Synchronisation- and Reversal-Bounded Analysis of Multithreaded Programs with Counters

Matthew Hague[1,2] and Anthony W. Lin[2]

[1] LIGM (Université Paris-Est), LIAFA (Université Paris Diderot) & CNRS
[2] Oxford University, Department of Computer Science

Abstract. We study a class of concurrent pushdown systems communicating by both global synchronisations and reversal-bounded counters, providing a natural model for multithreaded programs with procedure calls and numeric data types. We show that the synchronisation-bounded reachability problem can be efficiently reduced to the satisfaction of an existential Presburger formula. Hence, the problem is NP-complete and can be tackled with efficient SMT solvers such as Z3. In addition, we present optimisations to make our reduction practical, e.g., heuristics for removing or merging transitions in our models. We provide optimised algorithms and a prototypical implementation of our results and perform preliminary experiments on examples derived from real-world problems.

1 Introduction

Pushdown systems (PDS) are a popular abstraction of sequential programs with recursive procedure calls. Verification problems for these models have been extensively studied (e.g. [7, 17]) and they have been successfully used in the model checking of sequential software (e.g. [3, 5, 37]).

However, given the ubiquity and growing importance of concurrent software (e.g. in web-servers, operating systems and multi-core machines), coupled with the inherent non-determinism and difficulties in anticipating all concurrent interactions, the verification of concurrent programs is a pressing problem. In the case of concurrent pushdown systems, verification problems quickly become undecidable [33]. Because of this, much research has attempted to address the undecidability, proposing many different approximations, and restrictions on topology and communication behaviour (e.g. [29, 8–10, 35, 34, 21, 25]). A technique that has proved popular in the literature is that of *bounded context-switches* [34].

Bounded context-switching uses the observation that many real-world bugs require only a small number of inter-thread communications. It is known that, if the number of communications is bounded to a fixed k, reachability checking of pushdown systems becomes NP-complete [26]. The utility of this approach has been demonstrated by several successful implementations (e.g. [4, 30, 36]).

In addition to recursive procedure calls, numeric data types are an important feature of programs. By adding counters to pushdown systems one can accurately model integer variables and, furthermore, abstract certain data structures

– such as lists – by tracking their size. It is well known that finite-state machines augmented even with only two counters leads to undecidability of the simplest verification problems. One way to retain decidability of reachability is to impose an upper bound r on the number of reversals between incrementing and decrementing modes for each counter (cf. [12, 23]).

This restriction can be viewed in at least two ways (cf. [12, 24]). First, in the spirit of bounded-context switches, it provides a generalisation of bounded model checking – a successful verification technique which exploits the fact that many bugs occurring in practice are "shallow" (cf. [14]). Secondly, many counting properties — such as checking the existence of a computation where the number of calls to the functions f_1, f_2, f_3, and f_4 are the same — require no reversals (e.g. the number of memory allocations equals the number of frees). Similar counting properties (and their model checking problems) have been studied in many other contexts (cf. [27] and references therein).

In this paper, we study the problem of verifying reachability over a program model incorporating concurrency, numeric data types, and recursions. Our contributions are as follows:

1. We propose a concurrent extension of pushdown systems with reversal-bounded counters that communicate through shared counters and global synchronisations, and prove that the notion of global synchronisations subsumes context-bounded model checking.
2. We show that reachability checking for these systems is in NP, by reduction to existential Presburger, handled by efficient SMT solvers such as Z3 [13].
3. We provide several new optimisation techniques, including a minimisation routine for pushdown systems, that are crucial in making our reductions feasible in practice. These techniques keep the size of the computation objects small throughout reduction, while also producing smaller output formulas.
4. Finally, we provide two optimised, prototypical tools using these techniques. The first translates a simple programming language into our model, while the second performs our reduction to existential Presburger. We demonstrate the efficacy of our tools on several real-world problems.

The tool implementations and benchmarks can be obtained from the authors' homepages.

Related Work. In recent work [20], we showed that reachability analysis for pushdown systems with reversal-bounded counters is NP-complete. We provided a prototypical implementation of our algorithm and obtained encouraging results on examples derived from Linux device drivers.

Over reversal-bounded counter systems (without stack), reachability is NP-complete but becomes NEXP-complete when the number of reversals is given in binary [22]. On the other hand, when the numbers of reversals and counters are fixed, the problem is solvable in P [19]. The techniques developed by [19, 22], which reason about the maximal counter values, are very different to our techniques, which exploit the connection to Parikh images of pushdown automata (first explicated in Ibarra's original paper [23] though not in a way that gives optimal complexity or a practical algorithm).

Context-bounded model checking was introduced in 2005 by Qadeer and Rehof [34, 8, 32]. It has then been used in many different settings and many different generalisations have been proposed. For example, one may consider phase-bounds [38], ordered multi-stack machines [1], bounded languages [18], dynamic thread creation [2] and more general approaches [28].

In recent, independent work, Esparza *et al.* used a reduction to existential Presburger to tackle a generalisation of context-bounded reachability checking for multithreaded programs without counters [15]. Their work, however, does not allow the use of counters and it is not clear whether our global synchronisation conditions can be simulated succinctly in their framework.

Organisation. In S2, we define the models that we study. We prove decidability of the synchronisation-bounded reachability problem in S3. In S4 we show that synchronisation-bounded model checking subsumes context-bounded model checking. Our optimisations are presented in S5. In S6 we describe our implementation and experimental results. Finally, we conclude in S7.

2 Model Definition

In this section, we define the models that we study. For a vector $\boldsymbol{v} = (v_1, \ldots, v_n)$, we write $\boldsymbol{v}(i)$ to access v_i. For a formula θ over variables $(x_1, \ldots, x_n)$ we write $\theta(v_1, \ldots, v_n)$ to substitute the values $v_1, \ldots, v_n$ for the variables $x_1, \ldots, x_n$ respectively. Given an alphabet $\Gamma = \{\gamma_1, \ldots, \gamma_m\}$ and a word $w \in \Gamma^*$, we write $\mathbb{P}(w)$ to denote a tuple with $|\Gamma|$ entries where the ith entry counts the number of occurrences of γ_i in w. Given a language $\mathcal{L} \subseteq \Gamma^*$, we write $\mathbb{P}(\mathcal{L})$ to denote the set $\{ \mathbb{P}(w) \mid w \in \mathcal{L} \}$. We say that $\mathbb{P}(\mathcal{L})$ is the *Parikh image* of $\mathcal{L}$.

Pushdown Automata. A *pushdown automaton* $\mathcal{P}$ is a tuple $(\mathcal{Q}, \Sigma, \Gamma, \Delta, q_0, \mathcal{F})$ where $\mathcal{Q}$ is a finite set of control states, Σ is a finite stack alphabet with a special bottom-of-stack symbol $\bot$, Γ is a finite output alphabet, $q_0 \in \mathcal{Q}$ is an initial state, $\mathcal{F} \subseteq \mathcal{Q}$ is a set of final states, and $\Delta \subseteq (\mathcal{Q} \times \Sigma) \times \Gamma^* \times (\mathcal{Q} \times \Sigma^*)$ is a finite set of transition rules. We will denote a transition rule $((q, a), \boldsymbol{\gamma}, (q', w'))$ using the notation $(q, a) \overset{\boldsymbol{\gamma}}{\hookrightarrow} (q', w')$. Note that $\boldsymbol{\gamma} \in \Gamma^*$ is a sequence of output characters. This is for convenience, and optimisation. We can reduce this to single output characters using intermediate control states or stack characters. Note, a *pushdown system* is a pushdown automaton without a set of final states.

A configuration of $\mathcal{P}$ is a tuple (q, w), where $q \in \mathcal{Q}$ and $w \in \Sigma^*$ are the control state and stack contents. We say that a configuration (q, aw) has a *head* q, a. There exists a transition $(q, aw) \xrightarrow{\boldsymbol{\gamma}} (q', w'w)$ of $\mathcal{P}$ whenever $(q, a) \overset{\boldsymbol{\gamma}}{\hookrightarrow} (q', w') \in \Delta$. We call a sequence $c_0 \xrightarrow{\gamma_1} c_1 \xrightarrow{\gamma_2} \cdots \xrightarrow{\gamma_m} c_m$ a *run* of $\mathcal{P}$. It is accepting if $c_0 = (q_0, \bot)$ and $c_m = (q, w)$ with $q \in \mathcal{F}$. Let $\mathcal{L}(\mathcal{P})$ be the set of words labelling accepting runs. Finally, we write $c \to^* c'$ if there is a run from c to c'.

Pushdown Systems with Counters. A pushdown system with counters is a pushdown system which, in addition to the control states and the stack, has a number of counter variables. These counters may be incremented, decremented and compared against constants (given in binary).

An *atomic counter constraint* on counter variable $X = \{x_1, \ldots, x_n\}$ is an expression of the form $x_i \sim c$, where $c \in \mathbb{Z}$ and $\sim \in \{<, >, =\}$. A *counter constraint* $\theta(x_1, \ldots, x_n)$ on X is a boolean combination of atomic counter constraints on X. Let $Const_X$ denote the set of counter constraints on X.

A *pushdown system with n counters* (n-PDS) $\mathcal{P}$ is a tuple $(\mathcal{Q}, \Sigma, \Gamma, \Delta, X)$ where $\mathcal{Q}$ is a finite set of control states, Σ is a finite stack alphabet, Γ is a finite output alphabet, $X = \{x_1, \ldots, x_n\}$ is a set of n counter variables, and $\Delta \subseteq (\mathcal{Q} \times \Sigma \times Const_X) \times \Gamma^* \times (\mathcal{Q} \times \Sigma^* \times \mathbb{Z}^n)$ is a finite set of transition rules. We will denote a rule $((q, a, \theta), \boldsymbol{\gamma}, (q', w', \boldsymbol{u}))$ using $(q, a, \theta) \overset{\gamma}{\hookrightarrow} (q', w', \boldsymbol{u})$.

A configuration of $\mathcal{P}$ is a tuple $(q, w, \boldsymbol{v})$, where $q \in \mathcal{Q}$ is the current control state, $w \in \Sigma^*$ is the current stack contents, and $\boldsymbol{v} = (v_1, \ldots, v_n) \in \mathbb{N}^n$ gives the current valuation of the counter variables $x_1, \ldots, x_n$ respectively. There exists a transition $(q, aw, \boldsymbol{v}) \xrightarrow{\gamma} (q', w'w, \boldsymbol{v}')$ of $\mathcal{P}$ whenever

1. $(q, a, \theta) \overset{\gamma}{\hookrightarrow} (q', w', \boldsymbol{u}) \in \Delta$, and
2. $\theta(\boldsymbol{v}(1), \ldots, \boldsymbol{v}(n))$ is true, and
3. $\boldsymbol{v}'(i) = \boldsymbol{v}(i) + \boldsymbol{u}(i) \geq 0$ for all $1 \leq i \leq n$.

Communicating Pushdown Systems with Counters. Given $\mathcal{Q}_1, \ldots, \mathcal{Q}_m$, let $Y = \{y_1, \ldots, y_m, y'_1, \ldots, y'_m\}$ be a set of control state variables such that, for each i, y_i, y'_i range over $\mathcal{Q}_i$. Then, an *atomic state constraint* is of the form $y_i = q$ for some $y_i \in Y$ and $q \in \mathcal{Q}_i$. A *synchronisation constraint*, written $\delta(y_1, \ldots, y_m, y'_1, \ldots, y'_m)$, is a boolean combination of atomic state constraints. For example, let $n = 3$ and consider the constraint

$$\begin{array}{l}(y_1 = q_1 \wedge (y'_1 = q_1 \wedge y'_2 = q_2 \wedge y'_3 = q_3)) \;\vee \\ (y_1 = r_1 \wedge (y'_1 = r_1 \wedge y'_2 = r_2 \wedge y'_3 = r_3)) \;.\end{array}$$

This allows synchronisations where, whenever the first process has control state q_1, the other processes can simultaneously move to q_i (for all $1 \leq i \leq 3$), whereas, if process one has control state r_1, the processes move to states r_i instead. Let $StateCons_{\mathcal{Q}_1, \ldots, \mathcal{Q}_m}$ be the set of synchronisation constraints for $\mathcal{Q}_1, \ldots, \mathcal{Q}_m$.

Definition 1 (n-SyncPDSr). *Given a finite output alphabet Γ and set of n counter variables X, a system of* communicating pushdown systems with n counters $\mathbb{C}$ *is a tuple $(\mathcal{P}_1, \ldots, \mathcal{P}_m, \Delta_g, X, r)$ where, for all $1 \leq i \leq m$, $\mathcal{P}_i$ is a pushdown system $(\mathcal{Q}_i, \Sigma_i, \Gamma, \Delta_i, X)$ with n counters, and $\Delta_g \subseteq StateCons_{\mathcal{Q}_1, \ldots, \mathcal{Q}_m} \times Const_X \times \mathbb{Z}^n$ is a finite set of synchronisation constraints, and $r \in \mathbb{N}$ is a natural number given in unary.*

Notice that a system of communicating pushdown systems share a set of counters. A configuration of such a system is a tuple $(q_1, w_1, \ldots, q_m, w_m, \boldsymbol{v})$ where each $(q_i, w_i, \boldsymbol{v})$ is a configuration of $\mathcal{P}_i$. We have $(q_1, w_1, \ldots, q_m, w_m, \boldsymbol{v}) \overset{\gamma}{\Rightarrow} (q'_1, w'_1, \ldots, q'_m, w'_m, \boldsymbol{v}')$ whenever,

1. for some $1 \leq i \leq m$, we have $(q_i, w_i, \boldsymbol{v}) \xrightarrow{\gamma} (q'_i, w'_i, \boldsymbol{v}')$ is a transition of $\mathcal{P}_i$ and $q_j = q'_j$ and $w_j = w'_j$ for all $j \neq i$, or

2. $\boldsymbol{\gamma} = \varepsilon$ and $w_i = w'_i$ for all $1 \leq i \leq m$ and $(\delta, \theta, \boldsymbol{u}) \in \Delta_g$ with
   (a) $\delta(q_1, \ldots, q_m, q'_1, \ldots, q'_m)$ is true, and
   (b) $\theta(\boldsymbol{v}(1), \ldots, \boldsymbol{v}(n))$ is true, and
   (c) $\boldsymbol{v}'(i) = \boldsymbol{v}(i) + \boldsymbol{u}(i) \geq 0$ for all $1 \leq i \leq n$.

We refer to these two types of transition as *internal* and *synchronising* respectively. A run of $\mathbb{C}$ is a run $c_0 \xrightarrow{\gamma_1} c_1 \xrightarrow{\gamma_2} \cdots \xrightarrow{\gamma_m} c_m$.

Bounding Runs. During a run, the counter is in a non-decrementing mode if the last value-changing operation on that counter was an increment. Similarly, a counter may be in a non-incrementing mode. The number of reversals of a counter during a run is the number of times the counter changes from an incrementing to a decrementing mode, and vice versa. For example, if the values of a counter x in a path are $1, 1, 1, 2, 3, 4, 4, \overline{4, 3}, 2, \overline{2, 3}$, then the number of reversals of x is 2 (reversals occur in between the overlined positions). This sequence has three *phases* (i.e. subpaths interleaved by consecutive reversals or end points): non-decrementing, non-incrementing, and finally non-decrementing.

