Multi-Pushdown Systems with Budgets

Parosh Aziz Abdulla, Mohamed Faouzi Atig,
Othmane Rezine, and Jari Stenman

Uppsala University, Sweden

Abstract. We address the verification problem for concurrent programs modeled as multi-pushdown automata (MPDA). In general, they are Turing powerful and hence come along with undecidability of basic decision problems [15]. Therefore, several subclasses of MPDA have been proposed and studied in the literature [3, 13, 9, 2, 11]. In this paper, we mainly propose the class of bounded-budget MPDA where we restrict multi-pushdown automata, in the sense that each stack can perform a finite number of consecutive contexts without its size goes strictly below a given bound on that stack. We show that the reachability problem for this sub-class is Pspace-complete. Furthermore, we propose a code-to-code translation that takes as input 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'. We implement our analysis by a systematic code-to-code translation from multithreaded programs to sequential programs. By leveraging standard sequential analysis tools, we applied a prototype implementation on a set of benchmarks in order to show that our translation scheme is feasible.

1 Introduction

Writing and testing concurrent programs has proven to be hard. This is mainly due to the large number of interleavings that concurrent programs give rise to. For this reason, automatic analysis tools for concurrent programs are highly desirable. Unfortunately, analysis tools for concurrent programs based on traditional methods suffer even more from the state space explosion problem. In recent years, a number of tools have tried to leverage existing analysis tools for *sequential programs* by clever transformation schemes that automatically transform concurrent programs into sequential programs that emit the same behaviour. After this transformation, the sequential program can be analysed, and any errors in the program correspond to errors in the actual concurrent program. Following the intuition that most program errors manifest in a few number of context switches, these transformation schemes generally produce a sequential program that exactly simulates the concurrent one up to a certain bound k of context switches.

In this paper, we propose a new transformation scheme that takes an additonal parameter, namely a bound on the stack depth. In our transformation, a process may be preempted an unbounded number of times as long a the call

stack of that process is below the given bound. Whenever the call stack of the process goes above the given bound, the process is assigned a *budget* of k context switches. If at any point the process goes below the stack bound, this budget is reset.

Related work

2 Preliminaries

In this section, we introduce some basic definitions and notations that will be used in the rest of the paper.

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 ($\omega$ 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^+) 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. We use Σ_ϵ to denote the set $\Sigma \cup \{\epsilon\}$. Let u be a word over Σ. The length of u is denoted by $|u|$; we assume that $|\epsilon| = 0$.

Let S be a set of words over the alphabet Σ and let $w \in \Gamma^*$ be a word. We define $w.S = \{w.u \mid u \in S\}$. We define the *shuffle* 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 sets of words S_1 and S_2, we define shuffle over these sets as $\sqcup\!\sqcup(S_1, S_2) = \bigcup_{u \in S_1, v \in S_2} \sqcup\!\sqcup(u, v)$. The *shuffle* operator for multiple sets can be extended analogously.

Pushdown Automata A pushdown automaton is defined by a tuple $\mathcal{P} = (Q, \Sigma, \Gamma, \Delta, I, F)$ where: (1) Q is the 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 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 the binary relation $\Rightarrow_{\mathcal{P}}$.

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

Let $d \in \mathbb{N}$ be a natural number, 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 the binary relation $\rightarrow_{>d}$.

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 depth of the starting and target configuration is at most d. The transition relation $\rightarrow^*_{\leq d}$ is the reflexive transitive closure of the binary relation $\rightarrow_{\leq d}$.

Given natural numbers $d, k \in \mathbb{N}$, 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 $(q, \sigma, w) \rightarrow^*_{>d} (q', \sigma', w')$, $|\sigma| - |\sigma'| \leq k$, $|w'| = d$, and $|w| = d$. This means that the pushdown automaton can only reads k consecutive input symbols without its stack depth goes 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 $(q_n, \epsilon, \epsilon)$ 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 syntax) contains all the set of words accepted by the pushdown automaton $\mathcal{P}$ by the runs where the stack depth is always at most d.

