains all words accepted through runs of $\mathcal{P}$ where the stack depth is always bounded by d.

In a similar way, we can define the language $L^{\omega}_{(-1,d)}(\mathcal{P})$ accepting set of infinite words with the Büchi acceptance condition.

Lemma 1 *Let $d,k\in\mathbb{N}$ be two natural numbers and let $\mathcal{P}$ be a pushdown automaton. Then, it is possible to construct, in polynomial time, a pushdown automaton $\mathcal{P}'$ such that $L_{k+d+3}(\mathcal{P}')=L_{(k,d)}(\mathcal{P})$ (resp. $L^{\omega}_{k+d+3}(\mathcal{P}')=L^{\omega}_{(k,d)}(\mathcal{P})$).*

Proof In the following we give the proof of Lemma 1 for the case of finite words. The proof can be extended in a straightforward manner to the case of infinite words.

Let us first prove the following lemma:

Lemma 2 *Let $k\in\mathbb{N}$ be a natural number and $\mathcal{P}$ be a pushdown automaton. Then, it is possible to construct, in polynomial time, a pushdown automaton $\mathcal{P}'$ such that $L_{k+2}(\mathcal{P}')=L(\mathcal{P})\cap\Sigma^{\leq k}$.*

Proof Let us first recall some basic results about context-free languages.

A *context-free grammar* (CFG) G is a tuple $(\mathcal{X},\Sigma,R,S)$ where $\mathcal{X}$ is a finite non-empty set of *variables* (or *nonterminals*), Σ is an alphabet of *terminals*, $R\subseteq\big(\mathcal{X}\times(\mathcal{X}^2\cup\Sigma)\big)\cup(S\times\{\epsilon\})$ a finite set of *productions* (the production (X,w) may also be denoted by $X\rightarrow w$), and $S\in\mathcal{X}$

is a start variable. The size of G is defined by $|G| = (|\mathcal{X}| + |\Sigma|)$. Observe that the form of the productions is restricted, but it has been shown in [31] that every CFG can be transformed, in polynomial time, into an equivalent grammar of this form.

Given strings $u, v \in (\Sigma \cup \mathcal{X})^*$ we say $u \Rightarrow_G v$ if there exists a production $(X, w) \in R$ and some words $y, z \in (\Sigma \cup \mathcal{X})^*$ such that $u = yXz$ and $v = ywz$. We use $\Rightarrow_G^*$ for the reflexive transitive closure of $\Rightarrow_G$. We define the context-free language generated by $L(G)$ as $\{w \in \Sigma^* \mid S \Rightarrow_G^* w\}$.

Let $k \in \mathbb{N}$. A derivation α given by $\alpha :\stackrel{\text{def}}{=} u_0 \Rightarrow_G u_1 \Rightarrow_G \cdots \Rightarrow_G u_n$ is *k-bounded* if $|u_i| \leq k$ for all $i \in [1..n]$. We denote by $L^{(k)}(G)$ the subset of $L(G)$ such that for every $w \in L^{(k)}(G)$ there exists a k-bounded derivation $S \Rightarrow_G^* w$. We call $L^{(k)}(G)$ the *k-bounded approximation* of $L(G)$.

Lemma 3 *Given a context-free grammar G and $k \in \mathbb{N}$, then it is possible to construct, in polynomial time, a pushdown automaton $\mathcal{P}'$ such that $L_{k+2}(\mathcal{P}') = L^{(k)}(G)$.*

Proof It is well-known that for any context-free grammar? $G = (\mathcal{X}, \Sigma, R, S)$, it is possible to construct in polynomial time a pushdown automaton $\mathcal{P}_1$ such that $L(\mathcal{P}_1) = L(G)$ [21]. Formally, the pushdown automaton $\mathcal{P}_1 = (Q_1, \Sigma, \Gamma_1, \Delta_1, \{q_{\mathsf{init}}\}, \{q_{\mathsf{final}}\})$ is defined as follows:

- *States:* For each production t of the form $X \to YZ$ with $Y, Z \in \mathcal{X}$, the pushdown automaton $\mathcal{P}_1$ has two states $tmp_1(t)$ and $tmp_2(t)$. Moreover, $\mathcal{P}_1$ has four special states q_{init}, q, q', and q_{final}.
- *Stack alphabet:* The pushdown automaton $\mathcal{P}_1$ has $\mathcal{X} \cup \{\bot\}$ as stack alphabet where $\bot$ is a special stack symbol used to denote the bottom of the stack.
- *Transition relation:* To simulate a derivation of the context-free grammar G, the pushdown automaton $\mathcal{P}_1$ proceeds as follows:
  - Initial Phase: The pushdown automaton $\mathcal{P}_1$ pushes the symbol $\bot$ in the stack followed by $S \in \mathcal{X}$ and then moves to the state q'. Formally, $\mathcal{P}_1$ has the following two transitions $(q_{\mathsf{init}}, \epsilon) \xrightarrow{\epsilon} (q, \bot)$ and $(q, \epsilon) \xrightarrow{\epsilon} (q', S)$.
  - *Simulation Phase:* For each production t of the form $X \to YZ$ with $X, Y \in \mathcal{X}$ (resp. $X \to a$ with $a \in (\Sigma \cup \{\epsilon\})$) of G, the pushdown automaton $\mathcal{P}_1$ has the following transitions: $(q', X) \xrightarrow{\epsilon} (tmp_1(t), \epsilon)$, $(tmp_1(t), \epsilon) \xrightarrow{\epsilon} (tmp_2(t), Z)$, and $(tmp_2(t), \epsilon) \xrightarrow{\epsilon} (q', Y)$ (resp. $(q', X) \xrightarrow{a} (q', \epsilon)$).
  - Final Phase: The pushdown automaton $\mathcal{P}_1$ checks if the topmost stack symbol is $\bot$ and its current state is q' in which case $\mathcal{P}_1$ moves to the accepting state q_{final}. Formally, $\mathcal{P}_1$ contains the transition rule $(q', \bot) \xrightarrow{\epsilon} (q_{\mathsf{final}}, \epsilon)$.

It is easy to see that the derivation tree $\mathcal{T}$ of the grammar G is simulated by the pushdown automaton $\mathcal{P}_1$ in a leftmost way. Moreover, during the simulation of $\mathcal{T}$, the maximal depth of the stack of $\mathcal{P}_1$ is bounded by the maximal number of nodes in each path in the derivation tree $\mathcal{T}$ incremented by one (due to the bottom stack symbol).

Since the context-free grammar G is in the normal form, we know that any k-bounded derivation has a derivation tree $\mathcal{T}$ in which the number of leaves is at most k. Moreover, any path in this derivation tree $\mathcal{T}$ has at most $k + 1$ nodes (non-terminal symbols). This implies that $w \in L^{(k)}(G)$ iff $w \in L_{k+2}(\mathcal{P}_1)$ and $|w| \leq k$.

Then, let $\mathcal{P}'$ be the pushdown automaton recognizing the language $L(\mathcal{P}_1) \cap \Sigma^{\leq k}$ such that $L_{k+2}(\mathcal{P}') = L_{k+2}(\mathcal{P}_1) \cap \Sigma^{\leq k}$. Observe that such a pushdown automaton can be constructed in polynomial time in $|\mathcal{P}_1|$ and k by keeping track, in the control state of $\mathcal{P}'$, of the number

of input symbols read by $\mathcal{P}_1$ so far and bounding this number by k. Finally it is easy to see that $L_{k+2}(\mathcal{P}') = L^{(k)}(G)$. □

To prove Lemma 2, we will make use of the fact that for every pushdown automaton $\mathcal{P}$, it is possible to construct, in polynomial time in the size of $\mathcal{P}$, a context-free grammar G such that $L^{(k)}(G) = L(\mathcal{P}) \cap \Sigma^{\leq k}$ [21]. Moreover, we can assume that this context-free grammar is in the normal form (based on the result in [31] showing that every context-free grammar can be transformed, in polynomial time, into an equivalent grammar in the normal form). Then, we can apply Lemma 3 to construct, in polynomial time, a pushdown automaton $\mathcal{P}'$ such that $L_{k+2}(\mathcal{P}') = L^{(k)}(G)$. Hence, we have $L_{k+2}(\mathcal{P}') = L^{(k)}(G) = L(\mathcal{P}) \cap \Sigma^{\leq k}$. □

Let $\mathcal{P} = (Q, \Sigma, \Gamma, \Delta, I, F)$ be a pushdown automaton. To prove Lemma 1, we construct, in polynomial time, a pushdown automaton $\mathcal{P}'$ such that $L_{k+d+3}(\mathcal{P}') = L_{(k,d)}(\mathcal{P})$ as follows: The pushdown automaton $\mathcal{P}'$ mimics the pushdown automaton $\mathcal{P}$ if the current stack depth of $\mathcal{P}$ is less than or equal to d. Moreover, $\mathcal{P}'$ keeps track of its current stack depth in its control state (up to $d + k + 3$). If the current depth of the stack of $\mathcal{P}$ (and therefore $\mathcal{P}'$) is precisely d and $\mathcal{P}$ performs a push transition t from a state q, then the pushdown automaton $\mathcal{P}'$ guesses the return state q' (when the stack of $\mathcal{P}$ is again of depth d). Then, $\mathcal{P}'$ starts to mimic the pushdown automaton $\mathcal{P}_{(q,t,q')}$ (constructed using Lemma 2) from a pushdown automaton $\mathcal{P}'_{(q,t,q')}$ with a bound k. The pushdown automaton $\mathcal{P}'_{(q,t,q')}$ is constructed from $\mathcal{P}$ by setting its initial state to q and its final state to q'. Moreover, we constrain the pushdown automaton $\mathcal{P}'_{(q,t,q')}$ such that the only possible initial firable transition is precisely the push transition t. If the stack depth of $\mathcal{P}'$ is again d after the simulation of $\mathcal{P}_{(q,t,q')}$ and the current state of $\mathcal{P}_{(q,t,q')}$ is accepting, then $\mathcal{P}'$ resumes the simulation of $\mathcal{P}$ from the state q'. In the following, we give the formal definition of $\mathcal{P}'_{(q,t,q')}$ and $\mathcal{P}'$.

Let us assume that the push transition t is of the form $(q, \epsilon) \xrightarrow{a} (q'', u)$ with $u \in \Gamma$. The pushdown automaton $\mathcal{P}'_{(q,t,q')}$ is defined by the following tuple $((Q \times \{(q, t, q')\}) \cup \{q, q'\}, \Gamma, \Sigma, \Delta'_{(q,t,q')}, \{q\}, \{q'\})$ such that $\Delta'_{(q,t,q')}$ is defined as follows: (1) $\Delta'_{(q,t,q')}$ contains the following transition $(q, \epsilon) \xrightarrow{a} ((q'', (q, t, q')), \epsilon)$, (2) For each transition of $\mathcal{P}$ of the form $(p, v) \xrightarrow{a'} (p', v')$, $\Delta'_{(q,t,q')}$ contains $((p, (q, t, q')), v) \xrightarrow{a'} ((p', (q, t, q')), v')$, and (3) For each transition of $\mathcal{P}$ of the form $(p, u) \xrightarrow{a'} (q', \epsilon)$, $\Delta'_{(q,t,q')}$ contains $((p, (q, t, q')), u) \xrightarrow{a'} (q', \epsilon)$.

