LTL Model Checking for Communicating Concurrent Programs

Adrien Pommellet, Tayssir Touili

Noname manuscript No.
(will be inserted by the editor)

LTL Model Checking for Communicating Concurrent Programs

Adrien Pommellet · Tayssir Touili

June 5, 2020

Abstract We present in this paper a new approach to the static analysis of concurrent programs with procedures. To this end, we model multi-threaded programs featuring recursive procedure calls and synchronisation by rendez-vous between parallel threads with *communicating pushdown systems* (from now on CPDSs).

The reachability problem for this particular class of automata is unfortunately undecidable. However, it has been shown that an efficient abstraction of the execution traces language can nonetheless be computed. To this end, an algebraic framework to over-approximate context-free languages has been introduced by Bouajjani et al.

In this paper, we combine this framework with an automata-theoretic approach in order to approximate an answer to the model checking problem of the *linear-time temporal logic* (from now on LTL) on CPDSs. We then present an algorithm that, given a *single-indexed* or *stutter-invariant* LTL formula, allows us to prove that no run of a CPDS verifies this formula if the procedure ends.

1 Introduction

The use of parallel programs has grown in popularity in the past fifteen years, but these stay nonetheless fickle and vulnerable to specific issues such as race conditions or deadlocks. Static analysis methods for this class of programs remain therefore more relevant than ever.

The *model checking* framework has proven to be a cornerstone of modern static analysis techniques. The program is modelled as a simpler abstract mathematical *model*. Desirable properties and forbidden behaviours are then expressed using a well-defined *logical* framework, then checked against the abstract mathematical model of the program.

The *linear-time temporal logic* (LTL) encodes properties about the future of execution paths, that is, the sequence of configurations the model goes through. It can be used to express safety and liveness properties.

Pushdown systems (PDSs) were introduced to model the *call stack* of a program that stores information about the active procedures such as return addresses, passed parameters and local variables. Without such a stack, a finite-state automaton can't represent accurately programs with nested, recursive function calls, hence, the need for a more expressive model.

PDSs are a natural model for programs with sequential, recursive procedure calls [7]. Thus, networks of pushdown systems can be used to model multi-threaded programs, where each PDS in the network models a sequential component of the whole program.

Communicating pushdown systems (CPDSs) were introduced by Bouajjani et al. in [3] as a model for communicating multi-threaded programs. It is a natural abstraction: each thread is modelled as a PDS, and can synchronize by rendez-vous with other threads. Unfortunately, it has been proven by Ramalingam [16] that the reachability problem is undecidable for CPDSs. Therefore, the set of execution paths cannot be computed in an exact manner. To overcome this problem, Bouajjani et al. computed an abstraction of the execu-

Adrien Pommellet
LRDE, EPITA, 14-16 Rue Voltaire,
94270 Le Kremlin-Bicêtre, France
Tel.: +33672868668
E-mail: adrien@lrde.epita.fr

Tayssir Touili
LIPN, CNRS, and Université Paris 13,
99 Avenue Jean Baptiste Clément,
93430 Villetaneuse, France

tion paths language, using a framework based on Kleene algebras.

Solving the model checking problem of LTL for the class of CPDSs would be a worthy addition to the existing verification techniques. However, this problem is obviously undecidable: we therefore seek an approximate answer.

Our contributions in this paper are the following:

– We define the semantics of *single-indexed* LTL formulas for CPDSs, that is, formulas of the form $\varphi = (\psi_1, \ldots, \psi_n)$, where each LTL sub-formula ψ_i must hold for the i-th process with regards to the synchronized CPDS semantics.
– We show how to abstract the set of accepting traces of a Büchi pushdown system. To do so, we use the abstraction framework of [3] as well as the LTL model checking methods for PDSs introduced by Esparza et al. in [7].
– We use this abstraction on isolated pushdown components of the whole program in order to approximate the single-indexed LTL model checking problem for CPDSs.
– As a new contribution of the journal version of this article, we extend the abstraction framework to *universal* single-indexed model checking and *stutter invariant* LTL formulas.
– We apply this abstraction framework to detect race conditions in a toy example.

Some of these results were first presented in the conference version of this paper [13].

Related Work. *Multi-stack pushdown systems* (from then on MPDSs) are pushdown systems with two or more stacks; this class of automata can be used to model synchronized parallel programs. Qadeer et al. solved the model checking problem of LTL given a context bounding constraint on runs in [14], where a context is an uninterrupted sequence of actions on a single thread. This result still holds with a weaker phase-bounding constraint, where only a single stack can be popped from during a phase, as proven by La Torre et al. in [20].

Atig introduced in [1] *ordered multi-pushdown automata*, a sub-class of MPDSs such that the stacks are ordered and only the first-non empty stack can be popped from. Given this constraint, the model checking problem of LTL can be solved within an 2-ETIME upper bound. These models depend on bounding constraints on runs; our abstraction framework, while less accurate, does not.