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

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 MPDA are: (i) pushing a symbol into one stack, (ii) popping a symbol from one stack, or (iii) an internal action that changes its states while keeping the stacks unchanged.

Definition 2 (Multi-PushDown Systems). *A* multi-pushdown system *(MPDS) is a tuple $\mathcal{M} = (n, Q, \Gamma, \Delta, q_0)$ where $n \geq 1$ is the number of stacks, Q is the 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_0 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. 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 of $\mathcal{M}$. 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 a $(n+1)$-tuple $(q, w_1, \ldots, w_n)$ where $q \in Q$ is a state and for every $i \in [1..n]$, $w_i \in \Gamma_i^*$ is the content of the i-th stack of $\mathcal{M}$. We use $State(c)$ and $Stack_i(c)$, with $i \in [1..n]$, to denote respectively the state q and the stack content w_i. We denote by $c_{\mathcal{M}}^{\mathsf{init}} = (q_0, \epsilon, \epsilon, \ldots, \epsilon)$ the initial configuration of $\mathcal{M}$. The set of all configurations of $\mathcal{M}$ is denoted by $Conf(\mathcal{M})$.

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 $c' = (q', w'_1, \ldots, w'_n)$, and 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:** $t = (q, \gamma, a, q')$ 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 of the form $c_0 t_1 c_1 t_2 \cdots t_m c_m$ such that: (1) $c_0 = c$ and $c_m = c'$, and (2) $c_{i-1} \xrightarrow{t_i} c_i$ 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 $\pi_1 = c_0 t_1 c_1 t_2 c_2 \cdots t_m c_m$ and $\pi_2 = c_{m+1} t_{m+2} c_{m+2} t_{m+3} \cdots t_k c_k$, 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 general, multi-pushdown systems are Turing powerful resulting in the undecidability of all basic decision problems [15]. However, it is possible to obtain decidability for some problems, such a control state reachability, by restricting the allowed set of behaviors [13, 9, 2, 3, 10]. In the following, we purpose the class of *bounded-budget* computations of MPDS. Intuitively, for each stack $i \in [1..n]$, we associate two numbers $k_i, d_i \in \mathbb{N}_\omega$ such that the stack i can be at most active k consecutive contexts without its size goes strictly below 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 the context). (Observe that k_i and d_i could be the first limit ordinal ω.) Next, we describe formally bounded-budget computations.

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 every 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 the straightforward manner to sequence of contexts. Observe that if $\pi = \pi_1 \bullet \pi_2 \bullet \ldots \bullet \pi_m$, and each π_j is a context and if $i_1 < i_2 < \ldots < i_k$ are all the indices j such that π_j is a context of the 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 (reps. least) $d \in \mathbb{N}$ if and only if for every $j \in [0..m]$, $|Stack_i(c_j)| \leq d$ (reps. $|Stack_i(c_j)| \geq d$). 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 (reps. least) $d \in \mathbb{N}$ iff for every $j \in [1..m]$, π_j is of depth at most (reps. least) d.

Block A *block* ρ of a stack $i \in [1..n]$ of size $j \in \mathbb{N}$ and depth $d \in \mathbb{N}$ is a sequence of compatible contexts of the form $c_0 t_0 \cdot \pi_1 \bullet_i \pi_2 \bullet_i \cdots \bullet_i \pi_j \cdot t_j c_j$ of the stack i such that the following conditions are satisfied: (1) $|Stack_i(c_0)| = d$, (2) π_j is a context of depth at least $d+1$ for all $j \in [1..m]$, and (3)$|Stack_i(c_j)| = d$.