Definition 2 (r-Reversal-Bounded). *A run $c_0 \xrightarrow{\gamma_1} c_1 \xrightarrow{\gamma_2} \cdots \xrightarrow{\gamma_m} c_m$ is* r-reversal-bounded *whenever we can partition $c_0c_1 \ldots c_m$ into $C_1 \ldots C_r$ such that for all $1 \leq p \leq r$, there is some $\sim \in \{\leq, \geq\}$ such that for all c_jc_{j+1} appearing together in C_p, we have $c_j = (\ldots, \boldsymbol{v}_j)$, $c_{j+1} = (\ldots, \boldsymbol{v}_{j+1})$, and for all $1 \leq i \leq n$, $\boldsymbol{v}_j(i) \sim \boldsymbol{v}_{j+1}(i)$.*

Finally, we define the notion of *synchronisation-bounded.* We show in Section 4 that this notion subsumes context-bounded model checking.

Definition 3 (k-Synchronisation-Bounded). *A run π is* k-synchronisation-bounded *whenever π uses k or fewer synchronising transitions.*

3 Synchronisation-Bounded Reachability

The *r-reversal and k-synchronisation-bounded reachability problem* for a given $\mathbb{C}$, bound r and k asks, for given configurations c and c' of $\mathbb{C}$, is there a k-synchronisation-bounded run of $\mathbb{C}$ from c to c' using up to r reversals. We prove:

Theorem 1. *For two bounds r and k given in unary, the r-reversal and k-synchronisation-bounded reachability problem for n-SyncPDS is* NP*-complete.*

The proof extends the analogous theorem for r-reversal-bounded n-PDS [20]. We will construct, for each $\mathcal{P}_i$ in $\mathbb{C}$, an over-approximating pushdown automaton $\mathcal{P}'_i$ and use Verma *et al.* [40][3] to obtain an existential Presburger formula $\mathtt{Image}_i$ giving the Parikh image of $\mathcal{P}'_i$. Finally, we add additional constraints such that a solution exists iff the reachability problem has a positive answer.

[3] It is well known that [40] contains a small bug, fixed by Barner [6]. See the appendix for more details.

The encoding presented here is one of two encodings that we developed. This encoding is both simpler to explain and seems to be handled better by Z3 for almost all of our examples than the second encoding. However, the second encoding results in a smaller formula. Hence, we include both reductions as contributions, and present the second reduction in the appendix.

The key difference between the encodings is where we store the number of synchronisations performed so far. In the first encoding, we keep a component g in each control state; thus, from each $\mathcal{P}$ we build $\mathcal{P}'$ with $|\mathcal{Q}| \times N_{\max} \times (k+1)$ control states, where $\mathcal{Q}$ is set of control states of $\mathcal{P}$ and $N_{\max}$ is the number of mode vectors (where modes are defined below).

In the alternative encoding we put the number of synchronisations in the modes, resulting in $|\mathcal{Q}| \times (N_{\max} + k + 1)$ control states. This is important since our reduction is quadratic in the number of controls. Hence, if $k = 2$, the alternative results in pushdown automata a third of the size of the encoding presented here. However, the resulting formulas seem experimentally more difficult to solve.

Let $c = \left(q_1^0, w_1, \ldots, q_m^0, w_m, \boldsymbol{v}_0\right)$ and $c' = (f_1, w_1', \ldots, f_m, w_m', \boldsymbol{v}_f)$. By hard-coding the initial and final stack contents, we can assume that all $w_i = w_i' = \perp$.

Unfortunately, we cannot use the reduction for r-reversal-bounded n-PDS as a completely black box; hence, we will recall the relevant details and highlight the new techniques required. We refer the reader to the article [20] for further details. The correctness of the reduction is given in the appendix.

The final formula `HasRun` will take the shape

$$\exists \boldsymbol{m}_1, \ldots, \boldsymbol{m}_{N_{\max}} \exists \boldsymbol{z}_1 \ldots \boldsymbol{z}_m \left(\begin{array}{c} \texttt{Init}(\boldsymbol{m}_1) \wedge \texttt{GoodSeq}(\boldsymbol{m}_1, \ldots, \boldsymbol{m}_{N_{\max}}) \\ \wedge \bigwedge\limits_{1 \le i \le m} \texttt{Image}_i\,(\boldsymbol{z}_i) \\ \wedge\ \texttt{Respect}\left(\sum_{1 \le i \le m} \boldsymbol{z}_i, \boldsymbol{m}_1, \ldots, \boldsymbol{m}_{N_{\max}}\right) \\ \wedge\ \texttt{OneChange}\left(\sum_{1 \le i \le m} \boldsymbol{z}_i\right) \\ \wedge\ \texttt{EndVal}\left(\sum_{1 \le i \le m} \boldsymbol{z}_i\right) \wedge\ \texttt{Syncs}\left(\sum_{1 \le i \le m} \boldsymbol{z}_i\right) \end{array} \right)$$

where the formulas $\texttt{OneChange}\left(\sum_{1 \le i \le m} \boldsymbol{z}_i\right)$ and $\texttt{Syncs}\left(\sum_{1 \le i \le m} \boldsymbol{z}_i\right)$ are the main differences with the single thread case. In addition, further adaptations need to be made within other aspects of the formula. We remark at this point that the user may add to `HasRun` an additional constraint on the Parikh images of runs — such as restricting to runs where the number of characters γ output is greater than the number of γ'.

The Mode Vectors. We begin with the vectors $\boldsymbol{m}_1, \ldots, \boldsymbol{m}_{N_{\max}}$, which are unchanged from the case of r-reversal-bounded n-PDS. Let $d_1 < \ldots < d_h$ denote all the numeric constants appearing in an atomic counter constraint as a part of the constraints in the $\mathcal{P}_i$. Without loss of generality, we assume that $d_1 = 0$ for convenience. Let $\texttt{REG} = \{\varphi_1, \ldots, \varphi_h, \psi_1, \ldots, \psi_h\}$ be a set of formulas defined as follows. Note that these formulas partition $\mathbb{N}$ into $2h$ pairwise disjoint regions.

$$\varphi_i(x) \equiv x = d_i, \quad \psi_i(x) \equiv d_i < x < d_{i+1}\ (1 \le i < h), \quad \psi_h(x) \equiv d_h < x\ .$$

We call a vector in $\mathtt{REG}^n \times [0,r]^n \times \{\uparrow,\downarrow\}^n$ a *mode vector*. Given a path π from configurations c to c', we may associate a mode vector to each configuration in π. This vector records for each counter, which region its value is in, how many reversals it's used, and whether its phase is non-decrementing ($\uparrow$) or non-incrementing ($\downarrow$). Consider a sequence of mode vectors. A crucial observation is, once a change occurs to the mode information of a counter, the same information will not recur for that counter. For example, returning to the same region will incur an increase in the number of reversals. Thus, there are at most $N_{\max} := |\mathtt{REG}| \times (r+1) \times n = 2hn(r+1)$ distinct mode vectors in any sequence.

Constructing $\mathcal{P}'_i$. We define the pushdown automata

$$\mathcal{P}'_i = \left(\mathcal{Q}'_i, \Sigma_i, \Gamma', \Delta'_i, \left(q^0_i, 1, 1\right), \{f_i\} \times [1, N_{\max}] \times [1, k+1]\right)$$

for each $\mathcal{P}_i$ in $\mathbb{C}$. Note that each $\mathcal{P}'_i$ has the same output alphabet Γ'. We assume that all $\mathcal{Q}_i$ are pairwise disjoint. There are two main aspects to each $\mathcal{P}'_i$. First, we remove the counters. To replace them, we have $\mathcal{P}'_i$ output any counter changes or tests that would have been performed. E.g. where $\mathcal{P}_i$ would increment a counter, $\mathcal{P}'_i$ will output a symbol $(\mathtt{ctr}_j, 1, \ldots)$ indicating (amongst other things) that counter $\mathtt{ctr}_j$ should be increased by 1. Furthermore, $\mathcal{P}'_i$ guesses when, and keeps track of when, mode changes would have occurred. Secondly, we allow $\mathcal{P}'_i$ to non-deterministically make synchronisations (instead of communicating, the effect of external threads is guessed). In this case, the control state change, along with the number of synchronisations performed thus far, will be output. In this way, $\mathcal{P}'_i$ makes "visible" the counter tests, counter updates and synchronisations that would have been performed by $\mathcal{P}_i$ on the same run. Constraints described later in $\mathtt{HasRun}$ ensure these operations are valid.

More formally, let $\mathcal{Q}'_i = \mathcal{Q}_i \times [1, N_{\max}] \times [1, k+1]$ (that is, we add to $\mathcal{Q}_i$ the current mode and synchronisation number). We define Γ' implicitly from the transition relation. In fact, Γ' is a (finite) subset of

$$\begin{aligned}\Gamma \cup \{\ (\mathtt{ctr}_j, u, e, l) \ \mid j \in [1,n], u \in \mathbb{Z}, e \in [1, N_{\max}], l \in \{0,1\}\ \} \\ \cup\ (Const_X \times [1, N_{\max}]) \\ \cup \bigcup_{1 \le i \le m} \left(StateCons_{\mathcal{Q}_1,\ldots,\mathcal{Q}_m} \times \mathcal{Q}_i \times \mathcal{Q}_i \times [1, N_{\max}] \times [1, k+1] \times \{0,1\}\right).\end{aligned}$$

Characters $(\mathtt{ctr}_j, u, e, l)$ mean to add u to $\mathtt{ctr}_j$, in mode e, where l indicates whether the counter action changes the mode vector. Characters (θ, e) indicate a counter test in mode e. Finally, characters (δ, q, q', e, g, l) indicate a use of synchronisation rule δ, changing $\mathcal{P}_i$ from control state q to q', in mode e with g synchronisations performed so far.

We define Δ'_i to be the smallest set such that, if $(q, a, \theta) \overset{\gamma}{\hookrightarrow} (q', w, \boldsymbol{u}) \in \Delta_i$ where $\boldsymbol{u} = (u_1, \ldots, u_n)$ then for each $e \in [1, N_{\max}]$ and $g \in [1, k+1]$, Δ'_i contains

$$((q, e, g), a) \xhookrightarrow{\gamma(\theta,e)(\mathtt{ctr}_1,u_1,e,l)\ldots(\mathtt{ctr}_n,u_n,e,l)} ((q', e+l, g), w)$$

for all $l \in \{0,1\}$ if $e \in [1, N_{\max})$ and $l = 0$ otherwise. Thus, $l = 1$ signifies a mode changing transition.

These rules are the rules required in the single thread case. We need additional rules to reflect the multi-threaded environment. In particular, an external thread may change the mode, or a synchronising transition may occur. To account for this Δ'_i also has for each $q \in \mathcal{Q}_i$, $a \in \Sigma_i$, $e \in [1, N_{\max})$, and $g \in [1, k{+}1]$,

$$((q, e, g), a) \overset{\varepsilon}{\hookrightarrow} ((q, e+1, g), a) \qquad (*)$$

and, to model synchronisations, we have for all $q, q' \in \mathcal{Q}_i$, $e \in [1, N_{\max}]$, $g \in [1, k+1)$ and $(\delta, \theta, \boldsymbol{u}) \in \Delta_g$, when $i > 1$,

$$((q, e, g), a) \xhookrightarrow{(\delta,q,q',e,g,l)} ((q', e+l, g+1), a)$$

and when $i = 1$ and $\boldsymbol{u} = (u_1, \ldots, u_n)$,

$$((q, e, g), a) \xhookrightarrow{(\delta,q,q',e,g,l)(\theta,e)(\mathtt{ctr}_1,u_1,e,l)\ldots(\mathtt{ctr}_n,u_n,e,l)} ((q', e+l, g+1), a)$$

for all $l \in \{0, 1\}$ when $e \in [1, N_{\max})$ and $l = 0$ otherwise. That is, $\mathcal{P}'_i$ guesses the effect of non-internal transitions and $\mathcal{P}'_1$ is responsible for performing the required counter updates. Note that the information in the output character (δ, q, q', e, g, l) allows us to check that synchronising transitions take place in the same order and in the same modes across all threads.

Constructing The Formula. Fix an ordering $\gamma_1 < \ldots < \gamma_l$ on Γ'. By f we denote a function mapping γ_i to i for each $i \in [1, l]$. Let $\boldsymbol{z}$ denote a vector of l variables. The formula is `HasRun` given above, where `Init`, `GoodSeq`, `Respect`, and `EndVal` are defined as in the single thread case (using only variables which are unchanged from [20]); therefore, we describe them informally here, referring the reader to the appendix for the full definitions. We convert each $\mathcal{P}'_i$ to a context-free grammar (of cubic size) and use [40] to obtain $\mathtt{Image}_i$ such that for each $\boldsymbol{n} \in \mathbb{N}^l$ we have $\boldsymbol{n} \in \mathbb{P}(\mathcal{L}(\mathcal{P}'_i))$ iff $\mathtt{Image}_i(\boldsymbol{n})$ holds. Informally,

- `Init` ensures the initial mode vector $\boldsymbol{m}_1$ respects the initial configuration c;
- `GoodSeq` ensures that the sequence of mode vectors $\boldsymbol{m}_1, \ldots, \boldsymbol{m}_{N_{\max}}$ is valid. For example, if the direction of a counter changes, then an extra reversal is incurred on that counter;
- `Respect` requires that the counter tests and actions fired within a mode are allowed. For example, a subtraction may not occur on a counter in a non-decreasing phase, only one mode change action may occur per mode, and that counter tests only occur in sympathetic regions; and
- `EndVal` checks that the counter operations applied during the run leave each counter in the correct value, as given in the final configuration c'.

