Multi-Pushdown Systems with Budgets

Parosh Aziz Abdulla, Mohamed Faouzi Atig, Othmane Rezine and Jari Stenman
Department of Information Technology
Uppsala University
Uppsala, Sweden

***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 [1]–[4]. In this paper, we propose the class of bounded-budget MPDS where we restrict them in the sense that each stack can perform an unbounded number of context switches if its size is below a given bound, and is restricted to a finite number of context switches when its size is above that bound. We show that the reachability problem for this subclass is PSPACE-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 bounded-budget restriction yields the same set of reachable states as running 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.**

I. 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., [1]–[3], [5]). On the other hand, 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. To overcome this barrier, several subclasses of multi-pushdown systems have been proposed and studied in the literature. 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 [1] 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 [6].

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 [7]. 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 MPDS *with budgets*. Intuitively, for each thread (or stack), we associate two values $k, d \in \mathbb{N} \cup \{\omega\}$ (where ω is the first limit ordinal) 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 identify two subclasses of MPDS with budgets. We call the first subclass *uniformly bounded-budget MPDS*. Here, we associate finite values to the stack depth $d \in \mathbb{N}$ and context budget $k \in \mathbb{N}$ for each thread (or stack). For this case, 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 uniformly bounded-budget MPDS to the non-emptiness test for the synchronous product of a finite set of depth-bounded pushdown automata (which is PSPACE-complete).

Then, we consider the class of *singly unbounded-budget* MPDS where we have at most one thread that can perform an unbounded number of context switches regardless of its stack depth, and all the other threads have finite values for their stack depth and context-budget bounds. We show that

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.

the reachability problem for this class is EXPTIME-complete. The lower bound is proved by a reduction from the non-emptiness test of the intersection of a pushdown automaton with several regular languages (which is EXPTIME-complete). For the upper bound, we show that the reachability problem for a singly unbounded-budget MPDS can be reduced to the emptiness problem for a pushdown automaton whose size is exponential.

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 uniformly bounded-budget 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 uniformly bounded-budget 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 [8], ESBMC [9], and CBMC [7]. We also compare our results to the ones obtained using concurrent verification tools, namely ESBMC [9] and POIROT [10]. In our experiments, bugs (i.e. violations of state invariants) appear for small bounds.

Related work: Our model is inspired by the work of Finkel and Sangnier [11], 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 [1]–[3], [5]. The closest model to multi-pushdown systems with budgets is Scope-bounded Multistack PushDown Systems (SMPDS for short) [5] 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 the corresponding one for uniformly 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 [5] 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 uniformly 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).

To the best of our knowledge there is no decidable subclass of multi-stack pushdown system similar to the class of singly unbounded-budget MPDS (for which we show the reachability problem to be EXPTIME-COMPLETE).

Our code-to-code translation follows the line of research on compositional reductions from concurrent to sequential programs [12]–[15]. Recently, La Torre and Parlato have proposed in [16] 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 [17].

II. PRELIMINARIES

In this section, we fix some basic definitions and notations that will be used in the rest of the paper. We assume that the reader is familiar with automata and language theory.

a) Notations: Let $\mathbb{N}$ denote the non-negative integers, and let $\mathbb{N}^k$ and $\mathbb{N}^k_\omega$ denote the set of vectors of dimension k over $\mathbb{N}$ and $\mathbb{N}\cup\{\omega\}$, respectively (ω representing 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^+) the set of all words (resp. non empty 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 have $|\epsilon| = 0$).

Let L be a language over Σ and let $w \in \Sigma^*$ be a word. We define $w.L = \{w.u \mid u \in L\}$. We define the *shuffle* operator $\sqcup\!\sqcup$ over two words inductively as $\sqcup\!\sqcup(\epsilon, w) = \sqcup\!\sqcup(w, \epsilon) = \{w\}$ and $\sqcup\!\sqcup(a.u', b.v') = a.(\sqcup\!\sqcup(u', b.v')) \cup b.(\sqcup\!\sqcup(a.u', v'))$. Given two languages 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 languages can be extended analogously.

b) Pushdown Automata: A pushdown automaton is defined by a tuple $\mathcal{P} = (Q, \Sigma, \Gamma, \Delta, I, F)$ where: (1) Q is a finite non-empty set of states, (2) Σ is the input alphabet, (3) Γ is the stack alphabet, (4) Δ is the 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 the set of initial states, and (6) $F \subseteq Q$ is the 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, σ, w) where $q \in Q$ is the current state, $\sigma \in \Sigma^*$ is the remaining input word, and $w \in \Gamma^*$ is the stack content. We define the binary relation $\Rightarrow_\mathcal{P}$ between configurations as follows: $(q, a\sigma, uw) \Rightarrow_\mathcal{P} (q', \sigma, u'w)$ iff $(q, u) \xrightarrow{a} (q', u')$. The transition relation $\Rightarrow^*_\mathcal{P}$ is the reflexive transitive closure of $\Rightarrow_\mathcal{P}$.