Then, the relation between the pushdown automata $\mathcal{P}'_{(q,t,q')}$ and $\mathcal{P}$ is given by the following lemma:

Lemma 4 *Let $\sigma \in \Sigma^*$. Then, for every $q, q' \in Q$, we have $\sigma \in \bigcup_{t \in \Delta} L(\mathcal{P}'_{(q,t,q')})$ if and only if there are two configurations (q_1, w_1) and (q_2, w_2), two input symbols $a_1, a_2 \in \Sigma \cup \{\epsilon\}$, and a word $\sigma' \in \Sigma^*$ such that $(q, \epsilon) \xrightarrow{a_1}_{\mathcal{P}} (q_1, w_1) \xrightarrow{\sigma'}{}^{*}_{>0} (q_2, w_2) \xrightarrow{a_2}_{\mathcal{P}} (q', \epsilon)$ and $\sigma = a_1 \sigma' a_2$.*

Let $\mathcal{P}_{(q,t,q')} = (Q_{(q,t,q')}, \Sigma, \Gamma_{(q,t,q')}, \Delta_{(q,t,q')}, I_{(q,t,q')}, F_{(q,t,q')})$ be the pushdown automaton constructed from $\mathcal{P}'_{(q,t,q')}$ using Lemma 2 with the bound k (i.e., $L_{k+2}(\mathcal{P}_{(q,t,q')}) = L(\mathcal{P}'_{(q,t,q')}) \cap \Sigma^{\leq k}$).

By Lemma 2 and Lemma 4, the union of the pushdown automata of the form $\mathcal{P}_{(q,t,q')}$ (when their stack depth is bounded by $k+2$) accept exactly the words σ of length less than k generated by the run of $\mathcal{P}$ of the form $(q, \epsilon) \xrightarrow{a_1}_{\mathcal{P}} (q_1, w_1) \xrightarrow{\sigma'}{}^{*}_{>0} (q_2, w_2) \xrightarrow{a_2}_{\mathcal{P}} (q', \epsilon)$. Hence, we have:

Lemma 5 *Let $\sigma \in \Sigma^*$. Then, $(q, w) \xrightarrow{\sigma}_{(k,d)} (q', w)$ for some $w \in \Gamma^*$ such that $|w| = d$ if and only if $\sigma \in \bigcup_{t \in \Delta} L_{k+2}(\mathcal{P}_{(q,t,q')})$.*

Let us now give the definition of the pushdown automaton $\mathcal{P}'$. Intuitively, the pushdown automata $\mathcal{P}'$ will simulate $\mathcal{P}$ if its stack depth is less than d and the pushdown automaton $\mathcal{P}_{(q,t,q')}$ otherwise. Formally, we have $\mathcal{P}' = (Q', \Sigma, \Gamma', \Delta', I \times \{0\}, F \times \{0\})$ where:

- *States:* $Q' = (\bigcup_{(q,t,q') \in Q \times \Delta \times Q} Q_{(q,t,q')} \times [d+1..(k+d+3)]) \cup (Q \times [0..d])$. A state of the form (q, i) corresponds to that the current state of $\mathcal{P}$ (resp. $\mathcal{P}_{(q,t,q')}$) is q and the current depth of the stack is i with $i \leq d$ (resp. $i \in [d+1..(k+d+3)]$).
- *Stack alphabet:* $\Gamma' = (\bigcup_{(q,t,q') \in Q \times \Delta \times Q} \Gamma_{(q,t,q')}) \cup \Gamma \cup \{\sharp\}$ where $\sharp$ is a special stack symbol not contained in $(\bigcup_{(q,t,q') \in Q \times \Delta \times Q} \Gamma_{(q,t,q')})$ and Γ. The symbol $\sharp$ is used at the position $d+1$ of the stack of $\mathcal{P}'$ to separate the stack content of $\mathcal{P}$ (stored between the position 0 and d by $\mathcal{P}'$) and the stack content of $\mathcal{P}_{(q,t,q')}$ (stored between the position $d+2$ and $d+k+3$).
- *Transition Relation:* In order to simulate $\mathcal{P}$ if the depth of the stack is less than d and $\mathcal{P}_{(q,t,q')}$ otherwise, the pushdown automaton $\mathcal{P}'$ proceeds as follows:
  - *Pop transition and stack depth $\leq d$*: For every pop transition t of the form $(q, u) \xrightarrow{a} (q', \epsilon)$ of $\mathcal{P}$ with $u \in \Gamma$ and stack depth $i \in [1..d]$, Δ' contains the transition rule $((q, i), u) \xrightarrow{a} ((q', i-1), \epsilon)$.
  - *Push transition and stack depth $< d$*: For every push transition t of the form $(q, \epsilon) \xrightarrow{a} (q', u)$ of $\mathcal{P}$ with $u \in \Gamma$ and stack depth $i \in [0..d-1]$, Δ' contains the transition rule $((q, i), \epsilon) \xrightarrow{a} ((q', i+1), u)$.
  - *Nop transition and stack depth $\leq d$*: For every nop transition t of the form $(q, \epsilon) \xrightarrow{a} (q', \epsilon)$ of $\mathcal{P}$ and stack depth $i \in [0..d]$, Δ' contains the transition rule $((q, i), \epsilon) \xrightarrow{a} ((q', i), \epsilon)$.
  - *Push transition and stack depth $= d$*: For every push transition t of the form $(q, \epsilon) \xrightarrow{a} (q'', u)$ of $\mathcal{P}$ with $u \in \Gamma$, Δ' contains the transition rule $((q, d), \epsilon) \xrightarrow{\epsilon} ((p, d+1), \sharp)$ where $p \in \bigcup_{q' \in Q} I_{(q,t,q')}$.
  - *Pop transition and stack depth $> d+1$*: For every pop transition t of the form $(p, u) \xrightarrow{a} (p', \epsilon)$ of $\mathcal{P}_{(q,t,q')}$ with $u \in \Gamma$ and stack depth $i \in [d+2..d+k+3]$, Δ' contains the transition rule $((p, i), u) \xrightarrow{a} ((p', i-1), \epsilon)$.
  - *Push transition and stack depth $\geq d+1$*: For every push transition t of the form $(p, \epsilon) \xrightarrow{a} (p', u)$ of $\mathcal{P}_{(q,t,q')}$ with $u \in \Gamma$ and stack depth $i \in [d+1..d+k+2]$, Δ' contains the transition rule $((p, i), \epsilon) \xrightarrow{a} ((p', i+1), u)$.
  - *Nop transition and stack depth $\geq d+1$*: For every nop transition t of the form $(p, \epsilon) \xrightarrow{a} (p', \epsilon)$ of $\mathcal{P}_{(q,t,q')}$ and stack depth $i \in [d+1..d+k+3]$, Δ' contains the transition rule $((p, i), \epsilon) \xrightarrow{a} ((p', i), \epsilon)$.
  - *Pop transition and stack depth $= d+1$*: For every $p \in F_{(q,t,q')}$, Δ' contains the transition rule $((p, d+1), \sharp) \xrightarrow{\epsilon} ((q', d), \epsilon)$.

Finally it is easy that any reachable configuration of $\mathcal{P}'$ has a stack depth bounded by $d+k+3$. Moreover, we have $L_{k+d+3}(\mathcal{P}') = L_{(k,d)}(\mathcal{P})$. □

Synchronization of depth-bounded pushdown automata. Given pushdown automata $\mathcal{P}_0, \ldots, \mathcal{P}_n$ and bounds $d_0, \ldots, d_n \in \mathbb{N}$, we define *the non-emptiness test of the synchronization of depth-bounded pushdown automata* as the problem of checking the emptiness of the language $L_{d_0}(\mathcal{P}_0) \cap \sqcup\!\sqcup(L_{d_1}(\mathcal{P}_1), \ldots, L_{d_n}(\mathcal{P}_n))$.

Lemma 6 *The non-emptiness test of the synchronization of depth-bounded pushdown automata is* PSPACE*-complete.*

Proof The upper-bound can be obtained by an easy reduction to the emptiness problem for a Turing machine having $(n+1)$-tapes, where each tape $i \in [0..n]$ has d_i cells. The lower bound follows by a reduction from the non-emptiness test of the intersection of several regular languages (aparticular case of depth-bounded pushdown automata), which is well-known to be PSPACE-hard.

3 Multi-pushdown systems

In this section, we recall the definition of *multi-pushdown systems*. Multi-pushdown systems (or *MPDS* for short) have a finite set of states along with a finite number of read-write memory tapes (stacks) with a last-in-first-out rewriting policy. The types of transitions that can be performed by a MPDS are: (i) pushing a symbol onto one stack, (ii) popping a symbol from one stack, or (iii) an internal action that changes the state of the automaton while keeping the stacks unchanged. Note that since we are not interested in this model as a language acceptor, it does not include an input alphabet or final states.

Definition 1 A *multi-pushdown system* (MPDS) is a tuple $\mathcal{M} = (n, Q, \Gamma, \Delta, q_{\mathsf{init}})$ where $n \geq 1$ is the number of stacks, Q is a finite set of *states*, Γ is the *stack alphabet*, $\Delta \subseteq \big(Q \times [1..n] \times Q\big) \cup \big(Q \times [1..n] \times Q \times \Gamma\big) \cup \big(Q \times \Gamma \times [1..n] \times Q\big)$ is the *transition relation*, and q_{init} is the *initial state*.

Let $q, q' \in Q$ be two states, $i \in [1..n]$ a stack index, and $\gamma \in \Gamma$ a stack symbol. A transition of the form (q, i, q') is an internal operation that changes the state from q to q' while keeping the contents of the stacks unchanged. The stack index i is included in this operation for technical reasons. A transition of the form (q, i, q', γ) corresponds to a push operation that changes the state from q to q', and adds the symbol γ to the top of the i-th stack. Finally, a transition of the form (q, γ, i, q') corresponds to a pop operation that moves the state from q to q', and removes the symbol γ from the top of the i-th stack.

A configuration c of $\mathcal{M}$ is an $(n+1)$-tuple $(q, w_1, \ldots, w_n)$ where $q \in Q$ is a state and for every $i \in [1..n]$, $w_i \in \Gamma^*$ is the content of the i-th stack. We use $State(c)$ and $Stack_i(c)$, with $i \in [1..n]$, to denote q and w_i, respectively. We denote by $c^{\mathsf{init}}_{\mathcal{M}} = (q_{\mathsf{init}}, \epsilon, \epsilon, \ldots, \epsilon)$ the initial configuration of $\mathcal{M}$.

We define the transition relation $\rightarrow_{\mathcal{M}}$ on the set of configurations as follows. For configurations $c = (q, w_1, \ldots, w_n)$ and $c' = (q', w'_1, \ldots, w'_n)$, an index $i \in [1..n]$, and a transition $t \in \Delta$, we write $c \xrightarrow{t} {}_{\mathcal{M}}\, c'$ to denote that one of the following cases holds:

- **Internal operation:** $t = (q, i, q')$ and $w'_j = w_j$ for all $j \in [1..n]$.
- **Push operation:** $t = (q, i, q', \gamma)$ for some $\gamma \in \Gamma$, $w'_i = \gamma \cdot w_i$, and $w'_j = w_j$ for all $j \in ([1..n] \backslash \{i\})$.
- **Pop operation:** $t = (q, \gamma, i, q')$ for some $\gamma \in \Gamma$, $w_i = \gamma \cdot w'_i$, and $w'_j = w_j$ for all $j \in ([1..n] \backslash \{i\})$.

A finite *computation* π of $\mathcal{M}$ from a configuration c to a configuration c' is a sequence $c_0 t_1 c_1 t_2 \cdots t_m c_m$ such that: (1) $c_0 = c$ and $c_m = c'$, and (2) $c_{i-1} \xrightarrow{t_i} {}_{\mathcal{M}}\, c_i$ for all $i \in [1..m]$; each configuration c_i is said to be *reachable* from c. We use $initial(\pi)$ and $target(\pi)$ to denote c_0 and c_m, respectively.

An infinite computation π of $\mathcal{M}$ from a configuration c is an infinite sequence $c_0 t_1 c_1 t_2 \cdots$ such that $c_0 = c$ and for every $i \in \mathbb{N}$, we have $c_{i-1} \xrightarrow{t_i}_{\mathcal{M}} c_i$.