It remains for us to define `OneChange` and `Syncs`. We use `OneChange` to assert that only one thread may be responsible for firing the transition that changes a given mode of the counters to the next. That is,

$$\mathtt{OneChange}(\boldsymbol{z}) \equiv \bigwedge_{\substack{(\mathtt{ctr}_j,u,e,1) \\ (\mathtt{ctr}_j,u',e,1) \\ u' \neq u}} z_{f(\mathtt{ctr}_j,u,e,1)} > 0 \Rightarrow \begin{pmatrix} z_{f(\mathtt{ctr}_j,u,e,1)} = 1 \\ \wedge\ z_{f(\mathtt{ctr}_j,u',e,1)} = 0 \end{pmatrix} .$$

The role of Syncs is to ensure that the synchronising transitions taken by $\mathcal{P}'_1, \ldots, \mathcal{P}'_m$ are valid. Note that, by design, each $\mathcal{P}'_i$ will only output at most one character of the form (δ, q, q', e, g, l) for each $g \in [1, k]$. We assert, if one thread uses a global transition with condition δ, all do, and δ is satisfied. That is,

$$\texttt{Syncs}\left(\boldsymbol{z}\right) \equiv \bigwedge_{1 \le g \le k} \bigvee_{\substack{1 \le e \le N_{\max} \\ (\delta, \theta, \boldsymbol{u}) \in \Delta_g \\ l \in \{0,1\}}} \left(\begin{array}{l} \texttt{Fired}_{(\delta,e,g,l)}\left(\boldsymbol{z}\right) \Rightarrow \\ \quad \left(\texttt{Sync}_{(\delta,e,g,l)}\left(\boldsymbol{z}\right) \wedge \texttt{AllFired}_{(\delta,e,g,l)}\left(\boldsymbol{z}\right) \right) \end{array} \right)$$

where $\texttt{Sync}_{(\delta,e,g,l)}\left(\boldsymbol{z}\right)$ is $\delta\left(\boldsymbol{z}\right)$ with each atomic state constraints replaced as below.

$$(y_i = q) \equiv \bigvee_{(\delta,q,q',e,g,l)} z_{f(\delta,q,q',e,g,l)} > 0 \quad \text{and} \quad (y_i = q') \equiv \bigvee_{(\delta,q,q',e,g,l)} z_{f(\delta,q,q',e,g,l)} > 0 \ .$$

Finally, $\texttt{Fired}_{(\delta,e,g,l)}\left(\boldsymbol{z}\right) \equiv \bigvee_{(\delta,q,q',e,g,l)} z_{f(\delta,q,q',e,g,l)} > 0$, and

$$\texttt{AllFired}_{(\delta,e,g,l)}\left(\boldsymbol{z}\right) \equiv \bigwedge_{1 \le i \le m} \bigvee_{q,q' \in \mathcal{Q}_i} z_{f(\delta,q,q',e,g,l)} > 0 \ .$$

We remark upon a pleasant corollary of our main result. Consider a system of pushdown systems communicating only via reversal-bounded counters. Since such a system cannot use any synchronising transitions, all runs are 0-synchronisation-bounded; hence, their reachability problem is in NP.

4 Comparison with Context-Bounded Model Checking

Global synchronisations can be used to model classical context-bounded model checking. We present a simple encoding here. We begin with the definition.

Definition 4 (n-ClPDS). *A classical system of* communicating pushdown systems with n counters $\mathbb{C}$ *is a tuple* $(\mathcal{P}_1, \ldots, \mathcal{P}_m, G, X)$ *where, for all* $1 \le i \le m$*,* $\mathcal{P}_i$ *is a PDA with* n *counters* $(\mathcal{Q}_i, \Sigma_i, \Gamma, \Delta_i, X)$*,* X *is a finite set of counter variables and* $\mathcal{Q}_i = G \times \mathcal{Q}'_i$ *for some finite set* $\mathcal{Q}'_i$*.*

A configuration of a n-ClPDS is a tuple $(g, q_1, w_1 \ldots, q_m, w_m, \boldsymbol{v})$ where $g \in G$ and $(g, q_i) \in \mathcal{Q}_i$ for all i. We have a transition $(g, q_1, w_1, \ldots, q_m, w_m, \boldsymbol{v}) \overset{\gamma}{\Rightarrow} (g', q'_1, w'_1, \ldots, q'_m, w'_m, \boldsymbol{v}')$ when, for some $1 \le i \le m$, we have $((g, q_i), w_i, \boldsymbol{v}) \xrightarrow{\gamma} ((g', q'_i), w'_i, \boldsymbol{v}')$ is a transition of $\mathcal{P}_i$ and $q_j = q'_j$ and $w_j = w'_j$ for all $j \ne i$.

A run of $\mathbb{C}$ is a sequence $c_0 \overset{\gamma_1}{\Longrightarrow} c_1 \overset{\gamma_2}{\Longrightarrow} \cdots \overset{\gamma_m}{\Longrightarrow} c_m$. A k-context-bounded run is a run $c_0 \overset{\gamma_1}{\Longrightarrow} c_1 \overset{\gamma_2}{\Longrightarrow} \cdots \overset{\gamma_m}{\Longrightarrow} c_m$ that can be divided into k phases $C_1, \ldots, C_k$ such that during each C_i only transitions from a unique $\mathcal{P}_j$ are used. By convention, the first phase contains only transitions from $\mathcal{P}_1$.

We define an n-SyncPDS simulating any given n-ClPDS. It uses the synchronisations to pass the global component g of the n-ClPDS between configurations of the n-SyncPDS, acting like a token enabling one process to run. Since there are k global synchronisations, the run will be k-context-bounded.

Definition 5. *Given a n-ClPDS $\mathbb{C} = (\mathcal{P}_1, \ldots, \mathcal{P}_m, G, X)$. Let $\#$ be a symbol not in G. We define from each $\mathcal{P}_i = (\mathcal{Q}_i, \Sigma_i, \Gamma, \Delta_i, X)$ with $\mathcal{Q}_i = G \times \mathcal{Q}'_i$ the pushdown system $\mathcal{P}^S_i = \left(\mathcal{Q}^S_i \cup \{f_i\}, \Sigma_i, \Gamma, \Delta^S_i, X\right)$ where $\mathcal{Q}^S_i = \mathcal{Q}^i \times (G \cup \{\#\})$ and Δ^S_i is the smallest set containing Δ_i and $((g, q), a, \boldsymbol{tt}) \overset{\varepsilon}{\hookrightarrow} (f_i, w, \mathbf{0})$ for all q, a appearing as a head in the final configuration with $g = \#$ or with g also in the final configuration.*

Finally, let $\mathbb{C}^S = \left(\mathcal{P}^S_1, \ldots, \mathcal{P}^S_m, \Delta_g, X\right)$, where $\Delta_g = \{(\delta, \boldsymbol{tt}, \mathbf{0})\}$ such that the formula $\delta(q^1_1, \ldots, q^1_n, q^2_1, \ldots, q^2_n)$ holds only when there is some $g \in G$ and $1 \leq i \neq j \leq n$ such that

1. *$q^1_i = (g, q)$ and $q^2_i = (\#, q)$ for some q, and*
2. *$q^1_j = (\#, q)$ and $q^2_j = (g, q)$ for some q, and*
3. *for all $i' \neq i$ and $i' \neq j$, $q^1_{i'} = (\#, q)$ and $q^2_{i'} = (\#, q)$ for some q.*

We show in the appendix that an optimised version of this simulation — discussed in Section 5 — is correct. That is, there is a run of $\mathbb{C}$ to the final configuration $(g, q_1, w_1, \ldots, q_m, w_m, \boldsymbol{v})$ iff there is a run of $\mathbb{C}^S$ to $(f_1, w_1, \ldots, f_m, w_m, \boldsymbol{v})$ using the same number of reversals.

5 Optimisations

Our experiments suggest that without further optimisations our reduction from Section 3 is rather impractical. In this section, we provide several optimisations which considerably improve the practical aspect of our reduction. We discuss improving the encoding of the context-bounded model checking, identifying and eliminating "removable" heads from our models, and minimising the size of the CFG produced during the reduction using reachability information. The gist behind our optimisation strategies is to keep the size of the models (pushdown automata, CFG, etc.) as small as possible *throughout* our reduction, which can be achieved by removing redundant objects as early as possible. For the rest of this section, we fix an initial and a final configuration.

Context-Bounded Model Checking. The encoding context-bounded model checking encoding given in Section 4 allows context-switches to occur at any moment. However, we can observe that context-switches only need to occur when global information needs to be up-to-date. This restriction led to improvements in our experiments. We describe the positions where context-switches may occur informally here, and give a formal definition and proof in the appendix.

In our restricted encoding, context-switches may occur when an update to global component g occurs; the value of g is tested; an update to the counters occurs; the values of the counters are tested; or the control state of the active thread appears in the final configuration. Intuitively, we delay context-switches as long as possible without removing behaviours — that is, until the status of the global information affects, or may be affected by, the next transition.

Minimising Communicating Pushdown Systems with Counters. We describe a minimisation technique to reduce the size of the pushdown systems. It identifies heads q, a of the pushdown systems that are *removable*. We collapse pairs of rules passing through the head q, a into a single combined rule. Thus, we build a pushdown system with fewer heads, but the same behaviours. In the following definition, *sink states* will be defined later. Intuitively it means when q is reached, q cannot be changed in one local or global transition.

Definition 6. *A head q, a is* removable *whenever*

1. *q, a is not the head of the initial or final configuration, and*
2. *it is not a return location, i.e. there is no rule $(q_1, b_1, \theta) \overset{\gamma}{\hookrightarrow} (q_2, b_2w', \boldsymbol{u})$ with a appearing in w' and a does not appear below the top of the stack in the initial configuration, and*
3. *it is not a loop, i.e. there is no rule $(q, a, \theta) \overset{\gamma}{\hookrightarrow} (q, aw', \boldsymbol{u})$, and*
4. *it is not a synchronisation location, i.e. for all $(q_2, b, \theta) \overset{\gamma}{\hookrightarrow} (q_1, aw', \boldsymbol{u})$ or initial configuration containing q_1 and a, we have either, (i) for all $(\delta, \theta', \boldsymbol{u}') \in \Delta_g$ and $q_1^1, \ldots, q_m^1, q_1^2, \ldots, q_m^2$ such that $\delta(q_1^1, \ldots, q_m^1, q_1^2, \ldots, q_m^2)$ holds we have $q_i^j \neq q_1$ for all i, j, or (ii) q_1 is a sink state, and*
5. *it is not a counter access location, i.e. there is no rule $(q, a, \theta) \overset{\gamma}{\hookrightarrow} (q', w', \boldsymbol{u})$ such that θ depends on a counter or $\boldsymbol{u}$ contains a non-zero entry, and there is no rule $(q', b, \theta) \overset{\gamma}{\hookrightarrow} (q, aw', \boldsymbol{u})$ such that $\boldsymbol{u}$ contains a non-zero entry.*

Definition 7. *A state q is a* sink state *when for all rules $(q, a, \theta) \overset{\gamma}{\hookrightarrow} (q', w', \boldsymbol{u})$ we have $q' = q$ and for all $(\delta, \theta', \boldsymbol{u}') \in \Delta_g$ and $q_1^1, \ldots, q_m^1, q_1^2, \ldots, q_m^2$ with $q_i^1 = q$ for some i such that $\delta(q_1^1, \ldots, q_m^1, q_1^2, \ldots, q_m^2)$ holds we have $q_i^2 = q$.*

Removable heads can be eliminated by merging rules passing through them. In general, this may increase the number of rules, but in practice it leads to significant reductions (see Table 1). We show, in the appendix, that this optimisation preserves behaviours of the systems.

Definition 8. *Given a n-SyncPDS with global rules Δ_g and a pushdown system $\mathcal{P}$ with n counters $(\mathcal{Q}, \Sigma, \Gamma, \Delta, X)$ and a removable head q, a, we define $\mathcal{P}_{q,a}$ to be $(\mathcal{Q}, \Sigma, \Gamma, \Delta', X)$ where $\Delta' = \Delta$ if Δ_1 is empty and $\Delta' = \Delta_1 \cup \Delta_2$ otherwise, where*

$$\Delta_1 = \left\{ (q_1, b, \theta) \overset{\gamma_1, \gamma_2}{\longhookrightarrow} (q_2, w, \boldsymbol{u}_1 + \boldsymbol{u}_2) \;\middle|\; \begin{array}{l} (q_1, b, \theta_1) \overset{\gamma_1}{\hookrightarrow} (q, aw_1, \boldsymbol{u}_1) \in \Delta \;\wedge \\ (q, a, \theta_2) \overset{\gamma_2}{\hookrightarrow} (q_2, w_2, \boldsymbol{u}_2) \in \Delta \;\wedge \\ \theta = (\theta_1 \wedge \theta_2) \wedge w = w_2 w_1 \end{array} \right\}$$

$$\text{and } \Delta_2 = \Delta \setminus \left(\begin{array}{l} \left\{ (q_1, b, \theta) \overset{\gamma}{\hookrightarrow} (q_2, w', \boldsymbol{u}) \;\mid\; q_1, b = q, a \right\} \cup \\ \left\{ (q_1, b_1, \theta) \overset{\gamma}{\hookrightarrow} (q_2, b_2 w', \boldsymbol{u}) \;\mid\; q_2, b_2 = q, a \right\} \end{array} \right).$$

Minimising CFG size via pushdown reachability table. Recall that our reduction to existential Presburger formulas from an n-SyncPDS makes use of a standard language-preserving reduction from pushdown automata to context-free grammars (CFGs). Unfortunately, the standard translations from PDAs to CFGs incur a cubic blow-up. More precisely, if the input PDA is $\mathcal{P} = \big(\mathcal{Q}, \Sigma, \Gamma, \Delta, q^0, F\big)$, the output CFG has size $O(|\Delta| \times |\mathcal{Q}|^2)$. Our experiments suggest that this cubic blow-up is impractical without further optimisation, i.e., the naive translation failed to terminate within a couple of hours for most of our examples. Note that the complexity of translating from PDA to CFG is very much related to the reachability problem for pushdown systems, for which the optimal complexity is a long-standing open problem (the fastest algorithm [11] to date has complexity $O(n^3/\log n)$ under certain assumptions).

We will now describe two optimisations to improve the size of the CFG that is produced by our reduction in the previous section, the second optimisation gives better performance (asymptotically and empirically) than the first. Without loss of generality, we assume that: (A1) the PDA empties the stack as it accepts an input word, (A2) the transitions of the input PDA are of the form $(p, a) \overset{\gamma}{\hookrightarrow} (q, w)$ where $p, q \in \mathcal{Q}$, $\gamma \in \Gamma^*$, $a \in \Sigma$, and $w \in \Sigma^*$ with $|w| \leq 2$. [It is well-known that any input PDA can be translated into a PDA in this "normal form" that recognises the same language while incuring only a linear blow-up.] The gist behind both optimisations is to refrain from producing redundant CFG rules by looking at the reachability table for the PDA. Keeping the CFG size low in the first place results in algorithms that are more efficient than removing redundant rules *after* the CFG is produced.

Let us first briefly recall a standard language-preserving translation from PDA to CFG. Given a PDA $\mathcal{P} = \big(\mathcal{Q}, \Sigma, \Gamma, \Delta, q^0, F\big)$, we construct the following CFG with nonterminals $N = \{S\} \cup \{A_{p,a,q} : p, q \in \mathcal{Q}, a \in \Sigma\}$, terminals Γ, starting nonterminal S, and the following transitions:

(1) For each $(p, a) \overset{\gamma}{\hookrightarrow} (q, \epsilon) \in \Delta$, the CFG has $A_{p,a,q} \to \gamma$.
(2) For each $(p, a) \overset{\gamma}{\hookrightarrow} (p', b) \in \Delta$ and $q \in \mathcal{Q}$, the CFG has $A_{p,a,q} \to \gamma A_{p',b,q}$.
(3) For each $(p, a) \overset{\gamma}{\hookrightarrow} (p', cb) \in \Delta$ and $r, q \in \mathcal{Q}$, the CFG has $A_{p,a,q} \to \gamma A_{p',c,r} A_{r,b,q}$.
(4) Add $S \to A_{q^0,\perp,q_F}$ for each $q_F \in F$.