The language $L(\mathcal{P})$ accepted by $\mathcal{P}$ is defined by the set of finite words $\sigma \in \Sigma^*$ such that $(q_{\mathsf{init}}, \sigma, \epsilon) \Rightarrow^*_\mathcal{P} (q_{\mathsf{final}}, \epsilon, \epsilon)$ for some $q_{\mathsf{init}} \in I$ and $q_{\mathsf{final}} \in F$.

Let $d \in \mathbb{N}$. We define the transition relation $\rightarrow_{>d}$ between configurations of $\mathcal{P}$ as follows: $(q, \sigma, w) \rightarrow_{>d} (q', \sigma', w')$ if

and only if $(q,\sigma,w) \Rightarrow_{\mathcal{P}} (q',\sigma',w')$ and $|w'| > d$ or $|w| > d$. Intuitively, the transition relation $\rightarrow_{>d}$ can be performed only if the stack depth of the starting or 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: $(q,\sigma,w) \rightarrow_{\leq d} (q',\sigma',w')$ if and only if $(q,\sigma,w) \Rightarrow_{\mathcal{P}} (q',\sigma',w')$, $|w'| \leq d$ and $|w| \leq d$. Intuitively, the transition relation $\rightarrow_{\leq d}$ can only be performed when the stack depths of both the starting and target configurations are at most d. The transition relation $\rightarrow^*_{\leq d}$ is the reflexive transitive closure of $\rightarrow_{\leq d}$.

Given $d,k \in \mathbb{N}$, we define the relation $\rightarrow_{(k,d)}$ between configurations of depth d as follows: $(q,\sigma,w) \rightarrow_{(k,d)} (q',\sigma',w')$ if and only if $(q,\sigma,w) \rightarrow^*_{>d} (q',\sigma',w')$, $|\sigma| - |\sigma'| \leq k$, $w' = w$, and $|w| = d$. 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 words $\sigma \in \Sigma^*$ such that there is a sequence of configurations $c_0, c_1, \ldots, c_n$ where (1) c_0 is of the form (q_0,σ,ϵ) with $q_0 \in I$, (2) c_n is of the form (q_n,ϵ,ϵ) with $q_n \in F$, and (3) for every $i \in [1..n]$, we have $c_{i-1} \rightarrow_{(k,d)} c_i$ or $c_{i-1} \rightarrow^*_{\leq d} c_i$ holds. We call $L_{(k,d)}(\mathcal{P})$ the (k,d)-bounded language of $\mathcal{P}$.

We also define the language $L_{(-1,d)}(\mathcal{P})$ to be the set of words $\sigma \in \Sigma^*$ such that $(q_{\mathsf{init}},\sigma,\epsilon) \rightarrow^*_{\leq d} (q_{\mathsf{final}},\epsilon,\epsilon)$ 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 by the runs of $\mathcal{P}$ where the stack depth is always bounded by d.

Lemma 1: Let $d,k \in \mathbb{N}$ and $\mathcal{P}$ be a pushdown automaton. Then, it is possible to construct, in polynomial time, a pushdown automaton $\mathcal{P}'$ such that $L_{k+d}(\mathcal{P}') = L_{(k,d)}(\mathcal{P})$.

Proof: To prove this, it is sufficient to show 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 $L_k(\mathcal{P}') = L(\mathcal{P}) \cap \Sigma^{\leq k}$.

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 [4] 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 $\alpha \stackrel{\text{def}}{=} \alpha_0 \Rightarrow_G \alpha_1 \Rightarrow_G \cdots \Rightarrow_G \alpha_n$ is *k-bounded* if $|\alpha_i| \leq k$ 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 $L_k(\mathcal{P}) = L^{(k)}(G)$.

Proof: Since the context-free grammar G is in the normal form, we know that any k-bounded derivation have a derivation tree $\mathcal{T}$ in which the number of leaves is at most k. Moreover, any path of the derivation tree $\mathcal{T}$ has at most k nodes and each node has at most two outgoing edges. This implies that the we can construct a stateless pushdown automaton $\mathcal{P}'$ whose alphabet is exactly the set of variables of G. The derivation tree of the grammar G is simulated by the pushdown automaton $\mathcal{P}'$ in a leftmost way in the standard manner. Then, we construct a pushdown automaton $\mathcal{P}$ which results from the intersection of the pushdown automaton $\mathcal{P}'$ and the finite state automaton that recognizes words of length at most k. Now, it is easy to see that $L(\mathcal{P}) = L_k(\mathcal{P}) = L^{(k)}(G)$. ∎

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 $L^{(k)}(G) = L(\mathcal{P}) \cap \Sigma^{\leq k}$ [18]. Moreover, we can assume that this context-free grammar is in the normal form (based on the result in [4] 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, the pushdown automaton $\mathcal{P}'$ such that $L_k(\mathcal{P}) = L^{(k)}(G)$. Hence, we have $L_k(\mathcal{P}) = L^{(k)}(G) = L(\mathcal{P}) \cap \Sigma^{\leq k}$. ∎