Given a (possibly infinite) computation π of the form $c_0 t_1 c_1 t_2 \cdots$, we use $initial(\pi)$, $conf(\pi)$, $state(\pi)$, and $trace(\pi)$ to denote c_0, $c_0 c_1 \cdots$, $State(c_0)State(c_1)\cdots$, and $t_1 t_2 \cdots$, respectively.

Given a finite computation $\pi_1 = c_0 t_1 \cdots t_m c_m$ and (a possibly infinite) computation $\pi_2 = c_{m+1} t_{m+2} \cdots$, π_1 and π_2 are said to be *compatible* if $c_m = c_{m+1}$. Then, we write $\pi_1 \bullet \pi_2$ to denote the computation $\pi \stackrel{\text{def}}{=} c_0 t_1 c_1 t_2 c_2 \cdots t_m c_m t_{m+2} c_{m+2} t_{m+3} \cdots$.

In general, multi-pushdown systems are Turing powerful resulting in the undecidability of all basic decision problems [38]. However, it is possible to obtain decidability for some problems, such a control state reachability, by restricting the allowed set of behaviours [2,10,23,26,37]. In the following, we propose the class of *bounded-budget* computations of MPDS. Let k and d be two natural numbers. Intuitively, in a bounded-budget computation, each stack $i \in [1..n]$ can be active at most k consecutive contexts without its depth goes beyond a given bound d_i. A context is a run of $\mathcal{M}$ where operations are exclusive to one stack (the only active stack in that context). Next, we describe formally bounded-budget computations.

Contexts: A *context* of a stack $i \in [1..n]$ is a (possibly infinite) computation of the form $\pi = c_0 t_1 c_1 t_2 \cdots$ in which $t_j \in \Delta_i \stackrel{\text{def}}{=} \big(Q \times \{i\} \times Q\big) \cup \big(Q \times \{i\} \times Q \times \Gamma\big) \cup \big(Q \times \{i\} \times \Gamma \times Q\big)$ for all $j \in [1..|trace(\pi)|]$ and such that $1 \leq |trace(\pi)|$. Observe that every computation can be seen as the concatenation of a sequence of contexts $\pi_1 \bullet \pi_2 \bullet \cdots$.

Given a finite context π_1 and a (possibly infinite) context π_2 of the stack i, we write $\pi_1 \bullet_i \pi_2$ to denote that $Stack_i(initial(\pi_2)) = Stack_i(target(\pi_1))$ (in this case we say that π_1 and π_2 are compatible w.r.t. stack i). This notation is extended in a straightforward manner to sequence of contexts.

Let $\pi = \pi_1 \bullet \pi_2 \bullet \cdots$ be a computation where each π_j is a context. If $i_1 < i_2 < \cdots$ be the sequence of all the indices j such that π_j is a context of stack i, then we have $\pi_{i_1} \bullet_i \pi_{i_2} \bullet_i \cdots$.

A context $\pi = c_0 t_1 c_1 t_2 \cdots$ of the stack $i \in [1..n]$ is said to be of depth at most (resp. least) $d \in \mathbb{N}$ if and only if for every $j \in [0..|trace(\pi)|]$, $|Stack_i(c_j)| \leq d$ (resp. $|Stack_i(c_j)| \geq d$). The definition is extended in the straightforward manner to sequences of contexts as follows: The sequence $\pi = \pi_1 \bullet_i \pi_2 \bullet_i \cdots$ of compatible contexts w.r.t the stack i is of depth at most (resp. least) $d \in \mathbb{N}$ iff every $\pi_1, \pi_2, \ldots$ are of depth at most (resp. least) d.

Blocks: A *block* ρ of a stack $i \in [1..n]$ of size $m \in \mathbb{N}$ and depth $d \in \mathbb{N}$ ($(k,d) - block$ for short) is a sequence of compatible finite contexts of the form: (1) $c_0 t_0 \cdot \pi_0 \cdot t_1 c_1$ if $m = 0$, and (2) $c_0 t_0 \cdot \pi_0 \bullet_i \pi_1 \bullet_i \cdots \bullet_i \pi_m \cdot t_1 c_1$ otherwise, such that $|Stack_i(c_0)| = |Stack_i(c_1)| = d$ and π_j is a context of depth at least $d+1$ for all $j \in [0..m]$.

Budget-bounded computations: Let k and d be two natural numbers. Intuitively, in a budget-bounded computation, we associate with each stack $i \in [1..n]$, a budget of contexts k and depth bound d such that if we consider a point in the computation where the stack i is of depth d and a symbol is being pushed into this stack (i.e., the depth of the stack now becomes $d+1$), then this newly pushed stack symbol should be removed within k contexts involving the stack i. This implies that, in a budget-bounded computation, each computation of the stack i is a concatenation of contexts of depth at most d and (k,d)-blocks. The formal definition is as follows:

Let π be a computation of $\mathcal{M}$. We say that π is (k,d)-*budget-bounded* if it can be written as a concatenation $\pi_1 \bullet \pi_2 \bullet \cdots$ of finite contexts (observe that for all j, π_j and π_{j+1} could be contexts of the same stack) in such a way that if $\sigma_i = \pi^i_{i_1} \bullet_i \pi^i_{i_2} \bullet_i \cdots$ (with $i_1 < i_2 < \cdots$) is the maximal sub-sequence of contexts in π belonging to the stack $i \in [1..n]$, then there is a sequence $\rho_i = \rho^i_1 \bullet_i \rho^i_2 \bullet_i \cdots$ of contexts of depth at most d and (k,d)-blocks such that $\sigma_i = \rho_i$.

4 The budget-bounded reachability problem

In this section, we study the decidability and complexity of the reachability problem for MPDS under budget-bounding. Let $\mathcal{M} = (n, Q, \Gamma, \Delta, q_{\mathsf{init}})$ be a MPDS. Let $k \in \mathbb{N}$ be the context-budget and $d \in \mathbb{N}$ the stack depth. The (k, d)-*budget-bounded reachability problem* is to determine, for a given state $q_{\mathsf{final}} \in Q$, whether there is a (k, d)-budget-bounded computation from the initial configuration $c^{\mathsf{init}}_{\mathcal{M}}$ to the configuration $(q_{\mathsf{final}}, \epsilon, \ldots, \epsilon)$.

In the following, we show that the reachability problem for MPDS restricted to only budget-bounded computations is PSPACE-complete.

Theorem 1 *The (k, d)-budget-bounded reachability problem for MPDS is* PSPACE*-complete.*

The rest of this section is devoted to the proof of Theorem 1. The lower bound follows by a reduction from the non-emptiness test of the intersection of several regular languages (which is known to be PSPACE-hard).

To prove the upper bound, we reduce the reachability problem for MPDS restricted to only budget-bounded computations to the non-emptiness of the synchronization of depth-bounded pushdown automata, which is PSPACE-complete (see Lemma 6). The idea behind the proof is the following: Let ρ be a (k, d)-budget-bounded computation and let $i \in [1..n]$ be a stack of $\mathcal{M}$. Then, we know that the projection of π on the set of transitions performed by stack i is a compatible sequence ρ_i of contexts of the form $\pi^i_1 \bullet_i \pi^i_2 \bullet_i \cdots \bullet_i \pi^i_{m_i}$. Since the communication between stacks is done via control states, we can summarize each context π^i_j (with $j \in [1..m_i]$) by a pair of states of the form $(q^i_j, {q'}^i_j)$ where q^i_j (resp. ${q'}^i_j$) is the state at the beginning (resp. end) of the context π_j. Then, we can summarize the stack computation ρ_i by the summary sequence $(q^i_1, {q'}^i_1)(q^i_2, {q'}^i_2) \cdots (q^i_{m_i}, {q'}^i_{m_i})$. We show that it is possible to compute a pushdown automaton $\mathcal{P}_i$ such that the set of all possible summary sequences that can be generated by stack i along a (k, d)-budget-bounded computation can be characterized by the $(-1, d + k)$-bounded language of $\mathcal{P}_i$. Then, we show that we can put together all summary traces and hence produce only consistent interleaving of these summaries (for all stacks) that arises from (k, d)- budget-bounded computation.

Before we present the details, we introduce some notation and definitions that will be useful. For any context $\pi = c_0 t_1 c_1 t_2 \cdots t_m c_m$, we can associate a tuple $Summary(\pi) = (q, q')$ consisting of the pair of states encountered at the beginning and end of the context π (i.e., $q = State(c_0)$ and $q' = State(c_m)$). Let $\rho = \pi_1 \bullet_i \pi_2 \bullet_i \cdots \bullet_i \pi_\ell$ be a sequence of contexts for some $i \in [1..n]$. We can then extend the definition of context summaries to sequences of contexts as follows: $Summary(\rho) = Summary(\pi_1) Summary(\pi_2) \cdots Summary(\pi_\ell)$. The function $Summary$ is also extended in a straightforward manner to blocks, and to sequences of blocks and contexts.

Let $w = (q_1, q'_1)(q_2, q'_2) \cdots (q_m, q'_m)$ be a word over the summary alphabet $Q \times Q$. The word (or summary) w is said to be *consistent* if $q_1 = q_{\mathsf{init}}$, $q'_m = q_{\mathsf{final}}$ and $q'_j = q_{j+1}$ for all $j \in [1..m-1]$. Observe that the set of all consistent summaries can be recognized by a finite state automaton (i.e., a pushdown automaton of depth 0) whose size is polynomial in $\mathcal{M}$. Let $\mathcal{P}_0$ be such a pushdown automaton.

Let π be a (k, d)-budget-bounded computation that reaches the state q_{final}. We can assume that π is of the form $\pi_1 \bullet \pi_2 \bullet \pi_3 \bullet \cdots \bullet \pi_m$ where each π_j, with $j \in [1..m]$, is a stack context. Then, let $\sigma_i = \pi^i_{i_1} \bullet_i \pi^i_{i_2} \bullet_i \pi^i_{i_3} \bullet_i \cdots \bullet_i \pi^i_{m_i}$ (with $i_1 < i_2 < i_3 < \cdots < m_i$) be the maximal sub-sequence of contexts in π belonging to the stack $i \in [1..n]$. By definition, we know that for any stack $i \in [1..n]$, there is a sequence $\rho_i = \rho^i_1 \bullet_i \rho^i_2 \bullet_i \cdots \bullet_i \rho^i_{\ell_i}$ of contexts of depth at most d_i and (k, d)-blocks such that $\sigma_i = \rho_i$.

For every stack $i \in [1..n]$, we associate the summary $Summary(\rho_i)$ to the sequence ρ_i. Then, it is easy to see that there is a consistent word in $\sqcup\!\sqcup(\{Summary(\rho_1)\}, \ldots, \{Summary(\rho_n)\})$.

On the other hand, we can show that if for every stack $i \in [1..n]$, there is a compatible sequence ρ_i of contexts of depth at most d and blocks of size k and depth d such that there is a consistent word in $\sqcup\!\sqcup(\{Summary(\rho_1)\}, \ldots, \{Summary(\rho_n)\})$, then M has a (k, d)-budget-bounded computation that reaches q_{final}.

Let Θ be the set of tuples of the form (q, q) with $q \in Q$. For every $i \in [1..n]$, let $L(\mathcal{M}, i)$ be the set of words w such that there is a compatible sequence σ_i of finite contexts of depth at most d and (k, d)-blocks for stack i such that $w \in \sqcup\!\sqcup(\{Summary(\sigma_i)\}, \Theta^*)$.