Note that $A_{p,a,q}$ generates all words that can be output by $\mathcal{P}$ from configuration (p, a) ending in configuration (q, ϵ). Both of our optimisations refrain from generating: (i) CFG rules of type (2) above in the case when $(p', b) \not\to^* (q, \epsilon)$, and (ii) CFG rules of type (3) in the case when $(p', c) \not\to^* (r, \epsilon)$ or $(r, b) \not\to^* (q, \epsilon)$, and (iii) CFG rules of type (4) in the case when $\big(q^0, \perp\big) \not\to^* (q_F, \epsilon)$.

It remains to describe how to build the reachability lookup table for $\mathcal{P}$ with entries of the form (p, a, q) witnessing whether $(p, a) \to^* (q, \epsilon)$. The first optimisation achieves this by directly applying the *pre** algorithm for pushdown systems described in [16], which takes $O(|\mathcal{Q}|^2 \times |\Delta|)$ time. This optimisation holds for any input PDA and, hence, does not exploit the structure of the PDA that we generated in the previous section. Our second optimisation improves the *pre** algorithm for pushdown systems from [16] by exploiting the structure of the PDA generated in the previous section, for which each control state is of the form (p, i, j), where $i, j \in \mathbb{Z}_{>0}$. The crucial observation is that, due to the

PDA rules of type (*) generated from the previous section, the PDA that we are concerned with satisfy the following two properties:

(P0) $((p,i,j),v) \rightarrow^* ((q,i',j'),w)$ implies $i' \geq i$ and $j' \geq j$.
(P1) $((p,i,j),v) \rightarrow^* ((q,i',j'),w))$ implies for each $d_1, d_2 \in \mathbb{N}$ that we have $((p,i+d_1,j+d_2),v) \rightarrow^* ((q,i'+d_1,j'+d_2),w)$.
(P2) for each nonempty $v \in \Sigma^*$: $((p,i,j),av) \rightarrow^* ((q,i',j'),v)$ implies we have $((p,i,j),av) \rightarrow^* ((q,i'',j'),v)$ for each $i'' \geq i'$.

Properties **(P0)** and **(P1)** imply it suffices to keep track of the *differences* in the mode indices and context indices in the reachability lookup table, i.e., instead of keeping track of all values $((p,i,j),a) \rightarrow^* ((q,i',j'),\epsilon)$, each entry is of the form (p,a,q,d,d') meaning that $((p,i,j),a) \rightarrow^* ((q,i+d,j+d'),\epsilon)$ for each $i,j \in \mathbb{Z}_{>0}$. Property **(P2)** implies that if (p,a,q,d,d') is an entry in the table, then so is $(p,a,q,d+i,d')$ for each $i \in \mathbb{N}$. Therefore, whenever p,a,q,d' are fixed, it suffices to *only* keep track of the minimum value d such that (p,a,q,d,d') is an entry in the table. We describe the adaptation of the *pre** algorithm for pushdown systems from [16] for computing the specialised reachability lookup table in the appendix. The resulting time complexity for computing the specialised reachability lookup table becomes linear in the number of mode indices.

6 Implementation and Experimental Results

We implemented two tools: Pushdown Translator and SynPCo2Z3.

Pushdown Translator The Pushdown Translator tool, implemented in C++, takes a program in a simple input language and produces an n-SyncPDS. The language supports threads, boolean variables (shared between threads, global to a thread, or local), shared counters, method calls, assignment to boolean variables, counter increment and decrement, branching and assertions with counter and boolean variable tests, non-deterministic branching, goto statements, locks, output, and while loops. The user can specify the number of reversals and context-switches and specify a constraint on the output performed (e.g. find runs where the number of γ characters output equals the number of γ's). The full syntax is given in the appendix. The translation uses the context-switch technique presented in Definition 5 and the minimisation technique of Definition 8. Furthermore, the constructed pushdown systems only contain transitions for reachable states of each thread (assuming counter tests always pass, and synchronising transition can always be fired).

SynPCo2Z3 Our second tool SynPCo2Z3 is implemented in SWI-Prolog. The input is an n-SyncPDS, reversal bound r, and synchronisation-bound k. Due to the declarative nature of Prolog, the syntax is kept close to the n-SyncPDS definition. The output is an existential Presburger formulas in SMT-LIB format, supported by Z3. Moreover, the tool implements all the different translations that have been described in this paper (including appendix) with and without the optimisations described in the previous section. The user may also specify a constraint on the output performed by the input n-SyncPDS.

Experiments We tested our implementation on several realistic benchmarks. One benchmark concerns the producer-consumer examples (with one producer and one consumer) from [31]. We took two examples from [31]: one uses one counter and is erroneous, wherein both producer and consumer might be both asleep (a deadlock), and the other uses two counters and is correct. The n-SyncPDS models of these examples were hand-coded since they use synchronisations rather than the context-switches of Pushdown Translator.

The remaining benchmarks were adapted from modules found in Linux kernel 3.2.1, which contained list- and memory-management, as well as locks for concurrent access. These modules often provided "register" and "unregister" functions in their API. We tested that, when register was called as many times as unregister, the number of calls to `malloc` was equal to the number of calls to `free`. Furthermore, we checked that the module did not attempt to remove an item from an empty list. In all cases, memory and list management was correct. We then introduced bugs by either removing a call to `free`, or a lock statement. Note the translation from C to our input language was by hand, and an automatic translation is an interesting avenue of future work.

The results are shown in Table 1. All tests were run on a 2.8GHz Intel machine with 32GB of RAM. Each benchmark had two threads, two context-switches, one counter and one reversal. The size fields give the total number of pushdown rules in the n-SyncPDS, both before and after removable heads minimisation. Tran. Time gives the time it took to produce the SMT formula, Solve Time is the time taken by Z3 (v. 3.2, Linux build). Each cell contains two entries: the first is for the instance with a bug, the second for the correct instance.

File	Size	Min. Size	Tran. Time	Solve Time
`prod-cons.c`	22/13	-/-	0.8s/1.8s	4.2s/6.8s
`api.c` (rtl8192u)	654/660	202/208	28s/28s	4m19s/4m32s
`af_alg.c`	506/528	174/204	18s/21s	10m2s/4m47s
`hid-quirks.c`	557/559	303/303	47s/47s	18m41s/12m5s
`dm-target.c`	416/436	254/278	27s/29s	36m43s/10m1s

Table 1. Results of experimental runs.

7 Conclusions and Future Work

We have studied the synchronisation-bounded reachability problem for a class of pushdown systems communicating by shared reversal-bounded counters and global synchronisations. This problem was shown to be NP-complete via an efficient reduction to existential Presburger arithmetic, which can be analysed using fast SMT solvers such as Z3. We have provided optimisation techniques for the models and algorithms and a prototypical implementation of this reduction and experimented on a number of realistic examples, obtaining positive results.

There are several open problems. For instance, one weakness we would like to address is that we cannot represent data symbolically (using BDDs, for example). This prevents us from being competitive with tools such as Getafix [39] for context-bounded model-checking of pushdown systems without counters.

Furthermore, although we can obtain from the SMT solver a satisfying assignment to the Presburger formula, we would like to be able to construct a complete trace witnessing reachability. Additionally, the construction of a counter-example guided abstraction-refinement loop will require the development of new techniques not previously considered. In particular, heuristics will be needed to decide when to introduce new counters to the abstraction.

We may also consider generalisations of context-bounded analysis such as phase-bounds and ordered multi-stack automata. A further challenge will be to adapt our techniques to dynamic thread creation, where *each thread* has its own context-bound, rather than the system as a whole.

Acknowledgments. We thank EPSRC (EP/F036361 & EP/H026878/1), AMIS (ANR 2010 JCJC 0203 01 AMIS) and VAPF (Fondation de Coopération Scientifique du Campus Paris Saclay) for their support.

References

1. M. F. Atig, B. Bollig, and P. Habermehl. Emptiness of multi-pushdown automata is 2etime-complete. In *Developments in Language Theory*, pages 121–133, 2008.
2. M. F. Atig, A. Bouajjani, and S. Qadeer. Context-bounded analysis for concurrent programs with dynamic creation of threads. In *LMCS*, 7(4), 2011.
3. T. Ball, B. Cook, V. Levin, and S. K. Rajamani. Slam and static driver verifier: Technology transfer of formal methods inside microsoft. In *IFM*, pages 1–20, 2004.
4. T. Ball, V. Levin, and S. K. Rajamani. A decade of software model checking with slam. *Commun. ACM*, 54:68–76, July 2011.
5. T. Ball and S. K. Rajamani. Bebop: A symbolic model checker for boolean programs. In *SPIN*, pages 113–130, 2000.
6. S. Barner. H3 mit gleichheitstheorien. Diploma thesis, TUM, 2006.
7. A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model-checking. In *CONCUR*, pages 135–150, 1997.
8. A. Bouajjani, J. Esparza, S. Schwoon, and J. Strejcek. Reachability analysis of multithreaded software with asynchronous communication. In *FSTTCS*, pages 348–359, 2005.
9. A. Bouajjani, J. Esparza, and T. Touili. A generic approach to the static analysis of concurrent programs with procedures. *SIGPLAN Not.*, 38(1):62–73, 2003.
10. A. Bouajjani, M. Müller-Olm, and T. Touili. Regular symbolic analysis of dynamic networks of pushdown systems. In *CONCUR*, pages 473–487, 2005.
11. Swarat Chaudhuri. Subcubic algorithms for recursive state machines. In *POPL*, pages 159–169, 2008.
12. Z. Dang, O. H. Ibarra, T. Bultan, R. A. Kemmerer, and J. Su. Binary reachability analysis of discrete pushdown timed automata. In *CAV*, pages 69–84, 2000.
13. L. Mendonça de Moura and N. Bjørner. Z3: An efficient smt solver. In *TACAS*, pages 337–340, 2008.
14. V. D'Silva, D. Kroening, and G. Weissenbacher. A survey of automated techniques for formal software verification. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 27(7):1165–1178, 2008.
15. J. Esparza and P. Ganty. Complexity of pattern-based verification for multi-threaded programs. In *POPL*, pages 499–510, 2011.

16. J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. Efficient algorithms for model checking pushdown systems. In *CAV*, pages 232–247, 2000.
17. J. Esparza, A. Kucera, and S. Schwoon. Model checking ltl with regular valuations for pushdown systems. *Inf. Comput.*, 186(2):355–376, 2003.
18. P. Ganty, R. Majumdar, and B. Monmege. Bounded underapproximations. In *CAV*, pages 600–614, 2010.
19. E. M. Gurari and O. H. Ibarra. The complexity of decision problems for finite-turn multicounter machines. *J. Comput. Syst. Sci.*, 22(2):220–229, 1981.
20. M. Hague and A. W. Lin. Model checking recursive programs with numeric data types. In *CAV*, pages 743–759, 2011.
21. A. Heußner, J. Leroux, A. Muscholl, and G. Sutre. Reachability analysis of communicating pushdown systems. In *FOSSACS'10*, pages 267–281, 2010.
22. R. R. Howell and L. E. Rosier. An analysis of the nonemptiness problem for classes of reversal-bounded multicounter machines. *J. Comput. Syst. Sci.*, 34(1):55–74, 1987.
23. O. H. Ibarra. Reversal-bounded multicounter machines and their decision problems. *J. ACM*, 25(1):116–133, 1978.
24. O. H. Ibarra, J. Su, Z. Dang, T. Bultan, and R. A. Kemmerer. Counter machines and verification problems. *Theor. Comput. Sci.*, 289(1):165–189, 2002.
25. V. Kahlon. Parameterization as abstraction: A tractable approach to the dataflow analysis of concurrent programs. In *LICS*, pages 181–192, 2008.
26. A. Lal, T. Touili, N. Kidd, and T. Reps. Interprocedural analysis of concurrent programs under a context bound. In *TACAS*, pages 282–298, 2008.
27. F. Laroussinie, A. Meyer, and E. Petonnet. Counting CTL. In *FOSSACS*, pages 206–220, 2010.
28. P. Madhusudan and G. Parlato. The tree width of auxiliary storage. In *POPL*, pages 283–294, 2011.
29. R. Mayr. *Decidability and Complexity of Model Checking Problems for Infinite-State Systems.* PhD thesis, TU-München, 1998.
30. M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *PLDI*, pages 446–455, 2007.
31. Wikipedia. `http://en.wikipedia.org/wiki/Producer-consumer_problem`.
32. S. Qadeer. The case for context-bounded verification of concurrent programs. In *SPIN*, pages 3–6, 2008.
33. G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. In *TOPLAS*, 2000.
34. S. Qadeer and J. Rehof. Context-bounded model checking of concurrent software. In *TACAS*, pages 93–107, 2005.
35. K. Sen and M. Viswanathan. Model checking multithreaded programs with asynchronous atomic methods. In *CAV*, pages 300–314, 2006.
36. D. Suwimonteerabuth, J. Esparza, and S. Schwoon. Symbolic context-bounded analysis of multithreaded java programs. In *SPIN*, pages 270–287, 2008.
37. D. Suwimonteerabuth, S. Schwoon, and J. Esparza. jmoped: A java bytecode checker based on moped. In *TACAS*, pages 541–545, 2005.
38. S. La Torre, P. Madhusudan, and G. Parlato. A robust class of context-sensitive languages. In *In LICS*, pages 161–170. IEEE Computer Society, 2007.
39. S. La Torre, P. Madhusudan, and G. Parlato. Analyzing recursive programs using a fixed-point calculus. In *PLDI*, pages 211–222, 2009.
40. K. N. Verma, H. Seidl, and T. Schwentick. On the complexity of equational horn clauses. In *CADE*, pages 337–352, 2005.

A Fix for Verma *et al.* [40]

It is well known that the reduction of encoding of the Parikh image of a given CFG as an existential Presburger formula given by Verma *et al.* [40] contains a small mistake, resulting in a formula that is under-constrained. This bug has been fixed by Barner in a 2006 Master's thesis [6]. We describe briefly here the bug and the fix that we implemented in our work (which is logically equivalent to Barner's).

The formula uses a number of variables which correspond to a derivation tree of the CFG and are summarised below.

- Variables x_A for each terminal A denote the number of occurrences of the terminal A.
- Variables y_p for each production rule p denotes the number of times the production rule p is used in the derivation.

There are two parts to the formula. The first part expresses the fact that the number of production rules fired and terminals generated are consistent. E.g., if a rule $p = B \to A$ is fired i times, and is the only rule generating the terminal A, then x_A must equal y_p. Similarly, the number of times p is used cannot exceed the number of B non-terminals generated by other production rules (and the initial non-terminal).

The second part of the formula contains the mistake. It's role is to ensure that production rules don't appear "out of thin air". E.g., a rule $p = B \to AB$ (where A is a terminal and B a non-terminal) could be self-supporting under the conditions in the first part of the formula. E.g., if p is the only rule using and generating B, we can assign any value to y_p since the first part of the formula will only assert that the number of Bs generated is equal to the number consumed. That is, $y_p = y_p$.

In order to prevent such situations, Verma *et al.* show that it is enough to be able to define a kind of spanning tree, rooted at the initial non-terminal with an edge between two nodes α and β if a rule p has been used that has α on the right-hand size and β on the left. Thus, all terminals and non-terminals appearing in the derivation tree must be rooted to the initial non-terminal in this spanning tree.