Let $\mathcal{P}$ be a pushdown automaton. To prove Lemma 1, we construct, in polynomial time, a pushdown automaton $\mathcal{P}'$ such that $L_{k+d}(\mathcal{P}') = L_{(k,d)}(\mathcal{P})$ as follows: The pushdown automaton $\mathcal{P}'$ mimics the pushdown automaton $\mathcal{P}$ if the current stack depth of $\mathcal{P}$ is less or equal to d. Moreover, $\mathcal{P}'$ keeps track of the current stack depth in its control state. 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 $\mathcal{P}''$ (constructed using Lemma 2) from the pushdown automaton $\mathcal{P}''$ built from $\mathcal{P}$ by setting the initial state of $\mathcal{P}''$ to q and the final state of $\mathcal{P}''$ to q'. Moreover, we constrain the pushdown automaton $\mathcal{P}''$ such that the only first possible simulated transition of $\mathcal{P}$ is precisely the push transition t and the pushdown automaton $\mathcal{P}''$ halts when its stack is empty (except for the initial configuration). To detect that the stack of $\mathcal{P}''$ is empty, we can use a special symbol $\perp$ to mark the bottom of the stack. By construction, the pushdown automaton $\mathcal{P}'$ accepts exactly the words of length less than k (which are the set of words σ generated by the run of $\mathcal{P}$ of the form $(q,\sigma,\epsilon) \rightarrow_{(k,d)} (q',\epsilon,\epsilon)$ and where $|\sigma| \leq k$). Observe that $(q,\sigma\sigma',w) \rightarrow_{(k,d)} (q',\sigma,w)$ for some w such that $|w| = d$ holds if and only if $(q,\sigma,\epsilon) \rightarrow_{(k,d)} (q',\epsilon,\epsilon)$ holds. Since the stack depth of $\mathcal{P}'$ is at most k, the stack depth of $\mathcal{P}$ is at most

$d + k$. ∎

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 $L_{d_0}(\mathcal{P}) \cap \sqcup\!\sqcup(L_{d_1}(\mathcal{P}_1), \ldots, L_{d_n}(\mathcal{P}_n))$.

Lemma 4: 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 (particular case of depth-bounded pushdown automata) which is known to be PSPACE-hard. ∎

III. 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 into 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 moves 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 respectively denote q and w_i. 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 $j \in ([1..n] \setminus \{i\})$.
- **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 $j \in ([1..n] \setminus \{i\})$.

A *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 respectively c_0 and c_m.

Given two computations $\pi_1 = c_0 t_1 \cdots t_m c_m$ and $\pi_2 = c_{m+1} t_{m+2} \cdots t_k c_k$, 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 $\pi \stackrel{\text{def}}{=} c_0 t_1 c_1 t_2 c_2 \cdots t_m c_m t_{m+2} c_{m+2} t_{m+3} \cdots\cdots t_k c_k$.

In the following, we propose the class of *bounded-budget* computations of MPDS. Intuitively, with each stack $i \in [1..n]$, we associate two values $k_i, d_i \in \mathbb{N}_\omega$ such that the stack i can perform at most k_i contexts without its size going below d_i. A context is a run of $\mathcal{M}$ where operations are exclusive to one stack. (Observe that k_i and d_i could be ω.) Next, we describe bounded-budget computations formally.

Contexts: A *context* of a stack $i \in [1..n]$ is a computation of the form $\pi = c_0 t_1 c_1 t_2 \cdots t_m c_m$ 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 $j \in [1..m]$. Observe that every computation can be seen as the concatenation of a sequence of contexts $\pi_1 \bullet \pi_2 \bullet \ldots \bullet \pi_\ell$.

For any two contexts π_1 and π_2 of the stack i, we write $\pi_1 \bullet_i \pi_2$ to denote that $Stack_i(initial(\pi_2)) = Stack_i(target(\pi_1))$ (i.e., 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. Observe that if $\pi = \pi_1 \bullet \pi_2 \bullet \ldots \bullet \pi_m$ is a computation where each π_j is a context, then if $i_1 < i_2 < \ldots < i_k$ are all the indices j such that π_j is a context of stack i, then $\pi_{i_1} \bullet_i \pi_{i_2} \bullet_i \ldots \bullet_i \pi_{i_k}$.

A context $\pi = c_0 t_1 c_1 t_2 \cdots t_m c_m$ 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 $j \in [0..m]$, $|Stack_i(c_j)| \leq d$ (resp. $|Stack_i(c_j)| \geq d$). The definition is extended in the straightforward manner to sequences of contexts as follows: The sequence $\pi = \pi_1 \bullet_i \pi_2 \bullet_i \ldots \bullet_i \pi_m$ of compatible contexts of the stack i is of depth at most (resp. least) $d \in \mathbb{N}$ iff for every $j \in [1..m]$, π_j is of depth at most (resp. least) d.

Block: A *block* ρ of a stack $i \in [1..n]$ of size $m \in \mathbb{N}$ and depth $d \in \mathbb{N}$ is a sequence of compatible contexts of the form $c_0 t_0 \cdot \pi_1 \bullet_i \pi_2 \bullet_i \cdots \bullet_i \pi_m \cdot t_m c_m$ of stack i such that $|Stack_i(c_0)| = |Stack_i(c_m)| = d$ and π_j is a context of depth at least $d+1$ for all $j \in [1..m]$.