Budget-Bounded Computations Intuitively, in a budget-bounded computation, we associate for 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 i is now $d_i + 1$) 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 just a concatenation of context 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 $\sigma_i = \pi^i_{i_1} \bullet_i \pi^i_{i_2} \bullet_i \pi^i_{i_3} \bullet_i \cdots \bullet_i \pi^i_{m_i}$ (with $i_1 < i_2 < i_3 < \cdots < m_i$), $i \in [1..n]$, is the maximal sub-sequence of contexts in π belonging to the stack $i \in [1..n]$, then there is a sequence $\rho_i = \rho^i_1 \bullet_i \rho^i_2 \bullet_i \cdots \bullet_i \rho^i_{\ell_i}$, for each $i \in [1..n]$, of context 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 four interesting sub-classes of budget-bounded computations:

Definition 3 (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 bounded stack-depth computation, the size of the i-th stack, in each reachable configuration in π, is always bounded by d_i.

Definition 4 (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.

Definition 5 (Singly unbounded-budget computations). *We say that a $(\bar{k}, \bar{d})$-budget-bounded computation π is a* singly unbounded-budget *computation if and only if there is only one index $i \in [1..n]$ such that either $k_i = \omega$ or $d_i = \omega$.*

In the case of singly unbounded-budget computations, we must have exactly 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 $k_j \in \mathbb{N}$ consecutive contexts without its size goes below the given bound $d_j \in \mathbb{N}$.

Definition 6 (Uniformly Bounded-budget computations). *We say that 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 $i \in [1..n]$ can at most perform a *finite* number $k_i \in \mathbb{N}$ consecutive contexts without its size goes strictly below the given *finite* bound $d_i \in \mathbb{N}$.

4 The Budget-Bounded Reachability Problem for MPDS

We study in this section the decidability and complexity of the reachability problem for MPDS under budget-bounding. Let $\mathcal{M} = (n, Q, \Gamma, \Delta, q_0)$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 any even state $q_{final} \in Q$, whether there is a $(\bar{k}, \bar{d})$-budget-bounded computation π from the initial configuration $c_{\mathcal{M}}^{\mathsf{init}}$ such that $q_{final} = State(target(\pi))$.

In the following, we consider the reachability problem for MPDS under different budget-bounding policies.

4.1 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 7. *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 8. *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 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 8 can be obtained by an easy reduction to the emptiness problem for a Turing machine having n-tapes and such that each tape $i \in [1..n]$ has d_i cells.

The lower bound of Theorem 8 follows by a reduction from the non-emptiness test of the intersection of several regular languages (which is known to be PSPACE-hard). □

4.2 The Uniformly Budget-Bounded Reachability Problem

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

Theorem 9. *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 9. The lower bound follows immediately from Theorem 8. To prove the upper bound, we reduce the the reachability problem for MPDS restricted only to uniformly budget-bounded computations to the non-emptiness test of the intersection of several regular languages recognized by bounded-depth pushdown automata (which is known to be PSPACE-complete). The key 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 the stack i (say ρ_i) is a compatible sequence of contexts of the form $\pi_1^i \bullet_i \pi_2^i \bullet_i \cdots \bullet_i \pi_{m_i}^i$. Since the communication between stacks is done via control states, we can summarize each context π_j^i (with $j \in [1..m_i]$) by a pair of states of the form $(q_j^i, {q'}_j^i)$ where q_j^i (resp. ${q'}_j^i$) is the valuation of the control state at the beginning (resp. end) of the context π_j. Then, we can summarize the stack computation ρ_i by the summary sequence $(q_1, q_1')(q_2^i, q_2^i) \cdots (q_{m_i}^i, q_{m_i}^i)$. 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 the stack i along a $(\bar{k}, \bar{d})$-uniformly budget-bounded computation along 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 real $(\bar{k}, \bar{d})$-uniformly budget-bounded computation.

and We can summarize of depth at most d_i and blocks of size k_i and depth d_i.