Dynamic pushdown networks (DPNs) were introduced by Bouajjani et al. in [4]. A DPN models a concurrent program as a network with an unbounded number of pushdown components that can spawn new threads, also modelled as pushdown systems. Song et al. described in [18] a model checking framework for *single-indexed* LTL and CTL formulas. A DPN can spawn new threads according to a finite number of patterns, since it has a finite number of rules. A single-indexed formula on a DPN is defined as a tuple $\varphi = (\psi_1, \ldots, \psi_n)$, where each component ψ_i is a formula that must hold for the i-th thread pattern. While CPDSs can't model thread spawns, DPNs do not feature synchronization between threads, a crucial aspect of concurrent programs. Song et al. later added in [19] locks that prevent some transitions from being performed by a thread if a common resource shared with other components has not been released beforehand. This is a weaker form of synchronization than communication by rendez-vous.

Synchronized dynamic pushdown networks (DPNs) were later introduced by Pommellet et al. in [12]. The reachability problem for this class of automata is undecidable but can be abstracted. Abstractions for the model checking problem, however, have yet to be defined.

Paper outline. The paper is organised as follows. In Section 2, we present *communicating pushdown systems* (CPDSs) and detail how they can be used to model programs. In Section 3, we define the *single-indexed linear-time temporal logic* for CPDSs and remind the reader of results on the LTL model checking problem for PDSs. We briefly describe in Section 4 the abstraction framework designed by Bouajjani et al. in [3] to over-approximate the set of execution paths of a PDS. In Section 5, as a main contribution of this paper, we introduce an abstract model checking algorithm for single-indexed LTL. We then extend the abstraction framework to *stutter-invariant* LTL formulas in Section 6. We then apply this scheme in order to detect a race condition in Section 7. Finally, we show our conclusion in Section 8.

2 Communicating Pushdown Systems

2.1 Pushdown systems

Pushdown systems are a natural model for sequential programs with recursive procedure calls.

Definition 1 (Pushdown system) A *pushdown system* (PDS) is a tuple $\mathcal{P} = (P, \Sigma, \Gamma, \Delta, c_0)$ where P is a finite set of control states, Σ a finite input alphabet, Γ a finite stack alphabet, $\Delta \subseteq P \times \Gamma \times \Sigma \times P \times \Gamma^*$ a finite set of transition rules, and $c_0 \in P \times \Gamma^*$ a starting configuration.

If $d = (p, \gamma, a, p', w) \in \Delta$, we write $d = (p, \gamma) \xrightarrow{a} (p', w)$. We call a the *label* of d. We can assume without loss of generality that $\Delta \subseteq P \times \Gamma \times \Sigma \times P \times \Gamma^{\leq 2}$.

A *configuration* of $\mathcal{P}$ is a pair $\langle p, w \rangle$ where $p \in P$ is a control state and $w \in \Gamma^*$ a stack content. Let $Conf_{\mathcal{P}} = P \times \Gamma^*$ be the set of all configurations of $\mathcal{P}$.

The reachability relation. For each $a \in \Sigma$, we define the transition relation $\xrightarrow{a}_{\mathcal{P}}$ on configurations as follows: if $(p, \gamma) \xrightarrow{a} (p', w) \in \Delta$, for each $w' \in \Gamma^*$, $\langle p, \gamma w' \rangle \xrightarrow{a}_{\mathcal{P}} \langle p', ww' \rangle$. Intuitively, the automaton moves from state p to p' while the symbol γ is popped from the stack and a word w is pushed on the stack.

From these relations, we can then infer the *immediate successor* relation $\rightarrow_{\mathcal{P}} = \bigcup_{a \in \Sigma} \xrightarrow{a}_{\mathcal{P}}$. The *reachability* relation $\Rightarrow_{\mathcal{P}}$ is the reflexive and transitive closure of the immediate successor relation $\rightarrow_{\mathcal{P}}$. If $\mathcal{C}$ is a set of configurations, we introduce its set of *predecessors* $pre^*(\mathcal{P}, \mathcal{C}) = \{c \in P \times \Gamma^* \mid \exists c' \in \mathcal{C}, c \Rightarrow_{\mathcal{P}} c'\}$. We may omit the variable $\mathcal{P}$ when only a single PDS is being considered.

A *run* r starting from a configuration c is a sequence of configurations $r = (r_i)_{i \geq 0}$ such that $r_0 = c$ and $\forall i \geq 0$, $r_i \xrightarrow{a_i}_{\mathcal{P}} r_{i+1}$. The word $(a_i)_{i \geq 0}$ is then said to be the *trace matched to* r. Traces and runs may be finite or infinite.

Let $Runs_\omega(\mathcal{P}, \mathcal{C})$ (resp. $Runs(\mathcal{P}, \mathcal{C})$) be the set of all infinite (resp. finite) runs of $\mathcal{P}$ starting from a configuration $c \in \mathcal{C}$. We define $Traces_\omega(\mathcal{P}, \mathcal{C})$ and $Traces(\mathcal{P}, \mathcal{C})$ in a similar manner.

If $\mathcal{P}$'s initial configuration is c_0, we introduce the set $Runs_\omega(\mathcal{P}) = Runs_\omega(\mathcal{P}, \{c_0\})$. We define $Runs(\mathcal{P})$, $Traces(\mathcal{P})$, and $Traces_\omega(\mathcal{P})$ in a similar manner.