Budget-Bounded Computations: Intuitively, in a budget-bounded computation, we associate with each stack $i \in [1..n]$, a budget of contexts $k_i \in \mathbb{N}_\omega$ and depth bound $d_i \in \mathbb{N}_\omega$ such that if we consider a point in the computation where the stack i is of depth d_i and a symbol is being pushed into this stack (i.e., the depth of the stack now becomes $d_i + 1$), then this newly pushed stack symbol should be removed within k_i contexts involving this 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_i and blocks of size k_i and depth d_i. The formal definition is as follows:

Let π be a computation of $\mathcal{M}$. Let $\bar{k} = (k_1, k_2 \ldots, k_n) \in \mathbb{N}_\omega^n$ be the context-budget vector and $\bar{d} = (d_1, d_2, \ldots, d_n) \in \mathbb{N}_\omega^n$ the stack depth vector. We say that π is $(\bar{k}, \bar{d})$-*budget-bounded* if it can be written as a concatenation $\pi_1 \bullet \pi_2 \bullet \cdots \bullet \pi_m$ of 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 $\sigma_i = \pi^i_{i_1} \bullet_i \pi^i_{i_2} \bullet_i \cdots \bullet_i \pi^i_{m_i}$ (with $i_1 < i_2 < \cdots < m_i$) is the maximal sub-sequence of contexts in π belonging to the stack $i \in [1..n]$, then there is a sequence $\rho_i = \rho^i_1 \bullet_i \rho^i_2 \bullet_i \cdots \bullet_i \rho^i_{\ell_i}$ of contexts of depth at most d_i and blocks of size k_i and depth d_i such that $\sigma_i = \rho_i$.

By restricting the allowed bound vectors $(\bar{k}, \bar{d})$, we can distinguish two sub-classes of MPDS under budget-bounding.

Definition 2: A $(\bar{k}, \bar{d})$-budget-bounded computation π is a *singly unbounded-budget* computation if and only if there is at most one index $i \in [1..n]$ such that either $k_i = \omega$ or $d_i = \omega$.

In singly unbounded-budget computations, we have at most one stack $i \in [1..n]$ that can perform an unbounded number of contexts regardless of its depth. Any other stack j (with $i \neq j$) of $\mathcal{M}$ can at most perform a *finite* number consecutive contexts without its size going below a given *finite* bound.

Definition 3: A $(\bar{k}, \bar{d})$-budget-bounded computation π is a *uniformly bounded-budget* computation if and only if for every $i \in [1..n]$, we have $k_i \in \mathbb{N}$ and $d_i \in \mathbb{N}$.

Observe that in the case of uniformly bounded-budget computations, each stack $i \in [1..n]$ has a *finite* context-budget and a *finite* stack depth bound.

IV. 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 $\bar{k} = (k_1, k_2 \ldots, k_n) \in \mathbb{N}_\omega^n$ be the context-budget vector and $\bar{d} = (d_1, d_2, \ldots, d_n) \in \mathbb{N}_\omega^n$ the stack depth vector. The $(\bar{k}, \bar{d})$-*budget-bounded reachability problem* is to determine, for a given state $q_{\mathsf{final}} \in Q$, whether there is a $(\bar{k}, \bar{d})$-budget-bounded computation from the initial configuration $c^{\mathsf{init}}_{\mathcal{M}}$ to the configuration $(q_{\mathsf{final}}, \epsilon, \ldots, \epsilon)$. The input size of this problem is $n + |Q| + |\Gamma| + |\Delta| + k + d$, where k and d are the largest natural numbers (or 0 if they do not exist) in the vectors $\bar{k}$ and $\bar{d}$, respectively.

A. The Uniformly Budget-Bounded Reachability Problem

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

Theorem 5: The $(\bar{k}, \bar{d})$-budget-bounded reachability problem for MPDS is PSPACE-complete if for every $i \in [1..n]$, we have $k_i \in \mathbb{N}$ and $d_i \in \mathbb{N}$.

The rest of this section is devoted to the proof of Theorem 5. 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 uniformly budget-bounded computations to the non-emptiness test of the synchronization of depth-bounded pushdown automata which is PSPACE-complete (see Lemma 4). The idea behind the proof is the following: Let ρ be a $(\bar{k}, \bar{d})$-uniformly 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 $(\bar{k}, \bar{d})$-uniformly 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 interleavings of these summaries (for all stacks) that arises from $(\bar{k}, \bar{d})$-uniformly budget-bounded computation.

Before we present the details, we introduce some notations 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 $Summary(\pi) = (q, q')$ 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 sequence of contexts as follows: $Summary(\rho) = Summary(\pi_1) Summary(\pi_2) \cdots Summary(\pi_\ell)$. The function $Summary$ is also extended in straightforward manner to blocks and 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 $(\bar{k}, \bar{d})$-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 blocks of size k_i and depth d_i such that $\sigma_i = \rho_i$.