4.3 The Singly Budget-Bounded Reachability Problem

Theorem 10. *The reachability problem for singly unbounded-budget MPDS is EXPTIME-complete.*

5 Concurrent Programs

The building blocks of concurrent programs are processes, procedures and statements. The grammar of statements is given in Figure 1. We consider variables ranging over some (potentially infinite) data domain $\mathbb{D}$ and assume 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 then 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.

$\langle statement \rangle$ ::= **skip ;**
| $\langle identifier \rangle$ **=** $\langle value \rangle$ **;**
| **assume (** $\langle expression \rangle$ **) ;**
| **assert (** $\langle expression \rangle$ **) ;**
| **call** $\langle identifier \rangle$ **(** $\langle expression \rangle^*$ **) ;**
| **return ;**
| **if (** $\langle expression \rangle$ **) {** $\langle statement \rangle^+$ **} else {** $\langle statement \rangle^+$ **}**
| **while (** $\langle expression \rangle$ **) {** $\langle statement \rangle^+$ **}**
| **atomic {** $\langle statement \rangle^+$ **}**

Fig. 1. BNF grammar of statements

6 From Concurrent to Sequential

In this section, 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. Therefore, whenever a process contains no procedure calls, the simulation needs to be exact for that process.

6.1 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. In other words, 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 the program counter of that process. In the scheduling loop, each statement is enclosed by an `if`-statement which checks the corresponding program counter against increasing values and a nondeterministic boolean `?`. 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 Figure 2. The sequential program which simulates this is shown in Figure 3. It is easy to see that the program in Figure 3 simulates all the behavior of the concurrent program, including the interleaving `x = x + 2`, `x = 1`, `assert(x != 1)`, `x = 2` in which the assertion fails.

6.2 Programs with procedure calls

```
int x = 0;

process p1:

void main()
{
  x = 1;
  x = 2;
}

process p2:

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

Fig. 2. Example Program

When 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 $\mathcal{P}_i$ 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.

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.

Context switching In [La Torre, Madhusudan, Parlato], 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_o. Assume that the 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 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 [La Torre, Madhusudan, Parlato] 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.

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 that a process is always above its stack bound, the execution of that process will consist of $k+1$ 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 a process'es local state by executing from the beginning, we also store a virtual phase `phase'`, which is updated both during the reconstruction and the actual execution.

```
int pc1 = 1;
int pc2 = 1;
int running;

int x = 0;

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

    // schedule a process
    if( ?)
    {
      running = 1;
    }
    else
    {
      running = 2;
    }

    // process 1
    if(running == 1)
    {
      if(pc1 == 1 && ?)
      {
        x = 1; progress = true;
      }
      if(pc1 == 2 && ?)
      {
        x = 2; progress = true;
      }
    }

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

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. Until that procedure call, we do not modify the phase variables.

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 $\mathcal{S}$, the sets $S_0, \ldots, S_k$ 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, we insert a code block that, if the program is not recomputing the local state, saves the current values of the global variables in S_0. This code block is shown in Figure 4.

```
if(phase_t == 0)
{
  S_0 = S;
  S = S_0;
}
⋮
if(phase_t == k)
{
  S_2k = S;
  S = S_0;
}
```

Fig. 4. Transformation of Procedure Calls in Process t

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

- To simulate context switches, we add the code shown in Figure 5 between every statement that is *visible to the outside*.
- Between the same statements, we also add code to detect whether the local state has been reconstructed or not. The code for this is shown in Figure 6.
- 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, the execution is aborted.

```
if(!ret && ?)
{
  if(phase'_t == phase_t)
  {
    if(phase_t == 0)
    {
      S_1 = S;
    }
    ⋮
    if(phase_t == k)
    {
      S_{2k+1} = S;
    }

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

Fig. 5. Context Switches in Procedures of Process t

7 Experimental Results