To this end, the formula introduces variables z_A and z_B for each terminal and non-terminal, representing the distance in the spanning tree from the root (and take the value 0 if the (non-)terminal does not appear). We conjunct

- $x_A = 0 \vee z_A > 0$ for each terminal A, and
- For each terminal or non-terminal A where $p_1, \ldots, p_l$ are the productions with A on the right-hand side and $B_1, \ldots, B_l$ are their corresponding left-hand sides, then we have $(z_A = 0) \vee \bigvee_{i=1}^{l}(z_A = z_{B_i} + 1 \wedge y_{p_i} > 0 \wedge z_{B_i} > 0)$, unless B_i is the initial non-terminal, then the corresponding disjunct is $z_A = 1 \wedge y_{p_i} > 0$.

However, this formula is under-constrained: in the final conjunct we may have $z_A = 0$ or $\bigvee_{i=1}^{l}(\ldots)$. The first disjunct catches the case where a non-terminal A

is not generated, but does not enforce that it is in fact the case that A is not generated.

Consider, for example, the CFG defined by $p_1 = S \to A$ and $p_2 = B \to AB$, for initial non-terminal S, non-terminal B and terminal A. The only derivation is $S \to A$. However, by assigning $z_S = 1$, $z_A = 2$, and $z_B = 0$ we are able to satisfy the formula with any assignment to x_A and y_{p_2} such that $x_A + 1 = y_{p_2}$.

We correct the second part of the formula by replacing $(z_A = 0)$ with $(z_A = 0 \wedge \bigwedge_{i=1}^{l} y_{p_i} = 0)$, which enforces that, when $z_A = 0$, the non-terminal A is no rule generating A has been used. The full, corrected, second part of the formula is given below.

- $x_A = 0 \vee z_A > 0$ for each terminal A, and
- For each terminal or non-terminal A where $p_1, \ldots, p_l$ are the productions with A on the right-hand side and $B_1, \ldots, B_l$ are their corresponding left-hand sides, then we have $(z_A = 0 \wedge \bigwedge_{i=1}^{l} y_{p_i} = 0) \vee \bigvee_{i=1}^{l} (z_A = z_{B_i} + 1 \wedge y_{p_i} > 0 \wedge z_{B_i} > 0)$, unless B_i is the initial non-terminal, then the corresponding disjunct is $z_A = 1 \wedge y_{p_i} > 0$.

B Correctness of the Reachability Reduction

We prove the following lemmas for the reduction presented in Section 3, which implies Theorem 1. We begin by recalling the definitions missing from `HasRun`, since these are required for the proofs.

First, since a mode vector is a member of $\mathtt{REG}^n \times [0, r]^n \times \{\uparrow, \downarrow\}^n \times [0, k]$, we set $\boldsymbol{m}_e$ to be a tuple of variables $\{\ reg_j^e, rev_j^e, arr_j^e \mid j \in [1, n]\ \}$, where:

- reg_j^e is a variable that will range over $[1, 2h]$ denoting which region the jth counter is in (a number of the form $2i + 1$ refers to φ_i, while $2i$ refers to ψ_i).
- rev_j^e is a variable that will range over $[0, r]$ denoting the number of reversals that have been used thus far by the jth counter.
- arr_j^e is a variable that will range over $\{0, 1\}$ denoting the current arrow direction, e.g., 0/1 for $\uparrow / \downarrow$ (non-decrementing/non-incrementing mode).

We begin with $\mathtt{Init}(\boldsymbol{m}_1)$, which asserts that the initial mode vector respects the initial configuration $(q, u, \boldsymbol{v})$, where $\boldsymbol{v} = (c_1, \ldots, c_n)$. Recalling that $\mathtt{REG} = \{\varphi_1, \ldots, \varphi_h, \psi_1, \ldots, \psi_h\}$, we set

$$\mathtt{Init}(\boldsymbol{m}_1) \equiv \bigwedge_{j=1}^{n} \left(rev_j^1 = 0 \wedge \bigwedge_{i=1}^{h} \begin{pmatrix} reg_j^1 = 2i - 1 \iff \varphi_i(c_j) \wedge \\ reg_j^1 = 2i \iff \psi_i(c_j) \end{pmatrix} \right) .$$

Recall that the target configuration is $(q', u', \boldsymbol{v}')$, where $\boldsymbol{v}' = (c'_1, \ldots, c'_n)$. We assert that the end counter values match $\boldsymbol{v}'$. This definition is given as follows.

$$\mathtt{EndVal}(\boldsymbol{z}) \equiv \bigwedge_{j=1}^{n} \left(\sum_{(\mathtt{ctr}_j, u, e, l)} u \times z_{f(\mathtt{ctr}_j, u, e, l)} \right) = c'_j .$$

Next, `GoodSeq` ensures that the sequence of mode vectors $\boldsymbol{m}_1, \ldots, \boldsymbol{m}_{N_{\max}}$ is valid. For example, if the direction of a counter changes, then an extra reversal is incurred. The formula is a conjunction of smaller formulas defined below. One conjunct says that each rev_j^e must be a number in $[0, r]$. Likewise, we add conjuncts expressing that each reg_j^e (resp. arr_j^e) ranges over $[1, 2h]$ (resp. $\{0, 1\}$). We also need to state that changes in the direction arrows incur an extra reversal (otherwise, no reversal is incurred). Hence, we add

$$\bigwedge_{j=1}^{k} \bigwedge_{e=1}^{N_{\max}} 0 \leq rev_j^e \leq r \wedge \bigwedge_{j=1}^{k} \bigwedge_{e=1}^{N_{\max}-1} \begin{pmatrix} arr_j^e \neq arr_j^{e+1} \Rightarrow rev_j^{e+1} = rev_j^e + 1 \wedge \\ arr_j^e = arr_j^{e+1} \Rightarrow rev_j^{e+1} = rev_j^e \end{pmatrix} .$$

Next, the sequence $\{reg_j^e\}_{e=1}^{N_{\max}}$ must obey the changes in $\{arr_j^e\}_{e=1}^{N_{\max}}$. Hence, we have

$$\bigwedge_{j=1}^{k} \bigwedge_{e=1}^{N_{\max}-1} \left(reg_j^e < reg_j^{e+1} \Rightarrow arr_j^{e+1} = 0 \wedge reg_j^e > reg_j^{e+1} \Rightarrow arr_j^{e+1} = 1 \right) .$$

I.e., since regions denote increasing intervals, going to higher (resp. lower) regions mean the counter mode must be non-decrementing (resp. non-incrementing). This completes the definition of `GoodSeq`.

Lastly, we give the formula `Respect` which requires that the counter tests and actions fired within a mode are allowed. For example, a subtraction may not occur on a counter in a non-decreasing phase, and counter tests may only occur in sympathetic regions. Again, this is a conjunction. First, when the jth counter is non-incrementing (resp. non-decrementing), we allow only non-negative (resp. non-positive) counter increments:

$$\bigwedge_{j=1}^{n} \bigwedge_{e=1}^{N_{\max}} \begin{pmatrix} arr_j^e = 0 \Rightarrow \bigwedge_{\substack{(\mathtt{ctr}_j, u, e, l) \\ u < 0}} z_{f(\mathtt{ctr}_j, u, e, l)} = 0 \wedge \\ arr_j^e = 1 \Rightarrow \bigwedge_{\substack{(\mathtt{ctr}_j, u, e, l) \\ u > 0}} z_{f(\mathtt{ctr}_j, u, e, l)} = 0 \end{pmatrix} .$$

Secondly, the value of the jth counter at the beginning and end of each mode must respect the guessed mode vector. Let us first introduce some notation. Observe that the value of the jth counter at the *end* of eth mode vector can be expressed by $c_j + \left(\sum_{e'=1}^{e-1} \sum_{(\mathtt{ctr}_j, u, e', l) \in \Gamma'} u \times z_{f(\mathtt{ctr}_j, u, e', l)} \right) + \sum_{(\mathtt{ctr}_j, u, e, 0)} u \times z_{f(\mathtt{ctr}_j, u, e, 0)}$. Let us denote this term by $\mathit{EndCounter}_j^e$. Similarly, the value at the *beginning* of the eth mode is $c_j + \sum_{e'=1}^{e-1} \sum_{(\mathtt{ctr}_j, u, e', l) \in \Gamma'} u \times z_{f(\mathtt{ctr}_j, u, e', l)}$. We denote this term by $\mathit{StartCounter}_j^e$. Hence, this second conjunct is

$$\bigwedge_{j=1}^{k} \bigwedge_{e=1}^{N_{\max}} \bigwedge_{l=1}^{h} \begin{pmatrix} reg_j^e = 2l - 1 \Rightarrow \left(\varphi_l(\mathit{EndCounter}_j^e) \wedge \varphi_l(\mathit{StartCounter}_j^e) \right) \wedge \\ reg_j^e = 2l \Rightarrow \left(\psi_l(\mathit{EndCounter}_j^e) \wedge \psi_l(\mathit{StartCounter}_j^e) \right) \end{pmatrix} .$$

Finally, we need to express that no invalid counter tests are executed in a given mode. To test whether a constraint θ is satisfied by the values $\boldsymbol{v} = (c_1, \dots, c_n)$ of the counters, it is necessary and sufficient to test whether θ is satisfied by *some* vector $\boldsymbol{v}' = (c'_1, \dots, c'_n)$, where each c_i lies in the same region as c'_i. Therefore, the desired property can be expressed as:

$$\bigwedge_{e=1}^{N_{\max}} \bigwedge_{(\theta,e)\in\Gamma'} z_{f(\theta,e)} > 0 \Rightarrow \theta\left(\mathit{StartCounter}_1^e, \dots, \mathit{StartCounter}_n^e\right) \enspace .$$

We are now ready to prove the correctness of our reduction.

Lemma 1. *If there is an r-reversal and k-synchronisation-bounded run from c to c', then* `HasRun` *is satisfiable.*

Proof. Assume there is an r-reversal and k-synchronisation-bounded run of $\mathbb{C}$. We show that `HasRun` is satisfiable. Let $\pi = c_0 \xRightarrow{\gamma_1} c_1 \xRightarrow{\gamma_2} \cdots \xRightarrow{\gamma_h} c_h$ be the run of $\mathbb{C}$, where $c = c_0$ and $c' = c_h$.

First, we assign to $\boldsymbol{m}_1, \dots, \boldsymbol{m}_{N_{\max}}$ the values corresponding to the evolution of the modes in π.

Next, we assign $\boldsymbol{z}_1, \dots, \boldsymbol{z}_m$. We can divide $c_0 c_1 \dots c_h$ into $k + 1$ phases $C_1, \dots, C_{k+1}$ such that there exist $-1 = j_0 \leq \cdots \leq j_{k+1} = h$ and $C_1 = c_0 \dots c_{j_1}$, $\dots$, $C_k = c_{j_{(k-1)}+1} \dots c_{j_{k+1}}$. In addition, for all $i \in [1, k+1]$ and all $j_{(i-1)} < j < j_i$, we have $c_j \xRightarrow{\gamma_{j+1}} c_{j+1}$ is an internal transition, and, when $i \leq k$, $c_{j_i} \xRightarrow{\gamma_{j_{i+1}}} c_{j_{i+1}}$ is a synchronising transition. Note, that we allow the phases to be empty (when the start index is greater than the end index).

We will construct accepting runs of each $\mathcal{P}'_i$. We may split each phase further by considering the evolution of the modes.

For each $i \in [1, m]$, and each $j \in [1, k + 1]$ extract from C_j all transitions that are due to transitions of $\mathcal{P}_i$. Furthermore, augment all control states with the mode and the phase the transition took place in. This almost results in a run of $\mathcal{P}'_i$, except when an external process changes the mode. In this case we use the rules

$$((q, e, g), a) \overset{\varepsilon}{\hookrightarrow} ((q, e + 1, g), a)$$

to bridge the gap. Finally, we construct an accepting run of $\mathcal{P}'_i$ by concatenating the runs for each phase as follows.

For $i > 1$, we use the rules

$$((q, e, g), a) \xhookrightarrow{(\delta,q,q',e,g,l)} ((q', e + l, g + 1), a)$$

to form the connections, where $(\delta, \theta, \boldsymbol{u})$ is the synchronising transition used to bridge the phases, and $l \in \{0, 1\}$ is 1 only if the counter updates caused a mode change. For $i = 1$, we use

$$((q, e, g), a) \xhookrightarrow{(\delta,q,q',e,g,l)(\theta,e)(\mathtt{ctr}_1,u_1,e,l)\dots(\mathtt{ctr}_n,u_n,e,l)} ((q', e, g + 1), a) \enspace .$$

We assign to $\boldsymbol{z}_i$ the Parikh image of the run defined above. Because we assigned a valid sequence of modes to the mode vectors, and valid Parikh images of $\mathcal{P}'_i$ to $\boldsymbol{z}_i$ that respect the modes, we have that $\texttt{Init}, \texttt{GoodSeq}$, and $\texttt{Image}_i$ are satisfied. To see that $\texttt{Respect}$ and $\texttt{EndVal}$ are satisfied, observe that the counter actions come from a valid accepting run. Finally, $\texttt{OneChange}$ is true since only one thread changed the mode, and $\texttt{Syncs}$ is satisfied since we assigned valid synchronising transitions to the output tuples (δ, q, q', e, g, l).

Lemma 2. *If the formula* $\texttt{HasRun}$ *is satisfiable, then there is an r-reversal and k-synchronisation-bounded run from c to c'.*

Proof. Take an accepting assignment to $\boldsymbol{m}_1, \ldots, \boldsymbol{m}_{N_{\max}}$ and $\boldsymbol{z}_1, \ldots, \boldsymbol{z}_m$. Because $\texttt{Init}$, $\texttt{GoodSeq}$, $\texttt{Image}_i$, $\texttt{Respect}$ and $\texttt{EndVal}$ are satisfied, we obtain the following. First, since $\texttt{Init}\,(\boldsymbol{m}_1)$ is satisfied, we know that the assignment to the first mode vector is the initial vector. From $\texttt{GoodSeq}\,(\boldsymbol{m}_1, \ldots, \boldsymbol{m}_{N_{\max}})$, we know that the assignments to the sequence of mode vectors represents a valid progression of modes. Then, $\texttt{Respect}\left(\sum_{1 \leq i \leq m} \boldsymbol{z}_i, \boldsymbol{m}_1, \ldots, \boldsymbol{m}_{N_{\max}}\right)$ ensures that each $\mathcal{P}'_i$ only uses counter tests that are congruent with the mode e that $\mathcal{P}'_i$ is operating in, that the counter updates $(\texttt{ctr}_e, u, j, l)$ performed by the threads $\mathcal{P}'_i$ are also congruent with the current mode e, and finally that the sum of all counter actions performed by all threads results in counter values that respect their modes (since all threads output the same characters for their counter actions $\texttt{Respect}$ remains correct whether one thread is performing counter actions, or several). Similarly, $\texttt{EndVal}\left(\sum_{1 \leq i \leq m} \boldsymbol{z}_i\right)$ ensures that the total of the counter actions performed throughout the collected runs of the $\mathcal{P}'_i$ result in counter values matching the final configuration.