Then, it is easy to see that there is a consistent word in $\sqcup\!\sqcup(\{Summary(\sigma_1)\}, \ldots, \{Summary(\sigma_n)\})$. On the other hand, we can show that if for every stack $i \in [1..n]$, there is a compatible sequence σ_i of contexts of depth at most d_i and blocks of size k_i and depth d_i such that there is a consistent

word in $\sqcup\!\sqcup(\{Summary(\sigma_1)\},\ldots,\{Summary(\sigma_n)\})$, then M has a $(\bar{k},\bar{d})$-budget-bounded computation that reaches q_{final}.

Now, we can show that checking the existence of such a consistent word can be reduced, in polynomial time, to the non-emptiness test of the synchronization of depth-bounded pushdown automata (which is PSPACE-complete), and hence we obtain the completeness of Theorem 5.

Lemma 6: The problem of checking whether for every $i \in [1..n]$ there is a compatible sequence σ_i of contexts of depth at most d_i and blocks of size k_i and depth d_i for the stack i such that there is a consistent word in $\sqcup\!\sqcup(\{Summary(\sigma_1)\},\ldots,\{Summary(\sigma_n)\})$ can be reduced to the non-emptiness test of the synchronization of depth-bounded pushdown automata.

Proof (sketch). For every $i \in [1..n]$, let $L_i(\mathcal{M})$ be the set of words $Summary(\sigma_i)$ where σ_i is a compatible sequence of contexts of depth at most d_i and blocks of size k_i and depth d_i for stack i. Then, we can construct if $k_i > 0$ (resp. $k_i = 0$), in polynomial time, a pushdown automaton $\mathcal{P}_i$ whose (k_i, d_i)-bounded (resp. $(-1, d_i)$-bounded) language is precisely $L_i(\mathcal{M})$. 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 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_i or a context belonging to a block of size k_i and depth d_i (in the latter case $\mathcal{P}_i$ guesses also its position inside the block), then checks that all these assumptions hold when checking the feasibility of such contexts.

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 $L_{k_i+d_i}(\mathcal{P}_i') = L_i(\mathcal{M})$. Then, checking whether for every $i \in [1..n]$ there is a compatible sequence σ_i of contexts of depth at most d_i and blocks of size k_i and depth d_i for stack i such that there is a consistent word in $\sqcup\!\sqcup(\{Summary(\sigma_1)\},\ldots,\{Summary(\sigma_n)\})$ boils down to checking the non-emptiness of the language $L_0(\mathcal{P}_0) \cap \sqcup\!\sqcup(L_{k_1+d_1}(\mathcal{P}_1'),\ldots,L_{k_n+d_n}(\mathcal{P}_n'))$.

B. The Singly Unbounded-Budget Reachability Problem

In the following we show that the reachability problem for MPDS restricted only to singly unbounded-budget computations is EXPTIME-complete.

Theorem 7: The $(\bar{k},\bar{d})$-budget-bounded reachability problem for MPDS is EXPTIME-complete if there is at most one index $i \in [1..n]$ such that either $k_i = \omega$ or $d_i = \omega$.

The rest of this section is devoted to the proof of Theorem 7.

Lower bound: It is known that the following problem is EXPTIME-complete [19]: Given a pushdown automaton $\mathcal{P}$ recognizing a language L, and $n-1$ finite state automata $\mathcal{A}_i$ recognizing languages L_i, check the non-emptiness of $L \cap \bigcap_{i=2}^{n} L_i$. We can show that this problem can be reduced, in polynomial time, to the reachability problem for MPDS restricted only to singly unbounded-budget computations. The idea is the following: The first stack (with unbounded number of contexts regardless of its depth) is used to simulate $\mathcal{P}$, while each other stack $i \in [2..n]$ is used to simulate the automaton $\mathcal{A}_i$. Each stack $i \in [1..n]$ has a depth bound 1 and a context budget 0. Moreover, the stack i contains at most one symbol which is the current state of $\mathcal{A}_i$. (We assume here that the automaton $\mathcal{A}_i$ does not contain ϵ-transitions.)

The simulation proceeds as follows: An ϵ-labeled transition of $\mathcal{P}$ is simulated by a transition of the first stack while the other stacks remain unchanged. A labeled transition of $\mathcal{P}$ with an input symbol a is simulated by a transition of the first stack, followed by a sequence of transitions in which the other stacks are checked and then updated, one after the other, to ensure that each $\mathcal{A}_i$ is able to perform a transition labeled by a.

Upper bound: To prove the upper bound, we reduce the reachability problem for $\mathcal{M}$ restricted to singly unbounded-budget computations to the non-emptiness test of a pushdown automaton whose size is exponential in $\mathcal{M}$. Recall that the non-emptiness test for pushdown automata is in PTIME [20]. In the following we use the same notations and definitions as in the previous subsection. We assume here that only the first stack can perform an unbounded number of contexts regardless of its depth. Then, we can show that M has a $(\bar{k},\bar{d})$-budget-bounded computation that reaches q_{final} iff there is a compatible sequence σ_1 of contexts of the first stack and for every stack $i \in [2..n]$, there is a compatible sequence σ_i of contexts of depth at most d_i and blocks of size k_i and depth d_i such that there is a consistent word in $\sqcup\!\sqcup(\{Summary(\sigma_1)\},\ldots,\{Summary(\sigma_n)\})$. In fact, we can prove that checking the existence of such a consistent word can be reduced, in exponential time, to the non-emptiness test of a pushdown automaton, and hence we obtain the completeness of Theorem 7.