Many static analysis methods rely on being able to determine whether a given critical state is reachable or not from the starting configuration of a program, hence, the need for reachability analysis techniques.

Regular sets of configurations. A set of configurations $\mathcal{C}$ of a PDS $\mathcal{P}$ is said to be *regular* if $\forall p \in P$, there exists a finite-state automaton $\mathcal{A}_p$ on the alphabet Γ such that $\mathcal{L}(\mathcal{A}_p) = \{w \mid \langle p, w \rangle \in \mathcal{C}\}$, where $\mathcal{L}(\mathcal{A})$ stands for the language recognized by an automaton $\mathcal{A}$.

The following property holds:

Theorem 1 (Caucal [5]) *Given a PDS $\mathcal{P}$ and a regular set of configurations $\mathcal{C}$, the set of configurations $pre^*(\mathcal{C})$ is regular.*

Moreover, $pre^*(\mathcal{C})$ is effectively computable [2].

2.2 The model and its semantics

Let us consider a program with n threads. Let Act be a set of actions (input alphabet) such that:

– Act contains a special action τ that represents internal actions of a thread.
– Each pair (i, j) of threads in the network uses a dedicated set of signals $Lab_{i,j} = Lab_{j,i}$ disjoint from the other sets of signals and from the internal action τ. If we define $Lab_i = \bigcup_{j \neq i} Lab_{i,j}$, then $Act/\{\tau\} = \bigcup_{i=1,\ldots,n} Lab_i$; we write $Lab = Act/\{\tau\}$.

We then consider the following concurrent pushdown model:

Definition 2 (Bouajjani et al. [3]) A *communicating pushdown system* (CPDS) on the alphabet Act is a tuple $\mathcal{CP} = (\mathcal{P}_1, \ldots, \mathcal{P}_n)$ of pushdown systems such that each component $\mathcal{P}_i$ has input alphabet $Lab_i \cup \{\tau\}$.

Configurations and transitions. A *global configuration* of $\mathcal{CP}$ is a n-tuple $g = (c_1, \ldots, c_n)$ of configurations in $Conf_{\mathcal{P}_1} \times \ldots \times Conf_{\mathcal{P}_n} = Conf_{\mathcal{CP}}$. The *global starting configuration* of $\mathcal{CP}$ is the tuple $g_0 = (c_0^1, \ldots, c_0^n)$ where c_0^i is the starting configuration of the component $\mathcal{P}_i$.

We define the following transition relation $\xrightarrow{a}_{\mathcal{CP}}$ on global configurations:

– $(c_1, \ldots, c_n) \xrightarrow{\tau}_{\mathcal{CP}} (c'_1, \ldots, c'_n)$ if there exists an index i such that $c_i \xrightarrow{\tau}_{\mathcal{P}_i} c'_i$ and $c_j = c'_j$ for all $j \neq i$. A single process applies a pushdown operation on its own stack.
– $(c_1, \ldots, c_n) \xrightarrow{a}_{\mathcal{CP}} (c'_1, \ldots, c'_n)$ if there exist two indices i and j, $i \neq j$, such that $a \in Lab_{i,j}$, $c_i \xrightarrow{a}_{\mathcal{P}_i} c'_i$, $c_j \xrightarrow{a}_{\mathcal{P}_j} c'_j$ and $c_k = c'_k$ for all $k \neq i$ and $k \neq j$. Two synchronized processes perform a simultaneous action.

Intuitively, a thread can either perform an internal action labelled by τ, or must synchronize with another thread in order to perform an action labelled by a common synchronization signal in Lab.

We define runs and traces with regards to this transition relation in a manner similar to PDSs: a run is a sequence of global configurations starting from g_0 and connected by transition rules; its matching trace is the sequence of labels of these transitions.

Example 1 (A global run) We consider three runs

$$a_1 \xrightarrow{\tau}_{\mathcal{P}_1} b_1 \xrightarrow{x}_{\mathcal{P}_1} c_1$$
$$a_2 \xrightarrow{x}_{\mathcal{P}_2} b_2 \xrightarrow{y}_{\mathcal{P}_2} c_2$$
$$a_3 \xrightarrow{y}_{\mathcal{P}_3} b_3$$

on three PDSs $\mathcal{P}_1$, $\mathcal{P}_2$, and $\mathcal{P}_3$. Then

$$\begin{aligned}(a_1, a_2, a_3) &\xrightarrow{\tau}_{\mathcal{CP}} (b_1, a_2, a_3)\\ &\xrightarrow{x}_{\mathcal{CP}} (c_1, b_2, a_3)\\ &\xrightarrow{y}_{\mathcal{CP}} (c_1, c_2, b_3)\end{aligned}$$

is a global run g of $(\mathcal{P}_1, \mathcal{P}_2, \mathcal{P}_3)$. An internal transition of $\mathcal{P}_1$ is followed by a synchronized transition between $\mathcal{P}_1$ and $\mathcal{P}_2$, then another synchronized transition between $\mathcal{P}_2$ and $\mathcal{P}_3$.