Lemma 7 *There is a (k, d)-budget-bounded computation from the initial configuration $c^{\mathsf{init}}_{\mathcal{M}}$ to the configuration $(q_{\mathsf{final}}, \epsilon, \ldots, \epsilon)$ if and only if there is a consistent word in $\sqcup\!\sqcup(L(\mathcal{M}_1, 1), \ldots, L(\mathcal{M}, n))$.*

Now, we can show that checking the existence of such a consistent word can be reduced, in polynomial time, to the non-emptiness of the synchronization of depth-bounded pushdown automata (which is PSPACE-complete), and hence we obtain the completeness of Theorem 1.

Lemma 8 *The problem of checking whether there is a consistent word in $\sqcup\!\sqcup(L(\mathcal{M}, 1), \ldots, L(\mathcal{M}, n))$ can be reduced, in polynomial time, to the non-emptiness of the synchronization of depth-bounded pushdown automata.*

Proof In the following, we show that it is possible, for every $i \in [1..n]$, a pushdown automaton $\mathcal{P}_i$ whose (k, d)-bounded language is precisely $L(\mathcal{M}, i)$. The pushdown automaton $\mathcal{P}_i$ performs the same operations on its state and stack as the ones specified by Δ_i (i.e., the set of operations of stack i). More precisely, $\mathcal{P}_i$ (1) guesses the occurrence of a context π_i of the stack i while making visible as a transition label its summary $Summary(\pi_i) = (q_i, q'_i)$, and (2) checks if from the current stack content and the state q_i, the state q'_i is reachable (and this will mark the end of the simulation of the context π_i). Moreover, $\mathcal{P}_i$ guesses for each context if it is a context of depth at most d or a context belonging to a block of depth d, then checks that all these assumptions hold when checking the feasibility of such contexts.

Formally, the pushdown automaton $\mathcal{P}_i$ is defined by the tuple $(Q_i, \Sigma, \Gamma', \Delta'_i, \{(\sharp, 0)\}, \{(\sharp, 0)\})$ where:

- *States:* $Q_i = (\{\sharp\} \cup (Q \times Q \times \{c, b\})) \times ([0..d] \cup \{> d\})$. A state of the form (q, q', c, ℓ) (resp. (q, q', b, ℓ)) corresponds to the fact that the current state of $\mathcal{M}$ is q and that $\mathcal{P}_i$ is simulating a context of the stack i of depth at most d (resp. belonging to a block of depth d) with q' is the state of $\mathcal{M}$ at the end of this context and ℓ is the current depth of the stack if $\ell \in [0..d]$ and greater than $d + 1$ otherwise. The state $(\sharp, \ell)$ is an intermediary one used between two simulated contexts of $\mathcal{M}$ with ℓ is the current depth of the stack if $\ell \in [0..d]$ and greater than $d + 1$ otherwise.
- *Input Alphabet:* The pushdown automaton $\mathcal{P}_i$ has $\Sigma = Q \times Q$ as input alphabet.
- *Stack Alphabet:* $\Gamma' = (\Gamma \times [1..d + 1]) \cup \Gamma$ is the stack alphabet of $\mathcal{P}_i$. A stack symbol of the form (γ, i) corresponds to the fact that the stack symbol of $\mathcal{P}_i$ at the position is precisely γ. A stack symbol in Γ is only used when the stack depth is greater than $d + 2$.
- *Transition Relation:* Δ'_i contains the following transition rules:
  - *Simulating a context of depth at most d*: In the following we describe the simulation by $\mathcal{P}_i$ of a context of the stack i of depth at most d:

- *Guessing a context:* For every $\ell \in [0..d]$, Δ'_i contains a transition rule of the form $((\sharp, \ell), \epsilon) \xrightarrow{(q,q')} ((q, q', c, \ell), \epsilon)$ for all $q, q' \in Q$. This transition guesses an occurrence of a context of depth at most d and make visible the summary (q, q') of this context.
- *Simulating a pop transition:* For every pop transition of the form (q_1, γ, i, q_2) of $\mathcal{M}$, Δ'_i contains a transition rule of the form $((q_1, q, c, \ell), (\gamma, \ell)) \xrightarrow{\epsilon} ((q_2, q, c, \ell - 1), \epsilon)$ for all $\ell \in [1..d]$ and $q \in Q$.
- *Simulating a push transition:* For every push transition of the form (q_1, i, γ, q_2) of $\mathcal{M}$, Δ'_i contains a transition rule of the form $((q_1, q, c, \ell), \epsilon) \xrightarrow{\epsilon} ((q_2, q, c, \ell + 1), (\gamma, \ell + 1))$ for all $\ell \in [0..d - 1]$ and $q \in Q$.
- *Simulating a nop transition:* For every nop transition of the form (q_1, i, q_2) of $\mathcal{M}$, Δ'_i contains a transition rule of the form $((q_1, q, c, \ell), \epsilon) \xrightarrow{\epsilon} ((q_2, q, c, \ell), \epsilon)$ for all $\ell \in [0..d]$ and $q \in Q$.
- *Ending a context:* For every $\ell \in [0..d]$, $\mathcal{P}_i$ has a transition of the form $((q, q, c, \ell), \epsilon) \xrightarrow{\epsilon} ((\sharp, \ell), \epsilon)$ for all $q \in Q$. This transition of $\mathcal{P}_i$ checks if the current state of $\mathcal{M}$ matches the guessed one at the end of this context and if it is the case $\mathcal{P}_i$ moves to the intermediary state $\sharp$.

– *Simulating a context of block of depth d*: In the following we describe the simulation by $\mathcal{P}_i$ of a context of the stack i of a block of depth d:
- *Guessing a context:* For every $q, q' \in Q$, Δ'_i contains transition rules of the form $((\sharp, d), \epsilon) \xrightarrow{(q,q')} ((q, q', b, d), \epsilon)$ and $((\sharp, > d), \epsilon) \xrightarrow{(q,q')} ((q, q', b, > d), \epsilon)$. This transition guesses an occurrence of a context of a block of depth d and make visible the summary (q, q') of this context.
- *Simulating a push transition:* For every push transition of the form (q_1, i, γ, q_2) of $\mathcal{M}$, Δ'_i contains transition rules of the form $((q_1, q, b, d), \epsilon) \xrightarrow{\epsilon} ((q_2, q, b, > d), (\gamma, d + 1))$ and $((q_1, q, b, > d), \epsilon) \xrightarrow{\epsilon} ((q_2, q, b, > d), (\gamma, > d))$ for all $q \in Q$.
- *Simulating a pop transition:* For every pop transition of the form (q_1, γ, i, q_2) of $\mathcal{M}$, Δ'_i contains a transition rule of the form $((q_1, q, b, > d), (\gamma, > d)) \xrightarrow{\epsilon} ((q_2, q, b, > d), \epsilon)$ for all $q \in Q$.
- *Simulating a nop transition:* For every nop transition of the form (q_1, i, q_2) of $\mathcal{M}$, Δ'_i contains a transition rule of the form $((q_1, q, b, > d), \epsilon) \xrightarrow{\epsilon} ((q_2, q, b, > d), \epsilon)$ for all $q \in Q$.
- *Ending a context:* For every pop transition of the form (q_1, γ, i, q_2) of $\mathcal{M}$, Δ'_i contains a transition rule of the form $((q_1, q_2, b, > d), (\gamma, d + 1)) \xrightarrow{\epsilon} ((\sharp, d), \epsilon)$. Moreover, Δ'_i contains a transition rule of the form $((q, q, b, > d), \epsilon) \xrightarrow{\epsilon} ((\sharp, > d), \epsilon)$ for all $q \in Q$. This transition of $\mathcal{P}_i$ checks if the current state of $\mathcal{M}$ matches the guessed one at the end of this context and if it is the case $\mathcal{P}_i$ moves to the intermediary state $\sharp$.

From the definition of $\mathcal{P}_i$, we have:

Lemma 9 $L_{(k,d)}(\mathcal{P}_i) = L(\mathcal{M}, i)$.

Now, we can apply Lemma 1 to construct, for each pushdown automaton $\mathcal{P}_i$, a bounded-depth pushdown automaton $\mathcal{P}'_i$ such that $L_{k+d+3}(\mathcal{P}'_i) = L_{(k,d)}(\mathcal{P}_i) = L(\mathcal{M}, i)$. Then, checking whether there is a consistent word in $\sqcup\!\sqcup(L(\mathcal{M}, 1), \ldots, L(\mathcal{M}, n))$ boils down

to checking the non-emptiness of the language $L_0(\mathcal{P}_0) \cap \sqcup\sqcup(L_{k+d+3}(\mathcal{P}'_1), \ldots, L_{k+d+3}(\mathcal{P}'_n))$. □

5 Budget-bounded repeated reachability problem

We present in this section a procedure for solving the repeated reachability problem for MPDS under budget-bounding. Let $k, d \in \mathbb{N}$ be two natural numbers. The (k, d)-*budget-bounded repeated reachability problem* is to determine for any given MPDS $\mathcal{M} = (n, Q, \Gamma, \Delta, q_{\mathsf{init}})$ and a set of states $F \subseteq Q$, whether there is an infinite (k, d)-budget-bounded computation of $\mathcal{M}$ starting from the initial configuration that visits some state in F infinitely often (where a computation $\pi = c_0 t_1 c_1 t_2 \cdots$ visits a state q infinitely often if and only if for every $j \in \mathbb{N}$ there is an index $\ell > j$ such that $State(c_\ell) = q$). In the following, we show that this problem can be reduced to the emptiness problem for Büchi finite-state automata.

Theorem 2 *Let $k, d \in \mathbb{N}$ be two natural numbers, $\mathcal{M} = (n, Q, \Gamma, \Delta, q_{\mathsf{init}})$ an MPDS, and $F \subseteq Q$ a set of states. Then it is possible to construct a Büchi finite-state automaton $\mathcal{A}$ such that M has a (k, d)-budget bounded computation that visits infinitely often a state in F if and only if the language $L^\omega(\mathcal{A})$ is not empty. Moreover, the size of $\mathcal{A}$ is $O((|\mathcal{M}|)^{c(n+1)(d+1)(k+1)})$ for some constant c.*

To prove Theorem 2, we assume w.l.o.g. that F contains a single state q_{final}. The key idea behind the proof is the following: Consider any budget-bounded computation and fix a stack i. We *summarize* a context of this stack as a triple of the form (q, f, q'). This represents an execution where the MPDS executed a context of stack i starting at a configuration whose state is q to reach a configuration whose state q', at which point a context-switch occurred. The f component records if that context involved visits the state q_{final}. A full budget-bounded computation can be summarized w.r.t. to the stack i as a sequence of such summaries of the form $(q_1, f_1, q'_1)(q_2, f_2, q'_2)\cdots$. We then show that using similar ideas than the previous section that we can compute an over-approximation of the set of summaries of each stack arising from budget-bounded computations of the MPDS. It is an over-approximation as it is done without verifying whether the *gaps* from q'_1 to q_2, q'_2 to q_3 and so on, can be filled out by a real computation involving the other stacks in a budget bounded manner. We then show that we may put together these summaries and check for consistency across stacks and hence produce only combined summaries (for all stacks) that arise from real budget-bounded computations.

Before we present the details, we introduce some notations and definitions that will be useful. For any finite context $\pi = c_0 t_1 c_1 t_2 \cdots t_m c_m$, we can associate a tuple $Interface(\pi) = (q, f, q')$ of the pair of states encountered at the beginning and end of the context π (i.e., $q = State(c_0)$ and $q' = State(c_m)$), and a flag f indicating if the final state q_{final} was encountered along the context π (i.e., $f = 1$ if there is an index $r \in [1..m]$ such that $State(c_r) = q_{\mathsf{final}}$, otherwise $f = 0$).

Let $\rho = \pi_1 \bullet_i \pi_2 \bullet_i \cdots$ be a sequence of contexts for some stack $i \in [1..n]$, then we can extend the definition of context interfaces to sequence of contexts as follows: $Interface(\rho) = Interface(\pi_1)Interface(\pi_2)\cdots$. The function $Interface$ is also extended in straightforward manner to blocks, and to sequences of blocks and contexts.