```
if(!ret && ?)
{
  if(phase'_t < phase_t)
  {
    if(phase'_t == 0 && S
        == S_1)
    {
      phase'_t = 1;
      S = S_2;
    }
    ⋮
    if(phase_t == k-1 &&
        S == S_{2k-1})
    {
      phase'_t = k;
      S = S_{2k};
    }
}
```

Fig. 6. Checking Reconstruction of Local State in Process t

Our tool delivers, from a concurrent program specification, a sequential program corresponding to the $(K, L) - simulation$ of the concurrent one. First, the concurrent program, written in the previously defined syntax, is parsed into an internal abstract syntax tree representation. This representation is then transformed into a sequential internal representation following the transformation procedure described in previous section. Finally, the sequential abstract syntax tree is translated into one of the sequential program verification tool languages, namely: REMOPELA for MOPED [6], BPL for BOOGIE [12] and C-like languages for CBMC [4], ESBMC [5], UFO [1] and BLAST [8]. The resulting sequential program will then be assessed by those various sequential program verification and bug finding tools (cf. fig.7). Thus, our implementation can be easily extended to implement other transformation schemes or in order to use other back end sequential verification tools.

We compared the results obtained from our sequentialization approach to results obtained using concurrent verification tools, namely Poirot and Esbmc.

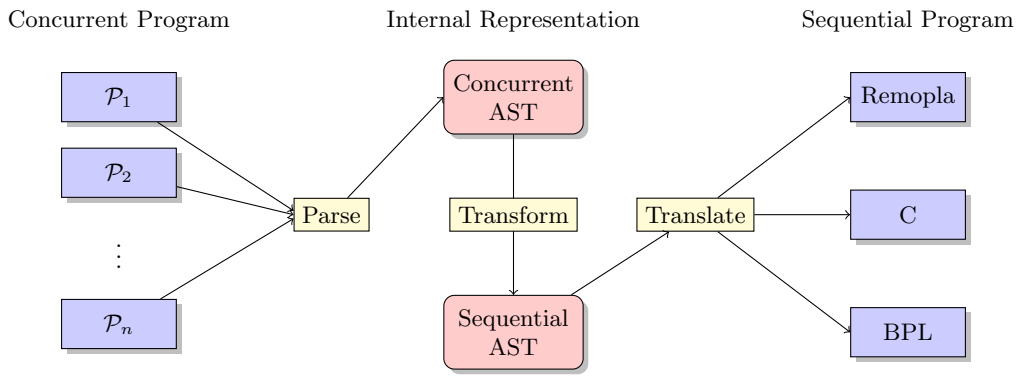

Fig. 7. Overview of the tool

The sequentialized code was verified using two sequential verification tools CBMC and MOPED. Moped is a symbolic model checker that verifies boolean and integer programs, while CBMC analyzes C programs up to a certain depth and uses SMT solvers as a back end.

In order to even the verification results obtained from the experiments, we present them as:

C, time Program proved to be correct in *time* seconds.
C(k), time Program proved to be correct, up to k context switches with 0 inlining, in *time* seconds.
BUG, time A bug was found, i.e. assertion violation was raised, in *time* seconds.
NA The verifier timed out, or stopped: It was not able to find the bug or to prove program correctness up to k context switches.

We applied our approach on a subset of examples composed of the `Windows NT BlueTooth driver` example [16], the `account` and the `token ring` examples taken from the `ESBMC` concurrency benchmark, the `non-terminating` example [14], the `x++` example [7] and we also constructed an example (cf. fig.8) to illustrate the benefit of our approach, an example in which a big number of context swtichs is needed in order to reach the bad state.

Most of these litterature examples were written in pseudo code or C-like code. In order to run them, we manually translated them to our syntax. This operation can be automatized. It is in fact possible to extend our tool in order to directly parse `C` code, using `CIL` or `LLVM` toolchains for exmaple.

The experiments presented in fig.9 were run on a (JARI_COMPUTER_ARCHI).