Given a global run g, we define g^i as its projection on its i-th component $Conf_{\mathcal{P}_i}$.

Example 2 (Projecting a global run) The projection g^3 of the global run

$$\begin{aligned}(a_1, a_2, a_3) &\xrightarrow{\tau}_{\mathcal{CP}} (b_1, a_2, a_3)\\ &\xrightarrow{x}_{\mathcal{CP}} (c_1, b_2, a_3)\\ &\xrightarrow{y}_{\mathcal{CP}} (c_1, c_2, b_3)\end{aligned}$$

on $Conf_{\mathcal{P}_3}$ is

$$a_3 \xrightarrow{\tau}_{\mathcal{CP}} a_3 \xrightarrow{x}_{\mathcal{CP}} a_3 \xrightarrow{y}_{\mathcal{CP}} b_3$$

Note that it is not a run of $\mathcal{P}_3$.

2.3 From a concurrent program to a CPDS model

We can assume that the concurrent program is represented by a n-tuple of *control flow graphs*, whose nodes represent control points of threads or procedures and whose edges are labelled by statements.

These statements can be variable assignments, procedure calls or returns, or communications between threads through unidirectional point-to point channels, where a thread sends a value x through a channel ch and another thread waits for this value then assigns it to a variable.

Without loss of generality, we assume that each thread can share some variables (from now on called thread variables) with the procedures it calls, but that there are no global variables shared between threads. We also consider that both local and thread variables can only take a finite number of values by abstracting their domain if needed.

As a consequence, threads can only synchronize through unidirectional, point-to-point channels: for all $1 \leq i, j \leq n$, $i \neq j$, there is a channel $ch_{i,j}$ that allows thread i to send values to another thread j. With a send statement $ch_{i,j}!(x)$, a value x is sent through channel $ch_{i,j}$ from thread i to thread j. With a receive statement $ch_{i,j}?(x)$, the value x from thread i received through channel $ch_{i,j}$ is assigned to a variable in thread j.

For each control flow graph, we will define a corresponding PDS $\mathcal{P} = (P, Act, \Gamma, \Delta, c_{init})$. The set of states P is the set of all possible valuations of thread variables (i.e. variables shared amongst procedures called by this thread). The stack alphabet Γ is the set of all pairs (n, l) where n is a node of the control flow graph and l is a valuation of the local variables of the current procedure. The whole program will be modelled by a tuple of these PDSs.

$Lab_{i,j}$ is the set of all possible synchronization actions $ch_{i,j}(x)$ between the components $\mathcal{P}_i$ and $\mathcal{P}_j$: value x is carried from the i-th thread to the j-th thread through channel $ch_{i,j}$. The set $Act = \tau \bigcup\limits_{i \neq j} Lab_{i,j}$ also contains an internal action τ.

For each statement s labelling an edge of the flow graph between nodes n_1 and n_2, we introduce the following transition rules in the corresponding PDS, where t_1 and t_2 (resp. l_1 and l_2) are the valuations of thread (resp. local) variables before and after the execution of the statement:

- If s is an assignment, it is represented by a rule of the form:
  $$(t_1, (n_1, l_1)) \xrightarrow{\tau} (t_2, (n_2, l_2))$$
  Assigning a new value to a variable in t_1 or l_1 results in a new valuation t_2 or l_2. Note that we have either $t_1 = t_2$ or $l_1 = l_2$, depending on whether the value is assigned to a local variable (in the former case) or a thread variable (in the latter case).
- If s is a procedure call, it is represented by a rule of the form
  $$(g_1, (n_1, l_1)) \xrightarrow{\tau} (g_1, (f_0, l_0)(n_2, l_1))$$
  where f_0 is the starting node of the called procedure and l_0 the initial valuation of its local variables.
- if s is a procedure return, it is represented a rule of the form
  $$(t_1, (n_1, l_1)) \xrightarrow{\tau} (t_2, \varepsilon)$$
  We simulate returns of values by introducing an additional thread variable and assigning the return value to it in the valuation t_2.
- If s is an assignment $ch?(x)$ of a value x carried through a channel ch, it is represented by a rule of the form
  $$(t_1, (n_1, l_1)) \xrightarrow{ch(x)} (t_2, (n_2, l_2))$$
  where t_1 and t_2 (resp. l_1 and l_2) are such that assigning the value x to a variable in t_1 (resp. l_1) results

in a new valuation t_2 (resp. l_2). Note again that we have either $t_1 = t_2$ or $l_1 = l_2$, depending on whether the variable modified belongs to a procedure or a thread.

– If s is an output $ch!(x)$ through a channel ch of the value x, it is represented by rules of the form

$$(t_1, (n_1, l_1)) \xrightarrow{ch(x)} (t_1, (n_2, l_1))$$

such that the variable copied and sent through the channel has value x in either t_1 or l_1.

Finally, we consider the starting configuration of each process $c_{init} = (t_{init}, (n_{init}, l_{init}))$ where t_{init} and l_{init} are respectively the initial valuations of the thread and local variables, and n_{init} the starting node of its initial procedure.