Given this, We will construct a valid r-reversal and k-synchronisation-bounded run of $\mathbb{C}$.

Let $\pi_i = c^i_0 \xrightarrow{\gamma^i_1} c^i_1 \xrightarrow{\gamma^i_2} \cdots \xrightarrow{\gamma^i_{h_i}} c^i_{h_i}$ be the accepting runs of the $\mathcal{P}'_i$. Since $\texttt{Syncs}$ is satisfied, we know that each π_i uses the same number of guessed synchronising transitions, and each phase change happens in the same mode of each $\mathcal{P}'_i$. For convenience, assume all k phases are used. For each $g \in [1, k+1]$, let j_g be the mode the gth phase starts in (this is the same for all i) and j'_g be the mode the gth phase ends in. Thus we can build $C^i_g = C^i_{(j_g, g)} C^i_{(j_g+1, g)} \cdots C^i_{(j'_g, g)}$ for all $g \in [1, k+1]$ where each $C^i_{(j,g)}$ is the sub-run of $\mathcal{P}'_i$ in mode j and phase g.

For each phase $g \in [1, k+1]$ and mode $j \in [1, N_{\max}]$ we build the following run π^g_j of $\mathbb{C}$. These will be used to build π^g for each phase. Observe that, since $\texttt{OneChange}$ is true, only one mode change rule can be fired amongst all threads. Hence, all but one $\mathcal{P}'_i$ must change modes using the rules

$$((q, e, g), a) \overset{\varepsilon}{\hookrightarrow} ((q, e+1, g), a)$$

in the case when it is not a synchronising transition that causes the mode change, and

$$((q, e, g), a) \xhookrightarrow{(\delta, q, q', e, g, 1)} ((q', e+1, g+1), a)$$

in the other case (note that this rule will not be used in a particular $C^i_{(j,g)}$, but rather as the synchronising transition discussed below).

In the first — non-synchronising — case, these rules bridge each $C^i_{(j,g)}$ and $C^i_{(j+1,g)}$ and are not included in π^g_j or π^g. Let r^g_j be the only mode changing transition amongst all is connecting mode j in phase g and mode $j+1$ (when $j+1 \leq N_{\max}$). Note that this transition does not appear in any sub-run spanned by a $C^i_{(j,g)}$ (since it connects two). If a synchronising transition causes the mode change, let r^g_j be some *no-op* rule assumed for convenience. Note, if a synchronising transition causes the mode change, j is the last mode in the phase.

The run π^g_j then is the concatenation of the runs of each $C^i_{(j,g)}$ for each i in any order. Then π^g is the concatenation of each π^g_j in order of the js, using the transition r^g_j to connect each mode together.

Finally, we append each π^g in order of the gs to form the required run of $\mathbb{C}$. To glue each π^g together, we use a synchronising transition that changes the mode if required. We know that this is possible since `Syncs` is satisfied. That the counters tests and actions are valid follows from `Respect` and `EndVal`. Note that `Syncs` also ensures that all processes agree on whether a synchronising transition also caused a mode change.

C Alternative Synchronisation-Bounded Reachability Construction

We present an alternative encoding of the synchronisation-bounded reachability problem that results in smaller formulas than the encoding presented in the main body. However, the reduced size, experimentally, comes at the cost of slower solve times.

In the main text, we kept a component g in each $\mathcal{P}'$ that we constructed, which stored the number of synchronisation-switches that had taken place. Conceptually, however, this number can be kept in a simple counter, and hence can be stored in the mode vectors. Doing this naively, however, would result in many more transitions of the pushdown systems as it would have to perform counter checks to determine what synchronisation actions to perform. However, if we make the encoding aware of the special nature of this extra counter, we do not require any additional rules. However, this means that we require a more detailed explanation of the reduction.

The final formula will take the shape

$$\exists \boldsymbol{m}_1, \ldots, \boldsymbol{m}_{N_{\max}} \exists \boldsymbol{z}_1 \ldots \boldsymbol{z}_m \left(\begin{array}{c} \texttt{Init}(\boldsymbol{m}_1) \wedge \texttt{GoodSeq}(\boldsymbol{m}_1, \ldots, \boldsymbol{m}_{N_{\max}}) \\ \wedge \bigwedge\limits_{1 \leq i \leq m} \texttt{Image}_i(\boldsymbol{z}_i) \\ \wedge\ \texttt{Respect}\left(\sum_{1 \leq i \leq m} \boldsymbol{z}_i, \boldsymbol{m}_1, \ldots, \boldsymbol{m}_{N_{\max}}\right) \\ \wedge\ \texttt{OneChange}\left(\sum_{1 \leq i \leq m} \boldsymbol{z}_i\right) \\ \wedge\ \texttt{EndVal}\left(\sum_{1 \leq i \leq m} \boldsymbol{z}_i\right) \wedge\ \texttt{Syncs}\left(\sum_{1 \leq i \leq m} \boldsymbol{z}_i\right) \end{array} \right)$$

where the formulas $\mathtt{OneChange}\left(\sum_{1\leq i\leq m}\boldsymbol{z}_i\right)$ and $\mathtt{Syncs}\left(\boldsymbol{z}_1,\ldots,\boldsymbol{z}_m\right)$ are the main differences compared to the single thread case. In addition, further adaptations need to be made within other aspects of the formula.

C.1 The Mode Vectors

We begin with the vectors $\boldsymbol{m}_1,\ldots,\boldsymbol{m}_{N_{\max}}$. This part remains largely unchanged from the first reduction, except for the addition of a mode component corresponding to the number of synchronisations performed so far.

A vector in $\mathtt{REG}^n \times [0,r]^n \times \{\uparrow,\downarrow\}^n \times [0,k]$ is said to be a *mode vector*. Given a path π from configurations c to c', we may associate a mode vector to each configuration in π that records for each counter: which region its value is in, how many reversals its used, whether its phase is non-decrementing ($\uparrow$) or non-incrementing ($\downarrow$) and, in the final component, how many synchronisations have taken place. There are at most $N_{\max} := |\mathtt{REG}| \times (r+1) \times n + (k+1) = 2hn(r+1)+(k+1)$ distinct mode vectors in any valid sequence of mode vectors.

C.2 Constructing $\mathcal{P}'_i$

We define the pushdown automata

$$\mathcal{P}'_i = \left(\mathcal{Q}'_i, \Sigma_i, \Gamma', \Delta'_i, \left(q^0_i,1,1\right), \{f_i\}\times[1,N_{\max}]\right)$$

for each $\mathcal{P}_i$ in $\mathbb{C}$ and assume that all $\mathcal{Q}_i$ are pairwise disjoint.

As before $\mathcal{P}'_i$ will make "visible" the counter tests, counter updates and synchronisations performed. Additional constraints introduced later ensure that these operations are valid.

More formally, let $\mathcal{Q}'_i = \mathcal{Q}_i \times [1,N_{\max}]$. We define Γ' implicitly from the transition relation. In fact, Γ' is a (finite) subset of

$$\begin{aligned} &\{\ (\mathtt{ctr}_e,u,j,l)\ \mid e\in[1,n], u\in\mathbb{Z}, j\in[1,N_{\max}], l\in\{0,1\}\ \} \\ &\qquad \cup\ (Const_X \times [1,N_{\max}]) \\ &\qquad \cup \bigcup_{1\leq i\leq m} (StateCons_{\times}\mathcal{Q}_i \times \mathcal{Q}_i \times [1,N_{\max}])\,. \end{aligned}$$

Here, elements in $\{0,1\}$ signify whether the mode vector should change. We define Δ'_i to be the smallest set such that, if $(q,a,\theta) \overset{\gamma}{\hookrightarrow} (q',w,\boldsymbol{u}) \in \Delta_i$ where $\boldsymbol{u} = (u_1,\ldots,u_n)$ then for each $e\in[1,N_{\max}]$, Δ'_i contains

$$((q,e),a) \xhookrightarrow{(\theta,e)(\mathtt{ctr}_1,u_1,e,0)\ldots(\mathtt{ctr}_n,u_n,e,0)} ((q',e),w)$$

and, if $e\in[1,N_{\max})$, Δ'_i also has

$$((q,e),a) \xhookrightarrow{(\theta,e)(\mathtt{ctr}_1,u_1,e,1)\ldots(\mathtt{ctr}_n,u_n,e,1)} ((q',e+1),w)\ \ .$$

These rules are the rules required in the single thread case. We need additional rules to reflect the multi-threaded environment. In particular, an external thread may change the mode, or a synchronising transition may occur. To account for this Δ'_i also has for each $q \in \mathcal{Q}_i$ $a \in \Sigma_i$, $e \in [1, N_{\max})$,

$$((q, e), a) \overset{\varepsilon}{\hookrightarrow} ((q, e+1), a)$$

and, for each $q, q' \in \mathcal{Q}_i$, $e \in [1, N_{\max})$ and $(\delta, \theta, \boldsymbol{u}) \in \Delta_g$ where $\boldsymbol{u} = (u_1, \ldots, u_n)$, when $i > 1$,

$$((q, e), a) \xhookrightarrow{\left(\delta, q, q', e\right)} ((q', e+1), a) \ .$$

and when $i = 1$,

$$((q, e), a) \xhookrightarrow{\left(\delta, q, q', e\right)(\theta, e)(\mathtt{ctr}_1, u_1, e, 0)\ldots(\mathtt{ctr}_n, u_n, e, 0)} ((q', e+1), a) \ .$$

Additionally, we have when $i = 1$,

$$((q, e), a) \xhookrightarrow{\left(\delta, q, q', e\right)(\theta, e)(\mathtt{ctr}_1, u_1, e, 1)\ldots(\mathtt{ctr}_n, u_n, e, 1)} ((q', e+1), a) \ .$$

That is, $\mathcal{P}'_i$ non-deterministically guesses the effect of non-internal transitions and $\mathcal{P}'_1$ is responsible for performing the required counter updates. Note that the information contained in the output character (δ, q, q', e) allows us to check that synchronising transitions take place in the same order and in the same modes across all threads.

C.3 Constructing The Formula

Fix an ordering $\gamma_1 < \ldots < \gamma_l$ on Γ'. By f we denote a function mapping γ_i to i for each $i \in [1, l]$. The formula we require is of the form given above. The formulas $\mathtt{Init}, \mathtt{GoodSeq}, \mathtt{Respect}$, and $\mathtt{EndVal}$ are defined as in the single thread case with adjustments to incorporate the number of synchronisations into the mode vector; the reader may refer to [20] for the original definitions and description. For the $\mathtt{Image}_i$ we apply the linear-time algorithm of [40] on each $\mathcal{P}'_i$ to obtain $\mathtt{Image}_i$ such that for each $\boldsymbol{n} \in \mathbb{N}^l$ we have $\boldsymbol{n} \in \mathbb{P}(\mathcal{L}(\mathcal{P}'_i))$ iff $\mathtt{Image}_i\,(\boldsymbol{n})$ holds.

Since a mode vector is a member of $\mathtt{REG}^n \times [0, r]^n \times \{\uparrow, \downarrow\}^n \times [0, k]$, we set $\boldsymbol{m}_e$ to be a tuple of variables $\left\{\ reg^e_j, rev^e_j, arr^e_j \ \mid j \in [1, n]\ \right\}$, where:

- reg^e_j is a variable that will range over $[1, 2h]$ denoting which region the jth counter is in (a number of the form $2i+1$ refers to φ_i, while $2i$ refers to ψ_i).
- rev^e_j is a variable that will range over $[0, r]$ denoting the number of reversals that have been used thus far by the jth counter.
- arr^e_j is a variable that will range over $\{0, 1\}$ denoting the current arrow direction, e.g., $0/1$ for $\uparrow/\downarrow$ (non-decrementing/non-incrementing mode).
- syn^e is a variable that ranges over $[0, k]$ and denotes the number of synchronisations performed thus far.

We begin with $\texttt{Init}(\boldsymbol{m}_1)$, which asserts that the initial mode vector respects the initial configuration $(q, u, \boldsymbol{v})$, where $\boldsymbol{v} = (c_1, \ldots, c_n)$. More precisely,

$$\texttt{Init}(\boldsymbol{m}_1) \equiv \bigwedge_{j=1}^{n} \left(rev_j^1 = 0 \wedge \bigwedge_{i=1}^{h} \left(\begin{array}{c} reg_j^1 = 2i - 1 \iff \varphi_i(c_j) \wedge \\ reg_j^1 = 2i \iff \psi_i(c_j) \end{array} \right) \right) \wedge syn^1 = 0 \ .$$

Next, $\texttt{GoodSeq}$ ensures that the sequence of mode vectors $\boldsymbol{m}_1, \ldots, \boldsymbol{m}_{N_{\max}}$ is valid. For example, if the direction of a counter changes, then an extra reversal is incurred on that counter, or, if a synchronisation occurs, then the synchronisation component is updated accordingly. The formula is a conjunction of smaller formulas defined below. One conjunct says that each rev_j^e must be a number in $[0, r]$. Likewise, we add conjuncts expressing that each reg_j^e (resp. arr_j^e, syn^e) ranges over $[1, 2h]$ (resp. $\{0, 1\}$, $[0, k]$). We also need to state that changes in the direction arrows incur an extra reversal (otherwise, no reversal is incurred). Hence, we add

$$\bigwedge_{j=1}^{k} \bigwedge_{e=1}^{N_{\max}} 0 \leq rev_j^e \leq r \wedge \bigwedge_{j=1}^{k} \bigwedge_{e=1}^{N_{\max}-1} \left(\begin{array}{c} arr_j^e \neq arr_j^{e+1} \Rightarrow rev_j^{e+1} = rev_j^e + 1 \wedge \\ arr_j^e = arr_j^{e+1} \Rightarrow rev_j^{e+1} = rev_j^e \end{array} \right) \ .$$

The sequence $\{reg_j^e\}_{e=1}^{N_{\max}}$ must obey the changes in $\left\{arr_j^e\right\}_{e=1}^{N_{\max}}$. Hence, we have

$$\bigwedge_{j=1}^{k} \bigwedge_{e=1}^{N_{\max}-1} \left(reg_j^e < reg_j^{e+1} \Rightarrow arr_j^{e+1} = 0 \wedge reg_j^e > reg_j^{e+1} \Rightarrow arr_j^{e+1} = 1 \right) \ .$$

I.e., since regions denote increasing intervals, going to higher (resp. lower) regions mean the counter mode must be non-decrementing (resp. non-incrementing). Finally, we need to assert that the number of synchronisations only increases, using

$$\bigwedge_{e=1}^{N_{\max}-1} \left(syn^e = syn^{e+1} \vee syn^e = syn^{e+1} + 1 \right) \ .$$

This completes the definition of $\texttt{GoodSeq}$.