Let $w = (q_1, f_1, q'_1)(q_2, f_2, q'_2)\cdots$ be an infinite word over the interface alphabet $Q \times \{0, 1\} \times Q$. The word (or interface) w is said to be *consistent* if $q_1 = q_{\mathsf{init}}$ and $q'_j = q_{j+1}$ for all $j > 1$. Moreover, the interface w visits the state q_{final} infinitely often if for every natural number $j \in \mathbb{N}$, there is a natural number $i > j$ such that $f_i = 1$.

Let ρ be a (k, d)-budget bounded computation that visits the state q_{final} infinitely often. We can assume that ρ is of the form $\pi_1 \bullet \pi_2 \bullet \pi_3 \bullet \cdots$ where each π_j, with $j \in \mathbb{N}$, is a finite stack context. Then, let $\sigma_i = \pi^i_{i_1} \bullet_i \pi^i_{i_2} \bullet_i \pi^i_{i_3} \bullet_i \cdots$ (with $i_1 < i_2 < i_3 < \cdots$) be the maximal sub-sequence of finite contexts in π belonging to the stack $i \in [1..n]$.

For every stack $i \in [1..n]$, we recall that σ_i is a sequence of the form $\rho^i_1 \bullet_i \rho^i_2 \bullet_i \rho^i_3 \bullet_i \cdots$ where each ρ^i_j is a finite context of depth at most d or an (k, d)-block. Let interface $Interface(\sigma_i)$ be the interface of the sequence σ_i. Then, it is easy to see that, in this case, there is an infinite consistent word in $\sqcup\!\sqcup(\{Interface(\sigma_1)\}, \ldots, \{Interface(\sigma_n)\})$ which visits q_{final} infinitely often.

Let Θ be the set of tuples of the form $(q, 0, q)$ with $q \in Q$. For every $i \in [1..n]$, let $L^\omega(\mathcal{M}, i)$ be the set of infinite words w such that there is a compatible finite or infinite sequence of contexts of depth at most d and blocks of size k and depth d for stack i such that $w \in \sqcup\!\sqcup(\{Interface(\sigma_i)\}, \Theta^\omega)$. Then, it is easy to see that:

Lemma 10 *There is a (k, d)-budget-bounded computation from the initial configuration $c^{\mathsf{init}}_{\mathcal{M}}$ that visits infinitely often the state q_{final} iff there is an infinite consistent word in $\sqcup\!\sqcup(L^\omega(\mathcal{M}_1, 1), \ldots, L^\omega(\mathcal{M}, n))$ that visits q_{final} infinitely often.*

Now, we can show that checking the existence of such an infinite consistent word can be reduced to the emptiness problem for Büchi finite-state automata whose size is $O((|\mathcal{M}|)^{c(n+1)(d+1)(k+1)})$ for some constant c.

Lemma 11 *The problem of checking whether there is an infinite consistent word in $\sqcup\!\sqcup(L^\omega(\mathcal{M}, 1), \ldots, L^\omega(\mathcal{M}, n))$ which visits q_{final} infinitely often can be reduced to the emptiness problem for a Büchi finite-state automaton whose size is $O((|\mathcal{M}|)^{c(d+1)(k+1)n})$ for some constant c.*

Proof Similar to the proof of Lemma 8, we can show that it is possible construct for each stack $i \in [1..n]$, a Büchi pushdown automaton $\mathcal{P}_i$ whose (k, d)-bounded language is precisely $L^\omega(\mathcal{M}, i)$. The Büchi pushdown automaton $\mathcal{P}_i$ performs the same operations on its state and stack as the ones specified by Δ_i. In fact, $\mathcal{P}_i$ (1) guesses the occurrence of a finite context π_i of the stack i while making visible as a transition label its interface $Interface(\pi_i) = (q_i, f_i, q'_i)$, and (2) checks if from the current stack content and the state q_i, the state q'_i is reachable (and this will mark the end of the simulation of the context π_i) and that the state q_{final} is visited along this context if $f_i = 1$. Moreover, $\mathcal{P}_i$ guesses for each context if it is a context of depth at most d or a context belonging to a block of depth d, then checks that all these assumptions hold when checking the feasibility of such contexts.

Formally, the pushdown automaton $\mathcal{P}_i$ is defined by the tuple $(Q_i, \Sigma, \Gamma', \Delta'_i, \{(\sharp, 0)\}, \{(\sharp\} \times [0..d)]\})$ where:

- *States:* $Q_i = (\{\sharp\} \cup (Q \times Q \times \{c, b\} \times \{0, 1\})) \times ([0..d] \cup \{> d\})$. A state of the form (q, q', c, ℓ, f) (resp. (q, q', b, ℓ, f)) corresponds to the fact that the current state of $\mathcal{M}$ is q and that $\mathcal{P}_i$ is simulating a context of the stack i of depth at most d (resp. belonging to a block of depth d) with q' is the state of $\mathcal{M}$ at the end of this context and ℓ is the current depth of the stack if $\ell \in [0..d]$ and greater than $d + 1$ otherwise. Moreover, if $f = 1$ then $\mathcal{P}_i$ should visits the state q_{final} along this context. The state $(\sharp, \ell)$ is an intermediary one used between two simulated contexts of $\mathcal{M}$ with ℓ is the current depth of the stack if $\ell \in [0..d]$ and greater than $d + 1$ otherwise.
- *Input Alphabet:* The pushdown automaton $\mathcal{P}_i$ has $\Sigma = Q \times \{0, 1\} \times Q$ as input alphabet.
- *Stack Alphabet:* $\Gamma' = (\Gamma \times [1..d + 1]) \cup \Gamma$ is the stack alphabet of $\mathcal{P}_i$. A stack symbol of the form (γ, i) corresponds to the fact that the stack symbol of $\mathcal{P}_i$ at the position is precisely γ. A stack symbol in Γ is used when the stack depth is greater than $d + 2$.

– *Transition Relation:* $\mathcal{P}_i$ has the following transition rules:

  – *Simulating a context of depth at most d*: In the following we describe the simulation by $\mathcal{P}_i$ of a context of the stack i of depth at most d:
    - *Guessing a context:* For every $\ell \in [0..d]$, Δ_i' contains a transition rule of the form $((\sharp, \ell), \epsilon) \xrightarrow{(q,f,q')} ((q, q', c, \ell, f), \epsilon)$ for all $q, q' \in Q$ and $f \in \{0, 1\}$. This transition guesses an occurrence of a context of depth at most d and make visible the interface (q, f, q').
    - *Simulating a pop transition:* For every pop transition of the form (q_1, γ, i, q_2) of $\mathcal{M}$, $\ell \in [1..d]$, $f \in \{0, 1\}$ and $q \in Q$, Δ_i' contains a transition rule of the form $((q_1, q, c, \ell, f), (\gamma, \ell)) \xrightarrow{\epsilon} ((q_2, q, c, \ell - 1, f'), \epsilon)$ such that $f' = 0$ if $q_2 = q_{\mathsf{final}}$ and $f' = f$ otherwise.
    - *Simulating a push transition:* For every push transition of the form (q_1, i, γ, q_2) of $\mathcal{M}$, $\ell \in [0..d-1]$, $f \in \{0, 1\}$ and $q \in Q$, Δ_i' contains a transition rule of the form $((q_1, q, c, \ell, f), \epsilon) \xrightarrow{\epsilon} ((q_2, q, c, \ell+1, f'), (\gamma, \ell+1))$ such that $f' = 0$ if $q_2 = q_{\mathsf{final}}$ and $f' = f$ otherwise.
    - *Simulating a nop transition:* For every nop transition of the form (q_1, i, q_2) of $\mathcal{M}$, $\ell \in [0..d]$, $f \in \{0, 1\}$ and $q \in Q$, $\mathcal{P}_i$ has a transition of the form $((q_1, q, c, \ell, f), \epsilon) \xrightarrow{\epsilon} ((q_2, q, c, \ell, f'), \epsilon)$ such that $f' = 0$ if $q_2 = q_{\mathsf{final}}$ and $f' = f$ otherwise.
    - *Ending a context:* For every $\ell \in [0..d]$, Δ_i' contains a transition rule of the form $((q, q, c, \ell, 0), \epsilon) \xrightarrow{\epsilon} ((\sharp, \ell), \epsilon)$ for all $q \in Q$. This transition of $\mathcal{P}_i$ checks if the current state of $\mathcal{M}$ matches the guessed one at the end of this context and that the condition on visiting the state q_{final} along this context is satisfied and if it is the case $\mathcal{P}_i$ moves to the intermediary state $\sharp$.
  – *Simulating a context of block of depth d*: In the following we describe the simulation by $\mathcal{P}_i$ of a context of the stack i of a block of depth d:
    - *Guessing a context:* For every $q, q' \in Q$ and $f \in \{0, 1\}$, Δ_i' contains the following transition rules $((\sharp, d), \epsilon) \xrightarrow{(q,f,q')} ((q, q', b, d, f), \epsilon)$ and $((\sharp, > d), \epsilon) \xrightarrow{(q,f,q')} ((q, q', b, > d, f), \epsilon)$. This transition guesses an occurrence of a context of a block of depth d and make visible the interface (q, f, q') of this context.
    - *Simulating a push transition:* For every push transition of the form (q_1, i, γ, q_2) of $\mathcal{M}$, $f \in \{0, 1\}$ and $q \in Q$, $\mathcal{P}_i$ has the following transitions $((q_1, q, b, d, f), \epsilon) \xrightarrow{\epsilon} ((q_2, q, b, > d, f'), (\gamma, d + 1))$ and $((q_1, q, b, > d, f), \epsilon) \xrightarrow{\epsilon} ((q_2, q, b, > d, f'), (\gamma, > d))$ such that $f' = 0$ if $q_2 = q_{\mathsf{final}}$ and $f' = f$ otherwise.
    - *Simulating a pop transition:* For every pop transition of the form (q_1, γ, i, q_2) of $\mathcal{M}$, $f \in \{0, 1\}$ and $q \in Q$, $\mathcal{P}_i$ has the following transition $((q_1, q, b, > d, f), (\gamma, > d)) \xrightarrow{\epsilon} ((q_2, q, b, > d, f'), \epsilon)$ such that $f' = 0$ if $q_2 = q_{\mathsf{final}}$ and $f' = f$ otherwise.
    - *Simulating a nop transition:* For every nop transition of the form (q_1, i, q_2) of $\mathcal{M}$, $f \in \{0, 1\}$ and $q \in Q$, $\mathcal{P}_i$ has a transition of the form $((q_1, q, b, > d, f), \epsilon) \xrightarrow{\epsilon} ((q_2, q, b, > d, f'), \epsilon)$ such that $f' = 0$ if $q_2 = q_{\mathsf{final}}$ and $f' = f$ otherwise.
    - *Ending a context:* For every pop transition of the form (q_1, γ, i, q_2) of $\mathcal{M}$ and $f \in \{0, 1\}$, $\mathcal{P}_i$ has a transition of the form $((q_1, q_2, b, > d, f), (\gamma, d + 1)) \xrightarrow{\epsilon} ((\sharp, d), \epsilon)$ if one of the following conditions holds: $f = 0$ or $q_2 = q_{\mathsf{final}}$.

Moreover, $\mathcal{P}_i$ has a transition of the form $((q, q, b, > d, 0), \epsilon) \xrightarrow{\epsilon} ((\sharp, > d), \epsilon)$ for all $q \in Q$.

From the definition of $\mathcal{P}_i$, it is easy to see that the following lemma holds:

Lemma 12 $L^{\omega}_{(k,d)}(\mathcal{P}_i) = L^{\omega}(\mathcal{M}, i)$.