3 LTL Model Checking on CPDSs

The most widely used variant of temporal logics is the *linear-time temporal logic* LTL introduced by Pnueli in [11].

3.1 The linear-time temporal logic LTL

Let AP be a finite set of *atomic propositions* used to express facts about a program. A *path* is an infinite word $\pi = (\pi_i)_{\geq 0}$ in the set $Paths = (2^{AP})^\omega$.

Definition 3 (LTL) The set of *LTL* formulas is given by the following grammar:

$$\varphi, \psi ::= True \mid p \in AP \mid \neg\varphi \mid \varphi \vee \psi \mid X\ \varphi\ \text{(Next)} \mid \varphi\ U\ \psi\ \text{(Until)}$$

X and U are called the *next* and *until* operators: the former means that a formula should happen at the next step, the latter, that a formula should hold at least until another formula becomes true. We consider the following semantics on paths:

Definition 4 (Semantics of LTL) Let φ be a *LTL* formula, $\pi \in Paths$, and $i \in \mathbb{N}$. We define inductively the semantics of the relation $\pi, i \models \varphi$:

$$\begin{aligned}
\pi, i \models \rho \text{ where } \rho \in AP &\Leftrightarrow \rho \in \pi_i \\
\pi, i \models X\ \varphi &\Leftrightarrow \pi, i+1 \models \varphi \\
\pi, i \models \varphi\ U\ \psi &\Leftrightarrow \exists j \geq i,\ \pi, j \models \psi \text{ and } \\
&\qquad \forall k \in \{i, \ldots, j-1\}, \pi, k \models \varphi \\
\pi, i \models \varphi \vee \psi &\Leftrightarrow (\pi, i \models \varphi) \vee (\pi, i \models \psi) \\
\pi, i \models \neg\varphi &\Leftrightarrow \pi, i \not\models \varphi \\
\pi, i \models True &\quad \text{always.}
\end{aligned}$$

Intuitively, $\pi, i \models \varphi$ means that the path π verifies φ from its i-th symbol onward. We consider the language $L(\varphi) = \{w \mid w \in (2^{AP})^\omega \text{ and } w, 0 \models \varphi\}$ of an LTL formula φ, that is, the set of all paths verifying φ according to the semantics outlined previously.

For convenience's sake, we introduce the operators $F\ \varphi = (True\ U\ \varphi)$ and $G\ \varphi = \neg(F\ \neg\varphi)$ that stand respectively for *finally* and *globally.*

LTL and Büchi automata. We consider the following class of finite state automata:

Definition 5 (Büchi automaton) A *Büchi automaton* (BA) $\mathcal{B}$ is a tuple $(Q, \Sigma, \delta, q_0, G)$ where Q is a finite set of states, Σ a finite input alphabet, $\delta \subseteq Q \times \Sigma \times Q$ a set of transitions, $G \subseteq Q$ a set of accepting states, and $q_0 \in Q$ the initial state.

The language $L(\mathcal{B})$ accepted by $\mathcal{B}$ is the set of all infinite sequences w in Σ^ω such that there is an infinite run r of $\mathcal{B}$ with trace w starting in state q_0 that visits accepting states in G infinitely often.

BA can be used in the following fashion:

Theorem 2 (Kesten et al. [8]) *Given a LTL formula φ, there exists an effectively computable Büchi automaton $\mathcal{B}$ on the alphabet $\Sigma = 2^{AP}$ such that $L(\mathcal{B}) = L(\varphi)$.*

3.2 LTL model checking for PDSs

Let $\nu : Conf_{\mathcal{P}} \to 2^{AP}$ be a valuation function on configurations of a PDS $\mathcal{P} = (P, Act, \Gamma, \Delta, c_0)$. It is said to be *simple* if for all $w, w' \in \Gamma^*$, $p \in P$, and $\gamma \in \Gamma$, we have $\nu(\langle p, \gamma w\rangle) = \nu(\langle p, \gamma w'\rangle)$. Intuitively, a simple valuation is equivalent to a function $\nu : P \times \Gamma \to 2^{AP}$ that only depends on the control state and the top stack symbol.

Let $r = (r_i)_{i\geq 0}$ be an infinite run of $\mathcal{P}$. We define the image $\nu(r) = (\nu(r_i))_{i\geq 0}$ in *Paths* of r by the valuation function ν. We write that $r \models_\nu \varphi$ if $\nu(r), 0 \models \varphi$. The *model checking* problem is defined as follows:

Definition 6 (Model checking) Given an LTL formula φ, a PDS $\mathcal{P}$ with a starting configuration c_0, and a simple valuation ν on configurations of $\mathcal{P}$, the *existential* model checking problem consists in determining whether $\exists r \in Runs_\omega(\mathcal{P})$, $r \models_\nu \varphi$.