The formula $\texttt{Respect}$ requires that the counter tests and actions fired within a mode are allowed. For example, a subtraction may not occur on a counter in a non-decreasing phase, and counter tests may only occur in sympathetic regions. It remains unchanged from the sequential case, and is reproduced below. Again, this is a conjunction. First, when the jth counter is non-incrementing (resp. non-decrementing), we allow only non-negative (resp. non-positive) counter increments:

$$\bigwedge_{j=1}^{n} \bigwedge_{e=1}^{N_{\max}} \left(\begin{array}{c} arr_j^e = 0 \Rightarrow \bigwedge_{\substack{(\texttt{ctr}_j, u, e, l) \\ u < 0}} z_{f(\texttt{ctr}_j, u, e, l)} = 0 \wedge \\ arr_j^e = 1 \Rightarrow \bigwedge_{\substack{(\texttt{ctr}_j, u, e, l) \\ u > 0}} z_{f(\texttt{ctr}_j, u, e, l)} = 0 \end{array} \right) \ .$$

Secondly, the value of the jth counter at the beginning and end of each mode must respect the guessed mode vector. Let us first introduce some notation. Observe that the value of the jth counter at the *end* of eth mode vector can be expressed by $c_j + \left(\sum_{e'=1}^{e-1} \sum_{(\mathtt{ctr}_j,u,e',l)\in\Gamma'} u \times z_{f(\mathtt{ctr}_j,u,e',l)}\right) + \sum_{(\mathtt{ctr}_j,u,e,0)} u \times z_{f(\mathtt{ctr}_j,u,e,0)}$. Let us denote this term by $\mathit{EndCounter}_j^e$. Similarly, the value at the *beginning* of the eth mode is $c_j + \sum_{e'=1}^{e-1} \sum_{(\mathtt{ctr}_j,u,e',l)\in\Gamma'} u \times z_{f(\mathtt{ctr}_j,u,e',l)}$. We denote this term by $\mathit{StartCounter}_j^e$. Hence, this second conjunct is

$$\bigwedge_{j=1}^{k} \bigwedge_{e=1}^{N_{\max}} \bigwedge_{l=1}^{h} \begin{pmatrix} \mathit{reg}_j^e = 2l-1 \Rightarrow \left(\varphi_l(\mathit{EndCounter}_j^e) \wedge \varphi_l(\mathit{StartCounter}_j^e)\right) \ \wedge \\ \mathit{reg}_j^e = 2l \Rightarrow \left(\psi_l(\mathit{EndCounter}_j^e) \wedge \psi_l(\mathit{StartCounter}_j^e)\right) \end{pmatrix} .$$

Finally, we need to express that no invalid counter tests are executed in a given mode. To test whether a constraint θ is satisfied by the values $\boldsymbol{v} = (c_1, \ldots, c_n)$ of the counters, it is necessary and sufficient to test whether θ is satisfied by *some* vector $\boldsymbol{v}' = (c'_1, \ldots, c'_n)$, where each c_i lies in the same region as c'_i. Therefore, the desired property can be expressed as:

$$\bigwedge_{e=1}^{N_{\max}} \bigwedge_{(\theta,e)\in\Gamma'} z_{f(\theta,e)} > 0 \Rightarrow \theta\left(\mathit{StartCounter}_1^e, \ldots, \mathit{StartCounter}_n^e\right) .$$

The final formula we take from the sequential case is `EndVal`, which checks that the counter operations applied during the run leave each counter in the correct value, as given in the final configuration. Recall that the target configuration is $(q', u', \boldsymbol{v}')$, where $\boldsymbol{v}' = (c'_1, \ldots, c'_n)$. We assert that the end counter values match $\boldsymbol{v}'$. This definition is given as follows.

$$\mathtt{EndVal}(\boldsymbol{z}) \equiv \bigwedge_{j=1}^{n} \left(\sum_{(\mathtt{ctr}_j,u,e,l)} u \times z_{f(\mathtt{ctr}_j,u,e,l)} \right) = c'_j .$$

It remains for us to define `OneChange` and `Syncs`. We use `OneChange` to assert that only one thread may be responsible for firing the transition that changes a given mode of the counters to the next. That is,

$$\mathtt{OneChange}\,(\boldsymbol{z}) \equiv \bigwedge_{\substack{(\mathtt{ctr}_j,u,e,1)\\(\mathtt{ctr}_j,u',e,1)\\u'\neq u}} z_{f(\mathtt{ctr}_j,u,e,1)} > 0 \Rightarrow \begin{pmatrix} z_{f(\mathtt{ctr}_j,u,e,1)} = 1 \\ \wedge\ z_{f(\mathtt{ctr}_j,u',e,1)} = 0 \end{pmatrix} .$$

The role of `Syncs` is to ensure that the synchronising transitions taken by $\mathcal{P}'_1, \ldots, \mathcal{P}'_m$ are valid. Note that, by design, each $\mathcal{P}'_i$ will only output at most one character of the form (δ, q, q', e) for each $e \in [1, N_{\max})$. We define

$$\texttt{Syncs}\,(\boldsymbol{z}) \equiv \bigwedge_{1 \le e < N_{\max}} \left(\begin{array}{l} syn^{e} < syn^{e+1} \Rightarrow \\ \qquad \bigvee_{(\delta,\theta,\boldsymbol{u}) \in \Delta_g} \Big(\texttt{Sync}_{(\delta,e)}\,(\boldsymbol{z}) \;\wedge\; \texttt{AllFired}_{(\delta,e)} \Big) \end{array} \right) \wedge \\ \bigwedge_{1 \le e < N_{\max}} \left(syn^{e} = syn^{e+1} \Rightarrow \bigwedge_{\delta,q,q'} z_{f(\delta,q,q',e)} = 0 \right)$$

where $\texttt{Sync}_{(\delta,e)}$ is δ with each atomic state constraint $y_i = q$ replaced by the formula

$$\bigvee_{(\delta,q,q',e)} z_{f(\delta,q,q',e)} > 0$$

and each atomic state constraint $y'_i = q'$ replaced by the formula

$$\bigvee_{(\delta,q,q',e)} z_{f(\delta,q,q',e)} > 0 \ .$$

Finally,

$$\texttt{AllFired}_{(\delta,e)} \equiv \bigwedge_{1 \le i \le m} \bigvee_{q,q' \in \mathcal{Q}_i} z_{f(\delta,q,q',e)} > 0 \ .$$

C.4 Correctness

We prove the following lemmas, which give the correctness of our encoding.

Lemma 3. *If there is an r-reversal and k-synchronisation-bounded run from c to c', then* `HasRun` *is satisfiable.*

Proof. Assume there is an r-reversal and k-synchronisation-bounded run of $\mathbb{C}$. We show that `HasRun` is satisfiable. Let $\pi = c_0 \xRightarrow{\gamma_1} c_1 \xRightarrow{\gamma_2} \cdots \xRightarrow{\gamma_h} c_h$ be the run of $\mathbb{C}$, where $c = c_0$ and $c' = c_h$.

First, we assign to $\boldsymbol{m}_1, \ldots, \boldsymbol{m}_{N_{\max}}$ the values corresponding to the evolution of the modes in π.

Next, we assign $\boldsymbol{z}_1, \ldots, \boldsymbol{z}_m$. We can divide $c_0 c_1 \ldots c_h$ into $N_{\max}$ phases — according to the changes in mode vectors — such that there exist $-1 = j_0 \le \cdots \le j_{N_{\max}} = h$ and $C_1 = c_0 \ldots c_{j_1}, \ \ldots, \ C_{N_{\max}} = c_{j_{(N_{\max}-1)}+1} \ldots c_{j_{N_{\max}}}$. In addition, for all $e \in [1, N_{\max}]$ and all $j_{(e-1)} < j < j_e$, we have $c_j \xRightarrow{\gamma_{j+1}} c_{j+1}$ is a transition that is not synchronising and does not change a counter mode, and, when $i < N_{\max}$, $c_{j_i} \xRightarrow{\gamma_{j_{i+1}}} c_{j_{i+1}}$ is a synchronising transition or a transition that changes a counter mode. Note, that we allow the phases to be empty (when the start index is greater than the end index).

We will construct accepting runs of each $\mathcal{P}'_i$. For each $j \in [1, N_{\max}]$ extract from C_j all transitions that are due to transitions of $\mathcal{P}_i$. Furthermore, augment all control states with the mode the transition took place in. This results in a run of $\mathcal{P}'_i$.

Finally, we construct an accepting run of $\mathcal{P}'_i$ by concatenating the runs for each phase as follows.

For $i > 1$, we use the rules

$$((q, e), a) \xhookrightarrow{(\delta,q,q',e)} ((q', e + 1), a)$$

to form the connections when $(\delta, \theta, \boldsymbol{u})$ is a synchronising transition used to bridge the phases. For $i = 1$, we use

$$((q, e), a) \xhookrightarrow{(\delta,q,q',e)(\theta,e)(\mathtt{ctr}_1,u_1,e,l)\ldots(\mathtt{ctr}_n,u_n,e,l)} ((q', e), a)$$

where $l = 1$ iff a counter mode is changed by the transition.

When the phases are bridged by a counter update by process $j \neq i$, we use the rules

$$((q, e), a) \xhookrightarrow{\varepsilon} ((q, e + 1), a)$$

to bridge the gap. If the phases are bridged by a counter update by process i, we use the corresponding transition

$$((q, e), a) \xhookrightarrow{(\theta,e)(\mathtt{ctr}_1,u_1,e,1)\ldots(\mathtt{ctr}_n,u_n,e,1)} ((q', e + 1), w)$$

of $\mathcal{P}'_i$.

We assign to $\boldsymbol{z}_i$ the Parikh image of the run defined above. Because we assigned a valid sequence of modes to the mode vectors, and valid Parikh images of $\mathcal{P}'_i$ to $\boldsymbol{z}_i$ that respect the modes, we have that $\mathtt{Init}, \mathtt{GoodSeq}$, and $\mathtt{Image}_i$ are satisfied. To see that $\mathtt{Respect}$ and $\mathtt{EndVal}$ are satisfied, observe that the counter actions come from a valid accepting run. Finally, $\mathtt{OneChange}$ is true since only one thread changed the mode, and $\mathtt{Syncs}$ is satisfied since we assigned valid synchronising transitions to the output tuples (δ, q, q', e).

Lemma 4. *If the formula* $\mathtt{HasRun}$ *is satisfiable, then there is an r-reversal and k-synchronisation-bounded run from c to c'.*

Proof. Take an accepting assignment to $\boldsymbol{m}_1, \ldots, \boldsymbol{m}_{N_{\max}}$ and $\boldsymbol{z}_1, \ldots, \boldsymbol{z}_m$. Because $\mathtt{Init}$, $\mathtt{GoodSeq}$, $\mathtt{Image}_i$, $\mathtt{Respect}$ and $\mathtt{EndVal}$ are satisfied we know that there are runs of $\mathcal{P}'_i$ such that the counter operations in each mode respect the mode vectors (which represent a valid mode sequence) and the initial and target values. We will construct a valid k-synchronisation-bounded run of $\mathbb{C}$.

Let $\pi_i = c^i_0 \xrightarrow{\gamma^i_1} c^i_1 \xrightarrow{\gamma^i_2} \cdots \xrightarrow{\gamma^i_{h_i}} c^i_{h_i}$ be the accepting runs of the $\mathcal{P}'_i$. Since $\mathtt{Syncs}$ is satisfied, we know that each π_i uses the same number of guessed synchronising transitions, and each phase change happens in the same mode of each $\mathcal{P}'_i$. For convenience, assume all k phases are used. For each $g \in [1, k + 1]$, let j_g be the mode the gth phase starts in (this is the same for all i) and j'_g be the mode the gth phase ends in. Thus we can build $C^i_g = C^i_{(j_g,g)} C^i_{(j_g+1,g)} \cdots C^i_{(j'_g,g)}$ for all $g \in [0, k]$ where each $C^i_{(j,g)}$ is the sub-run of $\mathcal{P}'_i$ in mode j and phase g.

Observe that the mode changes during a phase can only be caused by counter updates (since a synchronising transition will end the phase).

For each phase $g \in [0, k]$ and mode $j \in [1, N_{\max}]$ we build the following run π^g_j of $\mathbb{C}$. These will be used to build π^g for each phase. Observe that, since `OneChange` is true, only one mode change rule can be fired amongst all threads. Hence, all but one $\mathcal{P}'_i$ must change modes using the rules

$$((q, e), a) \overset{\varepsilon}{\hookrightarrow} ((q, e + 1), a)$$

in the case when it is not a synchronising transition that causes the mode change, and

$$((q, e), a) \xhookrightarrow{(\delta,q,q',e)} ((q', e + 1), a)$$

in the other case (note that this rule will not be used in a particular $C^i_{(j,g)}$, but rather as the synchronising transition discussed below).

In the first — non-synchronising — case, these rules bridge each $C^i_{(j,g)}$ and $C^i_{(j+1,g)}$ and are not included in π^g_j or π^g. Let r^g_j be the only mode changing transition amongst all is connecting mode j in phase g and mode $j + 1$ (when $j+1 \leq N_{\max}$). Note that this transition does not appear in any sub-run spanned by a $C^i_{(j,g)}$ (since it connects two). If a synchronising transition causes the mode change, let r^g_j be some *no-op* rule assumed for convenience. Note, if a synchronising transition causes the mode change, j is the last mode in the phase.

The run π^g_j then is the concatenation of the runs of each $C^i_{(j,g)}$ for each i in any order. Then π^g is the concatenation of each π^g_j in order of the js, using the transition r^g_j to connect each mode together.

Finally, we append each π^g in order of the gs to form the required run of $\mathbb{C}$. To glue each π^g together, we use a synchronising transition that changes the counters component of mode if required. We know that this is possible since `Syncs` is satisfied. That the counters tests and actions are valid follows from `Respect` and `EndVal`.

D Optimisation Proofs and Definitions

In this section we present proofs and definitions omitted from Section 5.

D.1 Context-Bounded Model Checking

We refine the encoding of context-bounded model checking given in Section 4 by minimising the number of positions where synchronisations may occur, leading to performance gains in our experiments. To do this, we add a boolean flag to the control state that indicates whether a synchronisation is permitted.