Now, we can apply Lemma 1 to construct, for each Büchi pushdown automaton $\mathcal{P}_i$, a bounded-depth Büchi pushdown automaton $\mathcal{P}'_i$ such that $L^{\omega}_{k+d+3}(\mathcal{P}'_i) = L^{\omega}_{(k,d)}(\mathcal{P}_i) = L^{\omega}(\mathcal{M}, i)$. Then, we construct a Büchi finite-state automaton $\mathcal{A}_i$ such that $L^{\omega}(\mathcal{A}_i) = L^{\omega}_{k+d+3}(\mathcal{P}'_i)$ and whose size is $O((|\mathcal{M}|)^{c'(d+1)(k+1)})$ for some constant c' (since the size of $\mathcal{P}'_i$ is polynomial in the size of $\mathcal{P}_i$, which, in turn, is polynomial in the size of $\mathcal{M}$). The idea is to encode the possible stack content of $\mathcal{P}'_i$ in the control state of $\mathcal{A}_i$. Notice that there is a finite number of possible stack content of $\mathcal{P}'_i$ since its stack depth is always bounded by $(k + d + 3)$. Finally, we can use standard automata construction, to show that we can construct a Büchi automaton $\mathcal{B}_0$ that accepts all the consistent infinite words over the alphabet $Q \times \{0, 1\} \times Q$, by ensuring that adjacent letters in the shuffle being generated are compatible, and also verify that the resulting sequence visits q_{final} infinitely often. Then, checking whether there is a consistent word in $\sqcup\sqcup(L^{\omega}(\mathcal{M}, 1), \ldots, L^{\omega}(\mathcal{M}, n))$ boils down to checking the non-emptiness of the language $L^{\omega}(\mathcal{A}_0) \cap \sqcup\sqcup(L^{\omega}(\mathcal{A}_1)), \ldots, L^{\omega}(\mathcal{A}_n))$ which can be recognized by a Büchi automaton $\mathcal{B}$ whose size is $O((|\mathcal{M}|)^{c(n+1)(d+1)(k+1)})$ for some constant c (i.e., the product of $\mathcal{B}_0, \mathcal{B}_1, \ldots, \mathcal{B}_n$). □

6 Budget-bounded model checking for MPDS

We consider in this section the linear-time model checking problem for MPDS's under budget-bounding. We consider that we are given ω-regular properties expressed in linear-time propositional temporal logic (LTL) [35] or in the linear-time propositional μ-calculus [41]. Let us fix a set of atomic propositions $Prop$, and let $k, d \in \mathbb{N}$ be two natural numbers. The (k, d)-*budget-bounded model-checking* problem is the following: Given a formula φ (in LTL or in the linear-time μ-calculus) with atomic propositions from $Prop$, and a MPDS $\mathcal{M} = (n, Q, \Gamma, \Delta, q_{\mathsf{init}})$ along with a labeling function $\Lambda : Q \to 2^{Prop}$ associating to each state $q \in Q$ the set of atomic propositions that are true in it, check whether all infinite (k, d)-budget-bounded computations of $\mathcal{M}$ from the initial configuration c_{init} satisfy φ.

To solve this problem, we adopt an automata-based approach similar to the one used in [7,8] to solve the analogous problem for pushdown systems. We construct a Büchi automaton $\mathcal{B}_{\neg\varphi}$ over the alphabet 2^{Prop} accepting the negation of φ [42,43]. Then, we compute the product of the MPDS $\mathcal{M}$ and the Büchi automaton $\mathcal{B}_{\neg\varphi}$ to obtain a MPDS $\mathcal{M}_{\neg\varphi}$ with a Büchi accepting set of states F and leaving us with the task if any of its its (k, d)-budget bounded runs is accepting. We can then reduce our model-checking problem to the (k, d)-budget bounded repeated reachability problem for MPDSs, which, by Theorem 2, can be solved.

Theorem 3 *The problem of budget-bounded model checking ω-regular properties of multi-pushdown systems is EXPTIME-complete.*

The lower bound of Theorem 3 follows immediately from the fact that the model-checking problem for LTL and linear time μ-calculus for pushdown systems (i.e., MPDS with only one stack) are EXPTIME-complete [7].

For the upper bound, it is well known that, given a MPDS $\mathcal{M}$ and an ω-regular formula φ, it is possible to construct a MPDS $\mathcal{M}'$ and a set of repeating states F of $\mathcal{M}'$ such that the

problem of budget-bounded model checking of $\mathcal{M}$ w.r.t. the formula φ is reducible to the (k, d)-budget-bounded repeated state reachability problem of a MPDS $\mathcal{M}'$ w.r.t. F. Moreover, the size of $\mathcal{M}'$ and F is exponential in the size of φ and polynomial in the size of $\mathcal{M}$, k and d. Applying Theorem 2 to the MPDS $\mathcal{M}'$ and F, we obtain our complexity result.

7 Transformation

In this section, we will describe an automatic code-to-code translation from concurrent to sequential programs. The resulting sequential program simulates the concurrent program running under the uniformly bounded-budget restriction. First, we will briefly explain the language for concurrent programs. The remainder of the section describes the translation.

We consider a C-like programming language where concurrent programs consist of processes, procedures and statements. We assume that variables range over some (potentially infinite) data domain $\mathbb{D}$ and that we have a language of expressions $\langle expr \rangle$ interpreted over $\mathbb{D}$. The statements consists of simple C-like statements, enriched with `nop`, `assume`, `assert` and `atomic`. A *procedure* consists of a sequence of *arguments*, a set of *local variables*, and a sequence of *statements*. A *process* is a tuple $\mathcal{P} = \langle G, \mathcal{F}_1 \dots \mathcal{F}_m \rangle$, where G is a finite set of *global variables* and each $\mathcal{F}_i$ is a procedure. For each process, there should be exactly one distinguished procedure called `main`, which constitutes the entry point of that process. A *concurrent program* is a tuple $\mathcal{C} = \langle S, \mathcal{P}_1 \dots \mathcal{P}_n \rangle$, consisting of a finite set S of *shared variables* and a sequence of processes.

Next, we describe an automatic transformation from a concurrent program $\mathcal{C} = \langle S, \mathcal{P}_1 \dots \mathcal{P}_n \rangle$ to a sequential program $\mathcal{S}$ which simulates the behavior of $\mathcal{C}$ up to a given bound k_i of context switches for each $\mathcal{P}_i$ whenever the stack of $\mathcal{P}_i$ grows above d_i. If the stack of $\mathcal{P}_i$ *never* grows above d_i, there is no limit on the number of times $\mathcal{P}_i$ can be switched out.

7.1 Programs without procedure calls

Assume that we have a concurrent program $\mathcal{C} = \langle S, \mathcal{P}_1 \cdots \mathcal{P}_n \rangle$, where no process $\mathcal{P}_i$ contains a procedure call, i.e. each process consists only of a main procedure. To construct the sequential program $\mathcal{S}$, we take each statement in the procedure and put it inside a scheduling loop. We introduce for each process $\mathcal{P}_i$ a variable `pc_i` which keeps track of its programs counter. In the scheduling loop, each statement is enclosed in a conditional which contains a nondeterministic check of a Boolean variable `?` and a check for the correct program counter value. If the program counter check succeeds, but `?` happens to be false, the statement will not be executed. Additionally, all other program counter checks will fail, so the control flow will fall through the remainder of the statements. In this way, a context switch is simulated.

As an example, consider the program in Fig. 1 along with the sequential program which simulates it. It is easy to see that the sequential program simulates all behaviors of the concurrent program, including the interleaving `x = x + 2, x = 1, assert(x != 1), x = 2` in which the assertion fails.

7.2 Programs with procedure calls

Assume now that we add procedure calls. There are two cases whenever a call happens in $\mathcal{P}_i$. Either the stack height is above d_i, in which case we must limit the number of preemptions of $\mathcal{P}_i$ to k_i as long as $\mathcal{P}_i$ stays above d_i, or the stack height is not above d_i, in which case the

```
process example:

int x = 0;

process p1:
void main(){
  x = 1;
  x = 2;
}

process p2:
void main(){
  x = x + 2;
  assert(x != 1);
}
```

Fig. 1 A simple example program

```
process transformed:

int pc1 = 1;
int pc2 = 1;
int running;
int x = 0;

void scheduler(){
  while(progress){
    progress = false;

    // schedule a process
    if( ? && pc1!=3){
      running = 1;
    }
    if( ? && pc2!=3) {
      running = 2;
    }
    // process 1
    if(running == 1){
      if(pc1==1 && ?){
        x = 1;
        progress = true;
        pc1 = 2;
      }
      if(pc1==2 && ?){
        x = 2;
        progress = true;
        pc1 = 3;
      }
    }
    // process 2
    if(running == 2){
      if(pc2==1 && ?){
        x = x + 2;
        progress = true;
        pc2 = 2;
      }
      if(pc2==2 && ?){
        assert(x != 1);
        progress = true;
        pc2 = 3;
      }
    }
  }
}
```

Fig. 2 Transformation of the program in Fig. 1

number of preemptions is unbounded. Instead of keeping track of these two possibilities, we will *inline* the procedure calls in the main procedure of each process $\mathcal{P}_i$ d_i times (Fig. 2).

7.2.1 Inlining

For any process $\mathcal{P}$, let $I(\mathcal{P})$ be the result of inlining all procedure calls in the main procedure of $\mathcal{P}$. Note that this inlining might create new local variables. Let I^m denote the result of composing I with itself m times. Given a concurrent program $\mathcal{C} = \langle S, \mathcal{P}_1 \ldots \mathcal{P}_n \rangle$, we construct an *inlined concurrent program* $\mathcal{C}' = \langle S, I^{d_1}(\mathcal{P}_1) \ldots I^{d_n}(\mathcal{P}_n) \rangle$. In the execution of $\mathcal{C}'$, any procedure call in $I^{d_i}(\mathcal{P}_i)$ means that the corresponding execution in $\mathcal{C}$ would take the process $\mathcal{P}_i$ above its stack limit d_i. This means that we can differentiate between code based

on whether it is inside or outside the main procedure of the process. Code that is outside the main procedure will be transformed in a way that takes into account the preemption bound k_i.

7.2.2 Context switching

In [24], La Torre, Madhusudan and Parlato describe a transformation that only keeps track of the local state of one process, at the expense of recomputing that state after context switches. More precisely, the transformation keeps track of $k+1$ valuations $s_0 \dots s_k$ of shared variables. The initial values of the shared variables are stored in s_0. Assume that process $\mathcal{P}_1$ starts running. When the context switch occurs, the values of the shared variables are stored in s_1. Another process then runs until there is another context switch, storing the shared varaibles in s_2. When $\mathcal{P}_1$ is switched in, it is executed *from the beginning* until the values of the shared variables equal s_1, i.e. the values when it was switched out. The shared variables are then assigned the values stored in s_2, and the execution continues. When the next context switch occurs, the shared variables are stored in s_3, and so on.

We use a similar approach to deal with context switches when a process $\mathcal{P}_i$ is above its stack bound d_i. The state of each process is thus stored explicitly up to the point where a process goes above its stack bound. When several processes are above their bounds, we only keep the local state of the one currently running. An important difference between our model and the one of [24] is that even when all processes are above their stack bound, we allow k preemptions *per process*. To facilitate this, we store $2k+1$ copies of the shared variables for each process.

7.2.3 Phases

An execution r of a single process in a concurrent program can be divided into a sequence $r_0, r_1, \dots$ of executions separated by preemptions. We call each r_i a *phase*. In other words, a phase is a continuous sequence of statements. A process begins in r_0 and executes statements until there is a context switch. When the process gets switched back in, it runs r_1, and so on.