Obviously, since the negation of an LTL formula is still an LTL formula as well, determining whether $\forall r \in Runs_\omega(\mathcal{P})$, $r \models_\nu \varphi$ is an equivalent problem, called the *universal* model checking problem. Indeed:

$$\begin{aligned}
&\neg(\forall r \in Runs_\omega(\mathcal{P}), r \models_\nu \varphi) \\
\Leftrightarrow\ &(\exists r \in Runs_\omega(\mathcal{P}), r \models_\nu \neg\varphi)
\end{aligned}$$

We will now present the automata-theoretic framework introduced in [2,7] in order to solve the existential model checking problem.

Using Büchi pushdown automata. We consider the following class of automata:

Definition 7 (Büchi pushdown automata) A Büchi pushdown automaton (BPDA) is a tuple $\mathcal{BP} = (P, Act, \Gamma, \Delta, c_0, G)$ such that $(P, Act, \Gamma, \Delta, c_0)$ is a PDS and $G \subseteq P$ a set of accepting states.

In a similar manner to BA, an *accepting run* of $\mathcal{BP}$ is an infinite run of the PDS $(P, Act, \Gamma, \Delta, c_0)$ that visits infinitely often configurations whose control state is in G. To these runs, we match *accepting traces.*

A BPDA can be seen as a product automaton between a PDS and Büchi automaton. The use of this class of automata is the following:

Theorem 3 *Given a PDS $\mathcal{P}$ and an LTL formula φ, there exists an effectively computable BPDA $\mathcal{BP}$ such that t is an accepting trace of $\mathcal{BP}$ if and only if t is a trace of $\mathcal{P}$ matched to a run r such that $r \models_\nu \varphi$.*

Thus, if we can solve the emptiness problem for BPDA, then we can solve the model-checking problem of LTL for PDSs as well. Indeed, if $\mathcal{L}(\mathcal{BP}) = \emptyset$, then there is no run r such that $r. \models_\nu \varphi$.

Repeating heads and the emptiness problem. A *repeating head* of $\mathcal{BP}$ is an element $\langle p, \gamma \rangle$ of $G \times \Gamma$ such that $\exists w \in \Gamma^*$, $\langle p, \gamma \rangle \rightarrow^{+}_{\mathcal{BP}} \langle p, \gamma w \rangle$. Let $Rep(\mathcal{BP})$ be the finite set of repeating heads of $\mathcal{BP}$. The following lemma characterizes accepting runs with regards to repeating heads:

Lemma 1 *r is an accepting run of a BPDA $\mathcal{BP}$ if and only if $\mathcal{BP}$ has a repeating head $\langle p, \gamma \rangle$ such that r visits configurations in $\langle p, \gamma \Gamma^* \rangle$ infinitely often.*

Therefore, a run of a BPDA is accepting if and only if it visits a repeating head infinitely often. Moreover, the set of repeating heads is obviously finite, being a subset of $P \times \Gamma$, and can be effectively computed in $O(|P| \cdot |\Delta|^2)$ time and $O(|P| \cdot |\Delta|)$ space [7].

Let $Rep(\mathcal{BP})\Gamma^* = \{\langle p, \gamma w \rangle \mid \langle p, \gamma \rangle \in Rep(\mathcal{BP}), w \in \Gamma^*\}$ be the regular set of configurations starting with a repeating head. A BPDA will admit at least one accepting run if and only if there exists a repeating head reachable from the initial configuration, hence, the following theorem:

Theorem 4 *Given a BPDA $\mathcal{BP}$ with starting configuration c_0, its language $\mathcal{L}(\mathcal{BP})$ is not empty if and only $pre^*(Rep(\mathcal{BP})\Gamma^*) \cap \{c_0\} \neq \emptyset$.*

Since the set $pre^*(Rep(\mathcal{BP})\Gamma^*)$ is regular and effectively computable by Theorem 1, we can therefore solve the model checking problem using Theorems 3 and 4.

3.3 Single-indexed LTL model checking for CPDSs

Let $\mathcal{CP} = (\mathcal{P}_1, \ldots, \mathcal{P}_n)$ be a CPDS, ν a simple valuation function on $Conf_{\mathcal{P}_1} \cup \ldots \cup Conf_{\mathcal{P}_n}$, and for $i = 1, \ldots, n$, let ψ_i be an LTL formula. The tuple $\varphi = (\psi_1, \ldots, \psi_n)$ is said to be a *single-indexed* LTL formula.

We define the following semantics for single-indexed LTL formula on CPDSs:

Definition 8 (Single-indexed LTL semantics) Given a CPDS $\mathcal{CP} = (\mathcal{P}_1, \ldots, \mathcal{P}_n)$, an infinite global run g of $\mathcal{CP}$, a simple valuation on pushdown components ν, and a single-indexed LTL formula $\varphi = (\psi_1, \ldots, \psi_n)$, $g \models_\nu \varphi$ if and only if for each $i = 1, \ldots, n$, $g^i \models_\nu \psi_i$. Finding such a global run g is called the *existential single-indexed model checking* problem.

Intuitively, each PDS $\mathcal{P}_i$ in the CPDS satisfies its own LTL formula ψ_i, but does so while synchronizing with the others PDSs. This is a simpler problem than solving the existential model checking for a single LTL formula and a valuation function on global configurations: we will therefore try to solve it first, then focus later on the general case.