Definition 9. *Given a n-ClPDS $\mathbb{C} = (\mathcal{P}_1, \ldots, \mathcal{P}_m, G, X)$. Let $\#$ be a symbol not in G. We define from each $\mathcal{P}_i = (\mathcal{Q}_i, \Sigma_i, \{\varepsilon\}, \Delta_i, X)$ with $\mathcal{Q}_i = G \times \mathcal{Q}'_i$ the pushdown system $\mathcal{P}^S_i = \left(\mathcal{Q}^S_i \cup \{f_i\}, \Sigma_i, \{\varepsilon\}, \Delta^S_i, X\right)$ where $\mathcal{Q}^S_i = \mathcal{Q}^i \times (G \cup \{\#\}) \times \{0, 1\}$ and Δ^S_i is the smallest set containing*

1. *$((g_1, q_1, 0), a, \theta_1) \overset{\gamma}{\hookrightarrow} ((g_1, q_2, 0), w, \boldsymbol{u}_1)$ when we have the rule $((g_1, q_1), a, \theta_1) \overset{\gamma}{\hookrightarrow} ((g_2, q_2), w, \boldsymbol{u}_1) \in \Delta_i$, and*
2. *$((g, q, 0), a, \boldsymbol{tt}) \overset{\varepsilon}{\hookrightarrow} (f_i, w, \boldsymbol{0})$ for all q, a appearing as a head in the final configuration and $g = \#$ or g appears in the final configuration, and*
3. *$((g_1, q_1, 0), a, \boldsymbol{tt}) \overset{\gamma}{\hookrightarrow} ((g_1, q_1, 1), a, \boldsymbol{0})$ whenever we have a rule $((g_1, q_1), a, \theta) \overset{\gamma}{\hookrightarrow} ((g_2, q_2), w, \boldsymbol{u}) \in \Delta_i$ such that*
   (a) an update to g occurs, i.e. $g \neq g'$, or
   (b) an update to the counters occurs, i.e. $\boldsymbol{u}$ contains a non-zero update, or
   (c) the counter values need to be up-to-date, i.e. θ is not independent of the counter values, or
4. *$((g, q_1, 0), a, \boldsymbol{tt}) \overset{\gamma}{\hookrightarrow} ((g, q_1, 1), a, \boldsymbol{0})$ for all $g \in G$ whenever the G component needs to be up-to-date, i.e. there exists some $g' \in G$ such that $((g', q_1), a, \theta) \overset{\gamma}{\hookrightarrow} ((g', q_2), w, \boldsymbol{u}) \in \Delta_i$ but we don't have the rule for all $g' \in G$.*
5. *$((g, q_1, 0), a, \boldsymbol{tt}) \overset{\gamma}{\hookrightarrow} ((g, q_1, 1), a, \boldsymbol{0})$ for all $g \in G$ whenever q_1 appears in the final configuration.*

Finally, let $\mathbb{C}^S = \left(\mathcal{P}_1^S, \ldots, \mathcal{P}_m^S, \Delta_g, X, r\right)$, where $\Delta_g = \{(\delta, \boldsymbol{tt}, \boldsymbol{0})\}$ such that the formula $\delta(q_1^1, \ldots, q_n^1, q_1^2, \ldots, q_n^2)$ holds only when there is some $g \in G$ and $1 \leq i \neq j \leq n$ such that

1. *$q_i^1 = (g, q, 1)$ and $q_i^2 = (\#, q, 0)$ for some q, and*
2. *$q_j^1 = (\#, q, 0)$ and $q_j^2 = (g, q, 0)$ for some q, and*
3. *for all $i' \neq i$ and $i' \neq j$, $q_{i'}^1 = (\#, q, 0)$ and $q_{i'}^2 = (\#, q, 0)$ for some q.*

We show that this simulation is correct.

Theorem 2. *Given a n-ClPDSr $\mathbb{C}$, there is an r-reversal and k-context-bounded run from the initial configuration $(g, q_1, w_1, \ldots, q_m, w_m, \boldsymbol{v})$ to the given final configuration $(g', q_1', w_1', \ldots, q_m', w_m', \boldsymbol{v}')$ iff there is an r-reversal and k-synchronisation-bounded run of $\mathbb{C}^S$ from $((g, q_1, 0), w_1, (\#, q_2, 0), \ldots, (\#, q_m, 0), w_m, \boldsymbol{v})$ to the configuration $(f_1, w_1', \ldots, f_m, w_m', \boldsymbol{v}')$.*

Proof. We prove the only if direction first. Take a run of $\mathbb{C}$. From this we construct a run of $\mathbb{C}^S$. Let $T_1, \ldots, T_k$ be the k phases, which are sequences $t_1^i, \ldots, t_{h_i}^i$ of the transitions fired during the ith phase. We can manipulate these phases such that the last configuration in each phase is of the form $(g_1, \ldots, q_j, w_j, \ldots)$, where j is the active pushdown system and either q_j is in the final configuration, or the next transition applied to $\mathcal{P}_j$ matches one of the cases 3-5 of Definition 9. We do this as follows. First, suppose there are no more counter tests, updates to the counters or g, or transitions depending on g occurring on $\mathcal{P}_j$. The we can move all subsequent transitions of $\mathcal{P}_j$ into the current phase since they neither depend on nor update any globally accessible information. Otherwise, we take all transitions occurring before the next such transition, and move them into the current phase. There are then two cases: either the last configuration has a final state of $\mathcal{P}_j$ or the next transition of $\mathcal{P}_j$ (in a later phase) leading from the last configuration matches a case 3-5 of Definition 9.

Let $\left(g^1, q_1^1, w_1^1, \ldots, q_m^1, w_m^1, \boldsymbol{v}^1\right)$ be the initial configuration of phase i and $\left(g^2, q_1^1, w_1^1, \ldots, q_j^2, w_j, \ldots q_m^1, w_m^1, \boldsymbol{v}^2\right)$ be the last. Using rules from case 1 of Definition 9 we obtain a run from

$$\left(\left(\#, q_1^1, 0\right), w_1^1, \ldots, \left(g^1, q_j^1, 0\right), w_j, \ldots, \left(\#, q_m^1, 0\right), w_m^1, \boldsymbol{v}^1\right)$$

to

$$\left(\left(\#, q_1^1, 0\right), w_1^1, \ldots, \left(g^2, q_j^2, 0\right), w_j, \ldots, \left(\#, q_m^1, 0\right), w_m^1, \boldsymbol{v}^2\right) \ .$$

Then, both of the above cases for the last configuration imply a transition to the configuration

$$\left(\left(\#, q_1^1, 0\right), w_1^1, \ldots, \left(g^2, q_j^2, 1\right), w_j, \ldots, \left(\#, q_m^1, 0\right), w_m^1, \boldsymbol{v}^2\right)$$

from which we can apply a global synchronisation to the first configuration of the next phase. We can then use rules from case 2 at the end of the last phase to reach the final configuration. Hence, since there are k phases, only k synchronisations are required, and we have a k-synchronisation-bounded run of $\mathbb{C}^S$.

In the if direction, we take an r-reversal and k-synchronisation-bounded run of $\mathbb{C}^S$. We separate this into phases $T_1, \ldots, T_k$ such that each T_i is connected to the next by a global synchronisation. By construction, during each T_i only one $\mathcal{P}_j$ may move, since all but one have # in their global component, from which no transitions are possible.

Next, we remove the last transition of each phase which made a change of the final component of some control state from 0 to 1. Note that now each phase only contains 0 in the last component of the control states. Finally, the last transition of each $\mathcal{P}_i^S$ replaces some $\left(g^i, q_i', 0\right)$ with f_i where g^i is either # or g'. Note that by construction, all but one g^i will be #, and the other will be g'. We remove these final transitions. It is now straightforward to project the run in each phase to a run of $\mathbb{C}$. These runs compose to form a k-context-bounded run of $\mathbb{C}$ satisfying the properties required.

D.2 Minimising Communicating Pushdown Systems with Counters

We show that the minimisation by elimination of removable heads — given in Definition 8 — preserves reachability properties.

Theorem 3. *For any r-reversal and k-synchronisation-bounded run from the initial to final configuration of a given n-SyncPDSr with a pushdown system $\mathcal{P}$ with n counters and a removable head q, a, there exists an r-reversal and k-synchronisation-bounded accepting run of the n-SyncPDS with $\mathcal{P}$ replaced by $\mathcal{P}_{q,a}$ with the same Parikh image, and vice versa.*

Proof. We can divide the run of the n-SyncPDS in to a number h of phases which we will represent as $T_0, t_1, T_1 \ldots, T_{h-1}, t_h, T_h$ where each T_i is a sequence of rules (transitions) of a pushdown system in the n-SyncPDS. The phases are such that between each t_i is either a global synchronisation or non-zero counter modification, and there are no such transitions within an R_i.

Now, take any two transitions t and t' in the run belonging to $\mathcal{P}$ that cross a configuration of $\mathcal{P}$ with the head q, a. Note that this cannot include any rules t_i since these are global synchronisations or counter updates, and no counter update rule can contain a removable head. Let c_1, c_2 and c_3 be the configurations of $\mathcal{P}$ these rules connect. Observe that since c_2 has a removable head, and removable heads cannot loop, neither c_1 nor c_3 have the head q, a.

If t and t' occur within the same T_i then we can safely replace them with the rule formed by their concatenation as in Δ_1 of the definition of $\mathcal{P}_{q,a}$. This is because the counters are not changed and no global synchronisation used the configuration c_2.

If t appears in T_i and t' in T_j such that $i < j$, then we can perform the same replacement. This remains safe for two reasons. First, we know q, a is removable and hence t' did not contain a counter test, so any counter updates between T_i and T_j will not affect the applicability of t' (hence it can be merged into t). Secondly, if any of the transitions between T_i and T_j are global synchronisations, we can use the fact that for q, a to remain removable, it must be the case that q is a sink state. Hence, the control state after t' remains q, and hence the global synchronisations can still be applied.

Note that when combining rules, in order to remain within the definition of Δ_1, we need that t is not a pop transition. This is implicit in c_2 having the head q, a: if t were a pop transition, then q, a would be a return location, and hence not removable.

Finally, we need to check that no removed transitions are used in the modified run. This follows since all instances of that are not in the initial or final configuration q, a have been removed. Furthermore, since q, a is removable, it cannot occur in the initial configuration.

After these modifications, we have a run of $\mathcal{P}_{q,a}$ with the same Parikh image.

The proof in the opposite direction is straightforward: take any run of $\mathcal{P}_{q,a}$. For each transition t in Δ_1 there exist two rules in Δ with which t may be replaced. The only subtlety in this replacement is that the counter test θ_2 may be invalidated by the update $\boldsymbol{u}_1$. However, a condition on q, a being a removable head is that θ_2 does not depend on the counter values, hence the two transitions may safely replace t.

D.3 Minimising CFG size via pushdown reachability table

We describe an algorithm for computing the reachability lookup table specialised for PDA from section 3. It is an adaptation of the *pre** algorithm from [16].

E Pushdown Translator Input Language

We present the syntax of the pushdown translator tool. A simple example program is as follows:

Algorithm 1 Compute reachability lookup table for PDA output from our translation

```
Input: $\mathcal{P} = \left(\mathcal{Q}, \Sigma, \Gamma, \Delta, \left(q^0, 1, 1\right), \{f\} \times [1, N_{\max}] \times [1, k+1]\right)$
Output: table $rel$ with entries of form $(p, a, q, d, d')$
  $rel := \emptyset$; $\Delta' := \emptyset$; $\Delta_1 := \emptyset$;
  for all $t \in \Delta$ such that $t = ((p, e, g), a) \overset{\gamma}{\hookrightarrow} ((p', e', g'), \epsilon)$ do
    $d_1 := e' - e$; $d_2 := g' - g$;
    add $(p, a, q, d_1, d_2)$ to $trans$;
  end for
  for all $t \in \Delta$ do
    $t = ((p, e, q), a) \overset{\gamma}{\hookrightarrow} ((p', e', g'), w)$;
    $d_1 := e' - e$; $d_2 := g' - g$;
    add $(p, a, q, w, d_1, d_2)$ to $\Delta_1$;
  end for
  while $trans \neq \emptyset$ do
    pop $t = (q, b, q', d_1, d_2)$ from $trans$;
    if $(q, b, q', d_1', d_2) \notin rel$ for each $d_1' \leq d_1$ then
      add $t$ to $rel$;
      for all $(p, a, q, b, d_1', d_2') \in \Delta_1 \cup \Delta'$ do
        if $(p, a, q', d, d_2 + d_2') \notin trans$ for each $d \leq d_1 + d_1'$ then
          add $(p, a, q', d_1 + d_1', d_2 + d_2')$ to $trans$;
        end if
      end for
      for all $(p, a, q, bc, d_1', d_2') \in \Delta_1$ do
        if $(p, a, q', c, d, d_2 + d_2') \notin \Delta'$ for each $d \leq d_1 + d_1'$ then
          add $(p, a, q', c, d_1 + d_1', d_2 + d_2')$ to $\Delta'$;
        end if
        for all $(q', c, q'', d_1'', d_2'') \in rel$ do
          if $(p, a, q'', d, d_2 + d_2' + d_2'') \notin trans$ for each $d \leq d_1 + d_1' + d_1''$ then
            add $(p, a, q'', d_1 + d_1' + d_1'', d_2 + d_2', d_2'')$ to $trans$;
          end if
        end for
      end for
    end if
  end while
  return $rel$
```

```
counter c reversals 0

start main1
start main2

constraint (once == 1) && (twice == 2) && (c == 1)

switches 1

procedure main1()
begin
  c++;
  echo once;
end;

procedure main2()
begin
  echo twice;
  echo twice;
end;
```

E.1 Program Header

These statements appear before the program procedure declarations and specify the shape of the program.

`shared bool` *ident*

Global boolean variable shared between all threads.

`bool` *ident*

Global boolean variable each thread has a copy of.

`counter` *ident* `reversals` *n*

Global shared counter variable with *n* reversals.

`start` *ident*

A thread starting with method *ident* (first thread is first run).

`constraint` *cc*

A counter constraint *cc* that should hold at the end of a run for an error to have occurred.

`switches` *n*

The number of context-switches.

E.2 Program Body

Procedures are defined with the following syntax.

`procedure` *ident* (`bool` *ident*, . . . , `bool` *ident*)
    `bool` *ident*
    . . .
    `bool` *ident*
    statement

> A method declaration taking several parameters and having several local variables. Note that return values should be passed through a global variable.

Statements then have the following form.

ident: *statement*

> A statement labelled *ident*.

`begin`
    statement
    . . .
    statement
`end`;

> A sequence of statements.

ident = *vc*, . . . , *ident* = *vc*;

> Assignment of boolean variables with boolean constraint evaluations. The value of the right hand sides is determined by the variable values preceding the statement.

`lock` *ident*;

> Locks a shared boolean variable. This assigns 1 to the variable if it is 0, and blocks otherwise.

`unlock` *ident*;

> Unlocks a shared boolean variable. This assigns 0 to the variable if it is 1, and blocks otherwise.

ident ++;

> Increment by one the counter *ident*.

ident −−;

> Decrement by one the counter *ident*.

ident += *n*;

> Increment the counter *ident* by *n*.

ident `-=` *n*;

Decrement the counter *ident* by *n*.

`if` {*vc*} [*cc*] `then`
  statement
`else`
  statement

A conditional branch. Note that either the boolean variable condition {*vc*} or the counter condition [*cc*] may be omitted. If both are present, both must hold. Both may be replaced by `??` for a non-deterministic branch. The `else` branch may also be omitted.

`while` {*vc*} [*cc*] `do`
  statement

While loop. The conditions are as in the if statement above.

`switch`
  `case`: *statement*
  ...
  `case`: *statement*
`end`;

A non-deterministic branch: executes one of the `case` statements.

`goto` *ident*;

Jump to a given statement label.

ident (*vc*, ..., *vc*);

Call procedure *ident* with parameter values given by the *vc*.

`return`;

Return from a procedure. Note, return values should be passed using global variables.

`assert` {*vc*} [*cc*];

Assert a condition. Note that either the boolean variable condition {*vc*} or the counter condition [*cc*] may be omitted. If both are present, both must hold. An error is found if an assertion evaluations to false.

`skip`;

A no-op statement.

`echo` *ident*;

Output a *ident* action. The counts of these actions can be used in the constraint specified using `constraint`.