In the special case where a process is always above its stack bound, the execution of that process may consist of at most $k+1$ phases. For this reason, we introduce for each process a variable `phase`, which keeps track of which phase the execution is in. This variable is increased whenever a context switch happens. Since we reconstruct the local state of a process by executing from the beginning, we also store a virtual phase `phase'`, which is updated both during the reconstruction and the actual execution. This means that as long as `phase' < phase`, we are reconstructing the local state.

In general, a process is not always above its stack bound. When a process goes below that bound, the budget of allowed preemptions is reset. In our transformation, this means that we reset the phase variables, starting again from `phase = 0` the next time a procedure call happens.

7.2.4 Transformation

For a concurrent program $\mathcal{C} = \langle S, \mathcal{P}_1 \dots \mathcal{P}_n \rangle$, we first construct the corresponding inlined concurrent program $\mathcal{C}' = \langle S, I^{d_1}(\mathcal{P}_1) \dots I^{d_n}(\mathcal{P}_n) \rangle$. We then transform $\mathcal{C}'$ into a sequential program $\mathcal{S}$ that simulates $\mathcal{C}'$. We can find among the global variables of $\mathcal{S}$, for each process $\mathcal{P}_t$, the sets $S_t^0, \dots, S_t^{2k+1}$ of copies of the shared variables of $\mathcal{C}$. The transformation of the statements in the main procedures of each process is done in the same way as previously,

```
1
2  if(phase_t == 0){
3    S_t^0 = S;
4    S = S_t^0;
5  }
6  ⋮
7  if(phase_t == k){
8    S_t^{2k} = S;
9    S = S_t^0;
10 }
```

```
1
2  if(!ret && ?){
3    if(phase'_t == phase_t){
4      if(phase_t == 0){
5        S_t^1 = S;
6      }
7      ⋮
8      if(phase_t == k){
9        S_t^{2k+1} = S;
10     }
11
12     phase_t =
13     phase_t + 1;
14     ret = true;
15 }
```

Fig. 3 *Left* Transformation of procedure calls in process t. *Right* Context switches in procedures of process t.

```
1  if(!ret && ?){
2    if(phase'_t < phase_t){
3      if(phase'_t == 0 && S == S_t^1){
4        phase'_t = 1;
5        S = S_t^2;
6      }
7      ⋮
8      if(phase_t == k-1 && S == S_t^{2k-1}){
9        phase'_t = k;
10       S = S_t^{2k};
11     }
12 }
```

Fig. 4 Checking reconstruction of local state in process t

with the exception of *procedure calls*. Before each procedure call in a process $\mathcal{P}_t$, we insert a code block that, if the process is not recomputing the local state, saves the current values of the global variables in S_t^0. This code block is shown in Fig. 3.

The set of procedures of the sequential program is the union of the transformed procedures of its processes. When we transform a procedure, we perform three steps:

- To simulate context switches, we add the code shown in the right side in Fig. 3 before any statement that contains shared variables and therefore is visible to the outside.
- Before any statement that contains shared variables, we also add code to detect whether the local context has been reconstructed or not. This code is shown in Fig. 4.
- At the end of the procedure, we check if we are about to return to the scheduling loop without having reconstructed the local state. In this case, we abort.

8 Experimental results

We have evaluated our approach on several examples, including one in which a large number of context switches is needed in order to reach a bad state. The experiments presented in Table 1 were run on a 2.2 GHz Intel Core i7 with 4 GB of memory. Most literature examples

Table 1 We report the running times of our experimentation results in seconds. We use the symbol – to denote a *timeout* (set to 900 s). The column k_1 contains the context-switch budget for our code-to-code translation. The columns k_2 and k_3 are the number of context switches given as input for POIROT and ESBMC, respectively. Note that these numbers are *per thread NF* Bug Not Found, *FP* False Positive

Examples	Type of analysis							
	Concurrent to sequential				Concurrent			
	k_1	MOPED	CBMC	ESBMC	k_2	POIROT	k_3	ESBMC
Account [15]	0	37.35	–	**1.1**	4	3.34	10	–
BigNum [22]	0	**8.39**	–	–	26	–	26	–
Bluetooth3a [22]	0	744.49	–	–	11	**18.38**	28	*FP*
Token Ring [15]	0	**0.13**	0.2	0.18	1	2.72	4	1.47
Account Bad [15]	0	2.31	0.48	1.41	1	2.13	4	**0.11**
BigNum Bad [22]	0	**5.9**	13.4	239.4	26	–	26	–
Bluetooth1 [40]	1	4.18	**0.37**	1.28	2	1.92	5	*NF*
Bluetooth2 [40]	1	0.64	5.68	34.38	3	2.9	5	**0.5**
Bluetooth3b [40]	1	1.26	**0.95**	5.0	2	2.5	5	*NF*
Infinite Loop 1 [36]	1	4.1	0.2	0.25	2	1.45	1	**0.08**
Infinite Loop 2 [36]	1	17.3	0.84	3.85	1	0.96	1	**0.09**
Token Ring Bad [15]	0	**0.13**	0.16	0.26	2	2.74	4	0.27

The time in bold corresponds to the best running time for each example

are written in pseudo code or C-like code. In order to run them, we manually translated them to our syntax. This operation can be automated.

We have implemented our code-to-code translation scheme in HASKELL [22]. The scheme inputs a concurrent program P and produces a sequential program P' such that running P under the budget-bounded restriction yields the same set of reachable states as running P'. The sequential program is delivered in different languages, namely: REMOPLA for MOPED [16], and a C-like language for CBMC [11] and ESBMC [12].

We use these three tools as backend to verify the obtained sequential code. Our experimental results are then compared to ones obtained using two verification tools for concurrent programs, namely ESBMC and POIROT [28]. The time required for sequentialization is negligible and not included in the results.

The Table 1 summarizes our experimental results. In the upper part of the table, only safe (correct) programs are considered. We fixed the context switch bounds k_1, k_2, and k_3 such that all compared tools are able to cover the same set of control locations. The results show that our approach manages to perform better in three out of four examples, in particular for the BigNum example where the concurrent tools timeout. In this example, a large number of context switches is required in order to explore all the interleavings that might lead to the assertion violation. Also, we noticed that ESBMC finds a bug in the correct example Bluetooth3a. It has been confirmed that this is a false positive [32].

In the lower part of the table, we consider faulty programs. For half of those programs, the experimental results show that our approach succeeds in finding all the bugs within a smaller amount of time compared to the concurrent tools. In particular, both POIROT and ESBMC timed out on the BigNum Bad example (Fig. 5), where a large number of context switches is required in order to violate the assertion. Also, ESBMC, which did as well as

```
1  int z=0;
2  bool stop1;
3  bool stop2;
4
5  process thread1:
6
7  void main()
8  {
9    stop1 = false;
10
11   while(!stop1)
12   {
13     z = z + 1;
14
15     stop1 = true;
16
17     stop2 = false;
18   }
19
20   assert(z < 50);
21 }
```

```
22
23
24
25
26 process thread2:
27
28 void main()
29 {
30   stop2 = false;
31
32   while(!stop2)
33   {
34     z = z + 1;
35
36     stop2 = true;
37
38     stop1 = false;
39   }
40
41 }
```

Fig. 5 The BigNum Bad example: in total, 50 or 51 context switches are needed in order to violate the assertion, depending on which thread starts

our approach in terms of time, failed to find bugs in the faulty examples Bluetooth1 and Bluetooth3b regardless of the allowed number of context-switches it was given.

9 Termination under fairness

We saw earlier that to check safety properties of a concurrent program running under the budget-bounded restriction, we can check that the *sequentialization* of the concurrent program satisfies those properties. A feature of our approach, as opposed to context-bounded sequentializations, is that it can also be applied to check liveness properties. In any context-bounded execution, only one thread can execute infinitely often. This makes context-bounded sequentializations inapplicable to the class of programs which contain *cordinated* nontermination, i.e. where the interaction of several processes leads to a nonterminating execution. In our approach, we allow executions with an infinite number of context switches as long as the stack bounds are respected.

However, checking liveness properties introduces additional difficulties. First, the scheduler itself may induce additional behaviour w.r.t. these properties. For example, if one does not enforce some kind of progress, the sequentialization may produce infinite traces even if the two processes contain no loops. Second, the sequentialization may produce some traces that would not be present in the real system because of fairness constraints in the *system* scheduler.

In this section, we consider termination of concurrent programs running under the budget-bounded restriction and some *fairness constraint*. In other words, we want to find out whether the concurrent program has any infinite trace which is fair. We consider two different levels of fairness, reflecting common fairness assumptions in process schedulers, namely *impartiality* and *strong fairness*:

Impartiality: If a process has not ended, it will eventually be scheduled.

```
1  bool g = false;
2
3  process thread1:
4
5  void main()
6  {
7      while(!g)
8      {
9          skip;
10     }
11
12     g = false;
13 }
```

```
14
15
16 process thread2:
17
18 void main()
19 {
20     g = true;
21
22     while(g)
23     {
24         skip;
25     }
26 }
```

```
1  bool g = false;
2
3  process thread1:
4
5  void main()
6  {
7      while(g)
8      {
9          g = false;
10     }
11 }
```

```
12
13
14 process thread2:
15
16 void main()
17 {
18     while(!g)
19     {
20         g = true;
21     }
22 }
```

Fig. 6 *Top* The program Ex1 which does not have any fair nonterminating runs. *Bottom* The program Ex2, with fair nonterminating runs

Strong fairness: If a process is infinitely often enabled, it will make progress infinitely often.

For example, the program Ex1, shown in the top half of Fig. 6, contains an infinite run, but not under any fairness constraint. If we consider this program under e.g. impartiality, it will always terminate, since the second process must eventually be scheduled. Note that this requires the second process to actually make *progress*, in the sense that it is not switched out before it executes any statement. This is a reasonable assumption to make, so we will take "eventually be scheduled" to mean "eventually be scheduled and executed".

We encode the above constraints as LTL formulas that we use as assumptions when we check for termination. To check for termination, we use a variable `end` which is initially false, and which we set to true after the scheduling loop. The system terminates under a given fairness constraint φ if the formula $\varphi \longrightarrow \Diamond end$ is satisfied. We encode the fairness constraints in the following way:

Impartiality:

$$\bigwedge_{i \in \{1,\ldots,n\}} \Box(pc_i \neq lastpc_i \longrightarrow \Diamond(running = i \wedge progress)$$

Strong Fairness:

$$\bigwedge_{i \in \{1,\ldots,n\}} \Box\Diamond(enabled_i) \longrightarrow \Box\Diamond(running = i \wedge progress)$$

```
1  inline aqcuire(x, i)
2  {
3    e[i] = false;
4
5    do
6    :: true ->
7       atomic
8       {
9         progress = true;
10        if
11        :: x -> skip;
12        :: else -> x =
                true; break;
13        fi;
14      }
15   od;
16
17   e[i] = true;
18 }
```

Fig. 7 Spin macro implementing `acquire(x,i)`

These formulas make use of several variables in the scheduler. The variables $\texttt{pc}_i$, $\texttt{lastpc}_i$ and `running` are used in the transformation described in the previous section. The formula for impartiality expresses, for each i, that if process i has not terminated, it will eventually be scheduled and make progress. The condition that $\texttt{pc}_i$ is not equal to $\texttt{lastpc}_i$ is needed to ensure that we do not discard runs where some process terminates but one (or more) processes keep running infinitely.

The formula for strong fairness expresses that if a process is continuously enabled at some point, it will be scheduled and make progress infinitely often. The variable `progress` is set to false before scheduling a process, and is set to true if a statement of that process is simulated. In the previous section, this is used as an optimization to prevent that a process is switched out before making progress.