Lemma 8: The problem of checking whether there is a compatible sequence σ_1 of contexts of the first stack and for every $i \in [2..n]$, there is a compatible sequence σ_i of contexts of depth at most d_i and blocks of size k_i and depth d_i for the stack i such that there is a consistent word in $\sqcup\!\sqcup(\{Summary(\sigma_1)\},\ldots,\{Summary(\sigma_n)\})$ can be reduced to the non-emptiness test of a pushdown automaton $\mathcal{P}$ whose size is exponential in $\mathcal{M}$.

Proof (sketch). We can construct, in polynomial time, a pushdown automaton $\mathcal{P}_1$ whose language $L(\mathcal{P}_1)$ is precisely the set of words $Summary(\sigma_1)$ where σ_1 is a compatible sequence of contexts of the first stack. On the other hand, as in the previous subsection, we can easily construct, in polynomial time, for every $i \in [2..n]$, a pushdown automaton $\mathcal{P}_i$ whose $(-1, d_i + k_i)$-bounded language is precisely $L_i(\mathcal{M})$. Then, checking whether there is a compatible sequence σ_1 of contexts of the first stack and for every $i \in [2..n]$, there is a compatible sequence σ_i of contexts of depth at most d_i and blocks of size k_i and

depth d_i for the stack i such that there is a consistent word in $\sqcup\sqcup(\{Summary(\sigma_1)\},\ldots,\{Summary(\sigma_n)\})$ boils down to checking the non-emptiness of the language $L_0(\mathcal{P}_0) \cap \sqcup\sqcup(L(\mathcal{P}_1), L_{k_2+d_2}(\mathcal{P}_2),\ldots,L_{k_n+d_n}(\mathcal{P}_n))$. Finally, we can use standard automata constructions, to show that we can construct a pushdown automaton $\mathcal{P}$ such that $L(P) = L_0(\mathcal{P}_0) \cap \sqcup\sqcup(L(\mathcal{P}_1), L_{k_2+d_2}(\mathcal{P}_2),\ldots,L_{k_n+d_n}(\mathcal{P}_n))$. Moreover, the size of $\mathcal{P}$ is exponential in $\mathcal{M}$.

V. Other Subclasses of MPDS with Budget-Bounded Computations

In this section, we will briefly mention two other interesting subclasses of MPDS.

Definition 4 (Bounded stack-depth computations): We say that a $(\bar{k},\bar{d})$-budget-bounded computation π is a *$\bar{d}$-bounded stack-depth* computation if and only if for every $i \in [1..n]$, we have $k_i = 0$ and $d_i \in \mathbb{N}$.

In the case of a bounded stack-depth computation, the size of the i-th stack in each reachable configuration in π is always bounded by d_i.

Definition 5 (Unbounded-budget computations): We say that a $(\bar{k},\bar{d})$-budget-bounded computation π is an *unbounded-budget* computation if and only if there are at least two different stacks $i,j \in [1..n]$ such that $i \neq j$ and for every $\ell \in \{i,j\}$, either $k_\ell = \omega$ or $d_\ell = \omega$.

Observe that in the case of unbounded-budget computations, we have at least two different stacks that are allowed to perform an unbounded number of contexts regardless of their stack depth.

A. Known Results

In the following we recall some well-known results for the reachability problem for MPDS under budgets. More precisely, we consider bounded stack-depth and unbounded-budget computations.

Recall that, in the case of unbounded-budget computations, we have at least two different stacks that can perform unbounded number of context-switches regardless of their stack sizes. This implies that the reachability problem for MPDS restricted only to unbounded-budget computations is undecidable. This result can be shown using a reduction from the problem of checking non-emptiness of the intersection of two context-free languages (which is an undecidable problem).

Theorem 9: The $(\bar{k},\bar{d})$-budget-bounded reachability problem for MPDS is undecidable if there are at least two different stacks $i,j \in [1..n]$ such that $i \neq j$ and for every $\ell \in \{i,j\}$, either $k_\ell = \omega$ or $d_\ell = \omega$.

One way to overcome this undecidability barrier is to bound the depth of each stack (which corresponds to case of MPDS restricted to bounded-stack-depth computations). In this case, we show:

Theorem 10: The $(\bar{k},\bar{d})$-budget-bounded reachability problem for MPDS is PSPACE-complete if for every $i \in [1..n]$, we have $k_i = 0$ and $d_i \in \mathbb{N}$.

Proof: (*sketch)* Since in the case of a bounded stack-depth computation π, the depth of the i-th stack is bounded by d_i for any reachable configuration in π, the upper-bound of Theorem 10 can be obtained by an easy reduction to the emptiness problem for a Turing machine having n-tapes, and where each tape $i \in [1..n]$ has d_i cells.