```
int z = 0;
bool stop1;
bool stop2;

process thread1:
void main()
{
  stop1 = false;

  while(!stop1)
  {
    z = z + 1;
    stop1 = true;
    stop2 = false;
  }
  assert(z < 50);
}

process thread2:
void main()
{
  stop2 = false;

  while(!stop2)
  {
    z = z + 1;
    stop2 = true;
    stop1 = false;
  }
}
```

Fig. 8. Big Number Example

	Type of Analysis			
	Concurrent to Sequential		Concurrent	
Examples	Moped	Cbmc	Poirot	Esbmc
Account				
Account Bad				
BigNum				
Bluetooth1				
Bluetooth2				
Bluetooth3a				
Bluetooth3b				
Non-terminating				
Non-terminating bug_1				
Non-terminating bug_2				
Token ring				
Token ring Bad				
x++				

Fig. 9. Verification results

8 Conclusion

References

1. Albarghouthi, A., Gurfinkel, A., Chechik, M.: From under-approximations to over-approximations and back. In: TACAS. pp. 157–172 (2012)
2. Atig, M.F., Bollig, B., Habermehl, P.: Emptiness of multi-pushdown automata is 2ETIME-complete. In: Proceedings of DLT'08. LNCS, vol. 5257, pp. 121–133. Springer (2008)
3. Breveglieri, L., Cherubini, A., Citrini, C., Crespi Reghizzi, S.: Multi-push-down languages and grammars. Intl. Journal of Foundations of Computer Science 7(3), 253–292 (1996)
4. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: Jensen, K., Podelski, A. (eds.) Tools and Algorithms for the Construction and

Analysis of Systems (TACAS 2004). Lecture Notes in Computer Science, vol. 2988, pp. 168–176. Springer (2004)
5. Cordeiro, L., Morse, J., Nicole, D., 0002, B.F.: Context-bounded model checking with esbmc 1.17 - (competition contribution). In: Flanagan, C., Knig, B. (eds.) TACAS. Lecture Notes in Computer Science, vol. 7214, pp. 534–537. Springer (2012)
6. Esparza, J., Kiefer, S., Schwoon, S.: Abstraction refinement with Craig interpolation and symbolic pushdown systems. In: Hermanns, H., Palsberg, J. (eds.) Proceedings of the 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). Lecture Notes in Computer Science, vol. 3920, pp. 489–503. Vienna, Austria (2006)
7. Ghafari, N., Hu, A.J., Rakamarić, Z.: Context-bounded translations for concurrent software: an empirical evaluation. In: Proceedings of the 17th international SPIN conference on Model checking software. pp. 227–244. SPIN'10, Springer-Verlag, Berlin, Heidelberg (2010), http://dl.acm.org/citation.cfm?id=1928137.1928160
8. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Software verification with blast. pp. 235–239. Springer (2003)
9. La Torre, S., Madhusudan, P., Parlato, G.: A robust class of context-sensitive languages. In: Proceedings of LICS. pp. 161–170. IEEE (2007)
10. La Torre, S., Napoli, M.: Reachability of multistack pushdown systems with scope-bounded matching relations. In: CONCUR. Lecture Notes in Computer Science, vol. 6901, pp. 203–218. Springer (2011)
11. Lange, M., Leiß, H.: To CNF or not to CNF ? An efficient yet presentable version of the CYK algorithm. Informatica Didactica 8 (2008-2010)
12. Leino, K.R.M.: This is boogie 2 (2008)
13. Qadeer, S., Rehof, J.: Context-bounded model checking of concurrent software. In: TACAS. LNCS, vol. 3440, pp. 93–107. Springer (2005)
14. Qadeer, S., Rajamani, S.K., Rehof, J.: Summarizing procedures in concurrent programs (2004)
15. Ramalingam, G.: Context-sensitive synchronization-sensitive analysis is undecidable. ACM Trans. Program. Lang. Syst. 22(2), 416–430 (2000)
16. Suwimonteerabuth, D.: Reachability in Pushdown Systems: Algorithms and Applications. Ph.D. thesis, Technische Universität München (2009)