For strong fairness, we must specify what it means for a process to be enabled, and take this into account when performing the sequentialization. For example, a process is not enabled when waiting for a lock or when it has finished executing. We have extended our language with the primitives `acquire(x)` and `release(x)`. In the translation to a Spin model, we use macros to implement `acquire(x)` and `release(x)`. Figure 7 shows the implementation of `acquire(x)`. The implementation takes the process number as an extra argument. Just before we wait for the lock, we mark that the process is disabled. When the process acquires the lock, we enable it again. In order for the impartiality constraint to work in the presence of locks, we set progress to true when checking the lock. In other words, switching to a process in order to try the lock still counts as progress. Note that this does not interfere with the strong fairness constraint.

We applied our budget-bounded sequentialization on three different programs, taken from [3], and checked termination under fairness with Spin, using the above mentioned fairness assumptions. We compare this to the results of checking termination wihtout any fairness constraint. As mentioned earlier, the program Ex1, shown in the upper half of Fig. 6, only allows nonterminating runs if the scheduler never allows one of the processes to make progress. This program terminates under any fairness condition.

Figure 8 shows the Promela model implementing the sequentialization of the program Ex1. The program contains the usual instrumentation variables and scheduling constructs mentioned in Sect. 7. Additionally, the Promela model contains LTL formulas expressing termination, termination under impartiality and termination under strong fairness.

```
#define true 1
#define false 0

int running;
bool progress;

int pc1;
int pc2;

bool g;
bool star;

bool ena[3];
bool end;

ltl strong_fairness_termination
{
(( []<>(ena[1]) -> []<>(running==1
     && progress))
&&
( []<>(ena[2]) -> []<>(running==2 &&
      progress)))
->
<>end
}

ltl impartiality_termination
{
(
[](!(pc1 == 4) -> <>(running==1 &&
     progress)) &&
[](!(pc2 == 4) -> <>(running==2 &&
     progress))
)
->
<>end
}

ltl termination
{
<>end
}

init {

  atomic {

    /* Initialize variables */
    running = 0;
    progress = true;
    pc1 = 1;
    pc2 = 1;
    g = false;
    ena[1] = true;
    ena[2] = true;
    end = false;

    /* Launch scheduler */
    run procA();
  }

}

proctype procA() {

/* Main loop, stops when no progress
      is made  */
  do
  :: progress ->

    progress = false;


/* Context switch block: */

/* Thread 1 statements: */

    .
    .
    .

  if(pc1 == lastpc1){
    ena[1] = false;
  }
/* Thread 2 statements: */

    .
    .
    .

  if(pc2 == lastpc2){
    ena[2] = false;
  }

  :: else -> break;
  od;

  /* Terminate */
  end = true;
}
```

Fig. 8 The sequentialization of the program Ex1.

In the program Ex2, shown in the lower half of Fig. 6, there is a fair nonterminating execution which manifests itself when the scheduler always switches out a process after the assignment to `g`. Such an execution satisfies impartiality and (since the processes are always enabled) both fairness constraints.

The program Ex3, shown in Fig. 9, implements a retrying mechanism often used in concurrent data structures (see e.g. [20]). A validation phase before the effect of an operation takes place ensures that there has been no interference from other threads. If there has been interference, the operation is simply tried again. This might lead to nonterminating executions in which another thread always interferes with the operation.

The results of checking the above examples using Spin are summarized in Table 2. For each example and property, we report whether the model satisfies the property or not, together with the running time for Spin.

```
bool x = true;
bool g = true;

process thread1:

void main()
{
    while( ? )
    {
        acquire(x);
        g = ?;
        release(x);
    }
}
```

```
process thread2:

void main()
{
    bool gi;
    bool done = false;

    while(!done)
    {
            gi = g;

        acquire(x);

        if(g == gi)
        {
            done=true;
        }

        release(x);
    }
}
```

Fig. 9 The program Ex3, which has fair nonterminating runs.

Table 2 Results of running SPIN on the budget-bounded sequentializations of the programs Ex1, Ex2 and Ex3, under different fairness conditions

Program	No fairness	Impartiality	Strong fairness
Ex1	No (< 0.01s)	Yes (0.02s)	Yes (0.02s)
Ex2	No (< 0.01s)	No (0.03s)	No (0.01s)
Ex3	No (< 0.01s)	No (0.18s)	No (0.1s)

For each program and fairness condition, we report whether the LTL formulas for termination under fairness is satisfied, and the time it takes for Spin to check the model

10 Conclusion

We have introduced the model of budget-bounded MPDS, in which each stack can perform an unbounded number of context switches if its height is below or equal to a given bound, while it is restricted to a finite number of context switches when its size is above that bound. We have shown that the reachability problem for budget-bounded MPDS is PSPACE-complete and that LTL model checking is EXPTIME-complete.

Moreover, we have proposed a code-to-code translation that inputs a concurrent program P and produces a sequential program P' such that, running P under the budget-bounded restriction yields the same set of reachable states as running P'. We have implemented a prototype tool, and applied it on a set of benchmarks. Furthermore, we have described how the budget-bounded translation can be used to check liveness properties. Specifically, we have checked termination under different fairness constraints.

Acknowledgments This work was supported in part by the Swedish Research Council and carried out within the Linnaeus centre of excellence UPMARC, Uppsala Programming for Multicore Architectures Research Center.

References

1. Abdulla PA, Atig MF, Rezine O, Stenman J (2012) Multi-pushdown systems with budgets. In: Cabodi G, Singh S (eds) FMCAD. IEEE, pp 24–33
2. Atig MF, Bollig B, Habermehl P (2008) Emptiness of multi-pushdown automata is 2ETIME-complete. In: DLT'08, LNCS, vol 5257. Springer, Berlin, pp 121–133
3. Atig MF, Bouajjani A, Emmi M, Lal A (2012) Detecting fair non-termination in multithreaded programs. In: Madhusudan P, S. A. Seshia (eds) CAV, lecture notes in computer science, vol 7358, pp 210–226
4. Atig MF, Bouajjani A, Kumar KN, Saivasan P (2012) Linear-time model-checking for multithreaded programs under scope-bounding. In: Chakraborty S, Mukund M (eds) ATVA, lecture notes in computer science. Springer, Berlin, pp 152–166
5. Atig MF, Kumar KN, Saivasan P (2013) Adjacent ordered multi-pushdown systems. In: Béal MP, Carton O (eds) Developments in language theory, lecture notes in computer science. Springer, Berlin, pp 58–69
6. Bouajjani A, Emmi M, Parlato G (2011) On sequentializing concurrent programs. In: SAS '11, proceedings of the 18th international symposium on static analysis. Springer, Berlin, pp 129–145
7. Bouajjani A, Esparza J, Maler O (1997) Reachability analysis of pushdown automata: application to model-checking. In: CONCUR, LNCS, vol 1243. Springer, Berlin, pp 135–150
8. Bouajjani A, Maler O (1996) Reachability analysis of pushdown automata. In: Proceedings of international workshop on verification of infinite-state systems (Infinity'96)
9. Bouajjani A, Müller-Olm M, Touili T (2005) Regular symbolic analysis of dynamic networks of pushdown systems. In: CONCUR'05, LNCS
10. Breveglieri L, Cherubini A, Citrini C, Crespi Reghizzi S (1996) Multi-push-down languages and grammars. Int J Found Comput Sci 7(3):253–292
11. Clarke E, Kroening D, Lerda F (2004) A tool for checking ANSI-C programs. In: TACAS, LNCS vol 2988, pp 168–176
12. Cordeiro L, Morse J, Nicole D, Fischer B (2012) Context-bounded model checking with esbmc 1.17—(competition contribution). In: TACAS, LNCS, vol 7214, pp 534–537
13. Eilenberg S (1974) Automata, languages, and machines. Academic Press Inc, Orlando, FL
14. Emmi M, Qadeer S, Rakamarić Z (2011) Delay-bounded scheduling. In: POPL. ACM, pp 411–422
15. Esbmc concurrency benchmark (2009)
16. Esparza J, Kiefer S, Schwoon S (2006) Abstraction refinement with Craig interpolation and symbolic pushdown systems. In: TACAS, LNCS vol 3920, pp 489–503
17. Finkel A, Sangnier A (2008) Reversal-bounded counter machines revisited. In: MFCS, LNCS, vol 5162. Springer, Berlin, pp 323–334
18. Ginsburg S (1975) Algebraic and automata-theoretic properties of formal languages. Elsevier Science Inc., New York, NY
19. Harrison M (1978) Introduction to formal language theory. Addison-Wesley Publishing Company, Reading, MA
20. Herlihy M, Shavit N (2008) The art of multiprocessor programming. Morgan Kaufmann, Los Altos, CA
21. Hopcroft JE, Ullman JD (1979) Introduction to automata theory, languages and computation. Addison-Wesley, Reading, MA
22. http://user.it.uu.se/jarst116/fmcad2012/ (2012)
23. La Torre S, Madhusudan P, Parlato G (2007) A robust class of context-sensitive languages. In: LICS. IEEE, pp 161–170
24. La Torre S, Madhusudan P, Parlato G (2009) Reducing context-bounded concurrent reachability to sequential reachability. In: CAV, LNCS, vol 5643. Springer, Berlin, pp 477–492
25. La Torre S, Madhusudan P, Parlato G (2010) Model-checking parameterized concurrent programs using linear interfaces. In: CAV, LNCS, vol 6174. Springer, Berlin, pp 629–644
26. La Torre S, Napoli M (2011) Reachability of multistack pushdown systems with scope-bounded matching relations. In: CONCUR, LNCS, vol 6901. Springer, Berlin, pp 203–218
27. La Torre S, Parlato G (2012) Scope-bounded multistack pushdown systems: fixed-point, sequentialization, and tree-width. Technical report, University of Southampton
28. Lahiri S, Lal A, Qadeer S (2012) Poirot microsoft research. http://research.microsoft.com/en-us/projects/verifierq/
29. Lal A, Reps T (2008) Reducing concurrent analysis under a context bound to sequential analysis. In: CAV, LNCS, vol 5123. Springer, Berlin, pp 37–51
30. Lal A, Reps TW (2009) Reducing concurrent analysis under a context bound to sequential analysis. Form Methods Syst Des 35(1):73–97
31. Lange M, Lei H (2009) To CNF or not to CNF ? An efficient yet presentable version of the CYK algorithm. Inf Didact 8:2008–2009

32. Morse J Personal communication
33. Musuvathi M, Qadeer S (2007) Iterative context bounding for systematic testing of multithreaded programs. In: PLDI. ACM, pp 446–455
34. Parlato G Personal communication
35. Pnueli A (1977) The temporal logic of programs. In: FOCS. IEEE, pp 46–57
36. Qadeer S, Rajamani SK, Rehof J (2004) Summarizing procedures in concurrent programs. In: ACM SIGPLAN Notices, vol 39, pp 245–255
37. Qadeer S, Rehof J (2005) Context-bounded model checking of concurrent software. In: TACAS, LNCS, vol 3440. Springer, Berlin, pp 93–107
38. Ramalingam G (2000) Context-sensitive synchronization-sensitive analysis is undecidable. ACM Trans Program Lang Syst 22(2):416–430
39. Sen K, Viswanathan M (2006) Model checking multithreaded programs with asynchronous atomic methods. In: CAV. LNCS 4144, pp 300–314
40. Suwimonteerabuth D (2009) Reachability in pushdown systems: algorithms and applications. Ph.D. thesis, Technische Universität München
41. Vardi MY (1988) A temporal fixpoint calculus. In: POPL, pp 250–259
42. Vardi MY (1995) Alternating automata and program verification. In: Computer science today, lecture notes in computer science, vol 1000. Springer, Berlin, pp 471–485
43. Vardi MY, Wolper P (1986) An automata-theoretic approach to automatic program verification (preliminary report). In: LICS. IEEE Computer Society LICS, pp 332–344