The lower bound of Theorem 10 follows by a reduction from the non-emptiness test of the intersection of several regular languages (which is known to be PSPACE-hard). ■

VI. From Concurrent to Sequential

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 $\mathcal{P} = \langle G, \mathcal{F}_1 \cdots \mathcal{F}_m\rangle$, 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 $\mathcal{C} = \langle S, \mathcal{P}_1 \cdots \mathcal{P}_n\rangle$, consisting of a finite set S of *shared variables* and a sequence of processes.

Next, we describe an automatic transformation from a concurrent program $\mathcal{C} = \langle S, \mathcal{P}_1 \cdots \mathcal{P}_n\rangle$ 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.

A. 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.

B. 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 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.

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 $\mathcal{C} = \langle S, \mathcal{P}_1 \cdots \mathcal{P}_n \rangle$, we construct an *inlined concurrent program* $\mathcal{C}' = \langle S, I^{d_1}(\mathcal{P}_1) \cdots I^{d_n}(\mathcal{P}_n) \rangle$. 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.

2) Context switching: In [13], 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 $s_0 \cdots s_k$ 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 [13] 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.

3) Phases: An execution r of a single process in a concurrent program can be divided into a sequence $r_0, r_1, \ldots$ 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.

4) Transformation: For a concurrent program $\mathcal{C} = \langle S, \mathcal{P}_1 \cdots \mathcal{P}_n \rangle$, we first construct the corresponding inlined concurrent program $\mathcal{C}' = \langle S, I^{d_1}(\mathcal{P}_1) \cdots I^{d_n}(\mathcal{P}_n) \rangle$. 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 $S_t^0, \ldots, S_t^{2k+1}$ 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, 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. 2.

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. 2 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. 3.
- 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.

VII. Experimental Results

We have evaluated our approach on several examples, including one in which a big number of context switches is needed in order to reach a bad state. The experiments presented in Fig. 4 were run on a 2.2 GHz Intel Core i7 with 4 GB of memory. Most literature examples 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. It is in fact possible to extend our tool in order to parse C code.

We have implemented our code-to-code translation scheme in Haskell [21]. The scheme inputs a concurrent program P and produces a sequential program P' such that running P under the uniformly bounded-budget restriction yields the same set of reachable states as running P'. The sequential program is delivered in different languages, namely: Remopla for Moped [8], and a C-like language for CBMC [7] and ESBMC [9]. We use these three tools as back end to verify the obtained sequential code. Our experimental results are

```
process example:

int x = 0;

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

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

```
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. 1. Left: Transformation of Procedure Calls in Process t. Right: Context Switches in Procedures of Process t.

```
if(phase_t == 0){
  S_t^0 = S;
  S = S_t^0;
}
⋮
if(phase_t == k){
  S_t^2k = S;
  S = S_t^0;
}
```

```
if(!ret && ?){
  if(phase'_t == phase_t){
    if(phase_t == 0){
      S_t^1 = S;
    }
    ⋮
    if(phase_t == k){
      S_t^2k+1 = S;
    }

    phase_t =
    phase_t + 1;
    ret = true;
}
```

Fig. 2. Left: Transformation of Procedure Calls in Process t. Right: Context Switches in Procedures of Process t.

```
if(!ret && ?){
  if(phase'_t < phase_t){
    if(phase'_t == 0 && S == S_t^1){
      phase'_t = 1;
      S = S_t^2;
    }
    ⋮
    if(phase_t == k-1 && S == S_t^2k-1){
      phase'_t = k;
      S = S_t^2k;
    }
}
```

Fig. 3. Checking Reconstruction of Local State in Process t

Examples		Type of Analysis: Concurrent to Sequential k_1	MOPED	CBMC	ESBMC	Concurrent k_2	POIROT	k_3	ESBMC
Account	[22]	0	-.-	-.-	**1.1**	4	3.34	10	-.-
BigNum	[21]	0	**8.39**	-.-	-.-	26	-.-	234	-.-
Bluetooth3a	[21]	0	744.49	-.-	-.-	11	**18.38**	28	*FP*
Token Ring	[22]	0	**0.13**	0.2	0.18	1	2.72	4	1.47
Account Bad	[22]	0	-.-	0.48	1.41	1	2.13	4	**0.11**
BigNum Bad	[21]	0	**5.9**	13.4	239.4	26	-.-	26	-.-
Bluetooth1	[23]	1	4.18	**0.37**	1.28	2	1.92	5	*NF*
Bluetooth2	[23]	1	0.64	5.68	34.38	3	2.9	5	**0.5**
Bluetooth3b	[23]	1	1.26	**0.95**	5.0	2	2.5	5	*NF*
Infinite Loop 1	[24]	1	4.1	0.2	0.25	2	1.45	1	**0.08**
Infinite Loop 2	[24]	1	17.3	0.84	3.85	1	0.96	1	**0.09**
Token Ring Bad	[22]	0	**0.13**	0.16	0.26	2	2.74	4	0.27

Fig. 4. We report the running times of our experimentation results in seconds. We use the symbol -.- to denote a *timeout* (set to 900 seconds). 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. NF: Bug Not Found. FP: False Positive.

then compared to ones obtained using two verification tools for concurrent programs, namely ESBMC and POIROT [10]. The time required for sequentialization is negligible and not included in the results.