If the single-indexed model checking problem for CPDSs were decidable, so would be the reachability problem. However, it has unfortunately been proven by Ramalingam in [16] that this problem is undecidable for synchronization-sensitive pushdown systems. Hence, since the latter reachability problem is obviously undecidable, the former model checking problem is as well.

We therefore seek to get at least an approximate answer to this problem. The issue with single-indexed model checking is the following: for each $i = 1, \ldots, n$, there may exist a run r_i of the PDS $\mathcal{P}_i$ satisfying a formula ψ_i, but a global, synchronized run on the CPDS $(\mathcal{P}_1, \ldots, \mathcal{P}_n)$ satisfying $\varphi = (\psi_1, \ldots, \psi_n)$ may not exist if the individual pushdown components of the system can't synchronize. We will tackle this issue in the next section.

4 An Abstraction Framework for Traces

We want to abstract the single-indexed LTL model checking problem for CPDSs. To this end, we seek to over-approximate the set

$$\mathcal{L}_{\mathcal{P}}(C', C) = \{t \in Traces(\mathcal{P}) | \exists c \in C, \exists c' \in C', c' \xrightarrow{t}^{*}_{\mathcal{P}} c\}$$

of traces of a PDS $\mathcal{P}$ leading from a regular set of configurations C' to another C. We will use the mathematical framework introduced by Bouajjani et al. in [3] to approximate the reachability problem for CPDSs. More details can be found in the appendix.

To this end, they consider a *Kleene algebra* $K = (A, \oplus, \odot, \overline{0}, \overline{1})$ matched to an *abstract lattice* $E = (A, \leq, \sqcup, \sqcap, \bot, \top)$ and a *Galois connection* that consists in an *abstraction* function $\alpha : Act^* \to K$ and a *concretization* function $\beta : K \to Act^*$ such that, given two languages L_1 and L_2 on the alphabet Act:

$$\begin{aligned} L_1 &\subseteq \beta(\alpha(L_1)) \\ L_2 &\subseteq \beta(\alpha(L_2)) \\ \alpha(L_1) \sqcap \alpha(L_2) = \bot &\Leftrightarrow \beta(\alpha(L_1)) \cap \beta(\alpha(L_2)) = \emptyset \end{aligned}$$

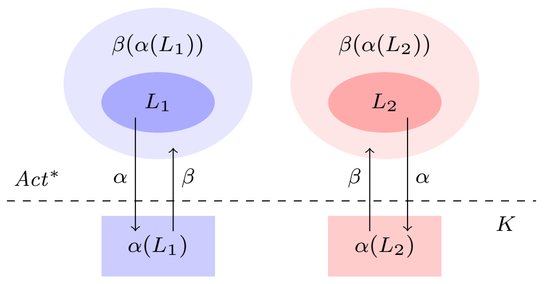

Fig. 1: From Act^* to the abstract domain K and back again.

Hence, as shown in Figure 1, the Galois connection can be used to over-approximate a language in 2^{Act^*} (e.g. the trace language of a PDS), and we can directly check the emptiness of the intersection (a common operation as far as model checking goes) of these approximations in a simpler, often finite domain. Here are two examples of such finite-domain *Kleene abstractions*:

Example 3 (The prefix abstraction) Let $n \geq 1$ be an integer and $W(n) = \{w \in Act^* \mid |w| \leq n\}$ be the set of words of length smaller than n.

We define the n-th order *prefix* abstraction α_n^{pref} as follows:

- The abstract lattice $A = 2^W$ is generated by the elements $v_a = \{a\}$, $a \in Act$.
- $\oplus = \cup$.
- $U \odot V = \{pref_n(uv) \mid u \in U, v \in V\}$ where $pref_n(w)$ stands for the prefix of w of length n (or lower if w is of length smaller than n).
- $\overline{0} = \emptyset$ and $\overline{1} = \{\varepsilon\}$.

From there, we build an abstract lattice where $\top = W$, $\sqcap = \cap$, and $\leq = \subseteq$.

This abstraction is accurate for the n-th first steps of a trace: the other steps are then over-approximated by Act^* for a finite trace or Act^ω for an infinite trace. Hence, if $w \in Act^\omega$, then the prefix abstraction of $\{w\}$ is

$$\alpha_n^{pref}(\{w\}) = \{pref_n(w)\}$$

its matching connection

$$\beta_n^{pref}(\{p\}) = p \cdot Act^*$$

and the resulting over-approximation the language

$$\beta_n^{pref}(\alpha_n^{pref}(\{w\})) = pref_n(w) \cdot Act^*$$

Example 4 (First occurrence ordering) Let W be the set of words

$$\{w \in Act^* \mid \forall a \in Act, |w|_a \leq 1\}$$

where each letter occurs at most once ($|w|_a$ is the number of a in the word w).