The table in Fig. 4 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 (26) is required to find 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 [25]. 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. Also, ESBMC, which did as well as our approach in terms of time, failed to find bugs in the faulty examples BLUETOOTH 1 and 3b regardless of the number of context-switches it was allowed.

VIII. CONCLUSION

We have introduced the class of MPDS with budgets where each stack can perform an unbounded number of context switches if its size 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 identified two decidable subclasses of MPDS with budgets, namely uniformly bounded-budget MPDS and singly unbounded-budget MPDS. We have shown that the reachability problem for uniformly bounded-budget MPDS and singly unbounded-budget MPDS is respectively PSPACE-complete and 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 uniformly bounded-budget restriction yields the same set of reachable states as running P'. We have implemented a prototype tool, and run it successfully on a set of benchmarks.

REFERENCES

[1] S. Qadeer and J. Rehof, "Context-bounded model checking of concurrent software," in *TACAS*, ser. LNCS, vol. 3440. Springer, 2005, pp. 93–107.

[2] S. La Torre, P. Madhusudan, and G. Parlato, "A robust class of context-sensitive languages," in *LICS*. IEEE, 2007, pp. 161–170.

[3] M. F. Atig, B. Bollig, and P. Habermehl, "Emptiness of multi-pushdown automata is 2ETIME-complete," in *DLT'08*, ser. LNCS, vol. 5257. Springer, 2008, pp. 121–133.

[4] M. Lange and H. Leiß, "To CNF or not to CNF ? An efficient yet presentable version of the CYK algorithm," *Informatica Didactica*, vol. 8, 2008-2010.

[5] S. La Torre and M. Napoli, "Reachability of multistack pushdown systems with scope-bounded matching relations," in *CONCUR*, ser. LNCS, vol. 6901. Springer, 2011, pp. 203–218.

[6] M. Musuvathi and S. Qadeer, "Iterative context bounding for systematic testing of multithreaded programs," in *PLDI*. ACM, 2007, pp. 446–455.

[7] E. Clarke, D. Kroening, and F. Lerda, "A tool for checking ANSI-C programs," in *TACAS*, ser. LNCS, vol. 2988, 2004, pp. 168–176.

[8] J. Esparza, S. Kiefer, and S. Schwoon, "Abstraction refinement with Craig interpolation and symbolic pushdown systems," in *TACAS*, ser. LNCS, vol. 3920, 2006, pp. 489–503.

[9] L. Cordeiro, J. Morse, D. Nicole, and B. F. 0002, "Context-bounded model checking with esbmc 1.17 - (competition contribution)." in *TACAS*, ser. LNCS, vol. 7214, 2012, pp. 534–537.

[10] S. Lahiri, A. Lal, and S. Qadeer, "Poirot," microsoft Research. [Online]. Available: http://research.microsoft.com/en-us/projects/poirot

[11] A. Finkel and A. Sangnier, "Reversal-bounded counter machines revisited," in *MFCS*, ser. LNCS, vol. 5162. Springer, 2008, pp. 323–334.

[12] A. Lal and T. W. Reps, "Reducing concurrent analysis under a context bound to sequential analysis," *Formal Methods in System Design*, vol. 35, no. 1, pp. 73–97, 2009.

[13] S. La Torre, P. Madhusudan, and G. Parlato, "Reducing context-bounded concurrent reachability to sequential reachability," in *CAV*, ser. LNCS, vol. 5643. Springer, 2009, pp. 477–492.

[14] ——, "Model-checking parameterized concurrent programs using linear interfaces," in *CAV*, ser. LNCS, vol. 6174. Springer, 2010, pp. 629–644.

[15] M. Emmi, S. Qadeer, and Z. Rakamarić, "Delay-bounded scheduling," in *POPL*. ACM, 2011, pp. 411–422.

[16] S. La Torre and G. Parlato, "Scope-bounded multistack pushdown systems: fixed-point, sequentialization, and tree-width," University of Southampton, Technical Report, march 2012.

[17] G. Parlato, personal communication.

[18] J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[19] A. Heußner, J. Leroux, A. Muscholl, and G. Sutre, "Reachability analysis of communicating pushdown systems," in *FOSSACS*, ser. LNCS, vol. 6014. Springer, 2010, pp. 267–281.

[20] A. Bouajjani, J. Esparza, and O. Maler, "Reachability analysis of pushdown automata: Application to model-checking," in *CONCUR*, ser. LNCS, vol. 1243. Springer, 1997, pp. 135–150.

[21] May 2012. [Online]. Available: http://user.it.uu.se/%7Ejarst116/fmcad2012/

[22] "Esbmc concurrency benchmark," Feb. 2009. [Online]. Available: http://users.ecs.soton.ac.uk/lcc08r/esbmc/concurrent-software-benchmarks.zip

[23] D. Suwimonteerabuth, "Reachability in pushdown systems: Algorithms and applications," Ph.D. dissertation, Technische Universität München, 2009.

[24] S. Qadeer, S. K. Rajamani, and J. Rehof, "Summarizing procedures in concurrent programs," 2004.

[25] J. Morse, personal communication.