We define the *first occurrence ordering* abstraction α^{foo} as follows:

- The abstract lattice $A = 2^W$ is generated by the elements $v_a = \{a\}$, $a \in Act$
- $\oplus = \cup$.
- $U \odot V = \{uv' \mid u \in U$ and $\exists v \in V, v'$ is the projection of v on letters not occuring in $u\}$.
- $\overline{0} = \emptyset$ and $\overline{1} = \{\varepsilon\}$.

From there, we build an abstract lattice where $\top = W$, $\sqcap = \cap$, and $\leq = \subseteq$.

If w is a trace in Act^*, its first occurrence ordering is

$$\alpha^{foo}(\{w\}) = a_1 \dots a_n$$

where the set of letters in w is exactly $\{a_1, \dots, a_n\}$ and they first appear in the order $a_1, \dots, a_n$. Its matching connection is

$$\beta_{foo}(a_1 \cdot \dots \cdot a_n) = a_1 a_1^* a_2 (a_1 + a_2)^* \dots a_n (a_1 + \dots + a_n)^*$$

The resulting over-approximation is

$$\beta_{foo}(\alpha^{foo}(\{w\})) = a_1 a_1^* a_2 (a_1 + a_2)^* \dots a_n (a_1 + \dots + a_n)^*$$

Example 5 Let $w = aacbabed(ab)^\omega$. Then:

$$\begin{aligned} \alpha_5^{pref}(\{w\}) &= aacbabed(ab)^\omega \\ \alpha^{foo}(\{w\}) &= aacbabed(ab)^\omega \end{aligned}$$

A *finite-chain* abstraction is such that the lattice $(A, \leq)$ has no infinite ascending chains: as a consequence, infinite sequences of concatenations can be defined and effectively computed, and the abstraction framework can be applied to countably infinite traces. The prefix and first occurrence ordering abstractions defined previously are finite-chain.

5 Abstract Model Checking of LTL for CPDSs

In this section, as a main contribution of this paper, we will introduce an approximation framework for single-indexed LTL model checking on CPDSs.

5.1 Abstracting accepting traces of a BPDA

We show in this subsection how to over-approximate the set of accepting traces of BPDA. To do so, we extend the abstract reachability analysis introduced in [3] to infinite runs.

Properties of accepting runs of BPDA. By Lemma 1, to each accepting run r of a BPDA $\mathcal{BP}$, we can match a repeating head $\langle p, \gamma \rangle \in Rep(\mathcal{BP})$ that it visits infinitely often. Conversely, if a run r of a BPDA $\mathcal{BP}$ visits a repeating head $\langle p, \gamma \rangle \in Rep(\mathcal{BP})$ infinitely often, it is then obviously accepting since $Rep(\mathcal{BP}) \subseteq G \times \Gamma$. We will use the former property to abstract the set of accepting traces.

Let $\langle p, \gamma \rangle \in Rep(\mathcal{BP})$ be a repeating head. An accepting trace visiting the set $\langle p, \gamma\Gamma^* \rangle$ infinitely often can be split into two parts:

(1) first, it must reach the set $\langle p, \gamma\Gamma^* \rangle$ from the initial configuration c_0;
(2) then, it must infinitely often move from $\langle p, \gamma\Gamma^* \rangle$ to $\langle p, \gamma\Gamma^* \rangle$, using a sequence of transitions of length superior or equal to one.

Part (1) of an accepting trace visiting $\langle p, \gamma \rangle$ must therefore be in the set $\mathcal{L}_{\mathcal{BP}}(\{c_0\}, \langle p, \gamma\Gamma^* \rangle)$, and part (2), in the set $[\mathcal{L}_{\mathcal{BP}}(\langle p, \gamma\Gamma^* \rangle, \langle p, \gamma\Gamma^* \rangle)/\{\varepsilon\}]^\omega$.

As a consequence, the set

$$\mathcal{L}_{\mathcal{BP}}(\{c_0\}, \langle p, \gamma\Gamma^* \rangle) \cdot (\mathcal{L}_{\mathcal{BP}}(\langle p, \gamma\Gamma^* \rangle, \langle p, \gamma\Gamma^* \rangle)/\{\varepsilon\})^\omega$$

is an over-approximation of the set of accepting traces visiting the repeating head $\langle p, \gamma \rangle$ infinitely often. Since it cannot be computed directly, we will apply the framework of Section 4 to abstract it.

Abstracting the set of accepting traces. Our method to abstract the set of accepting traces of a BPDA $\mathcal{BP}$ is the following:

1. Compute its finite set of repeating heads $Rep(\mathcal{BP})$.
2. For each repeating head $\langle p, \gamma \rangle \in Rep(\mathcal{BP})$, find if it is reachable from the initial configuration c_0 by checking that

$$pre^*(\langle p, \gamma\Gamma^* \rangle) \cap \{c_0\} \neq \emptyset$$

This way, we can compute the set $Rep^+(\mathcal{BP})$ of reachable repeating heads.

$$\begin{tikzcd} c_0 \arrow[r, "I_{\langle p,\gamma\rangle}"] & \langle p, \gamma\Gamma^* \rangle \arrow[loop right, "R_{\langle p,\gamma\rangle}"] \end{tikzcd}$$

Fig. 2: Abstracting the infinite set of traces matched to the repeating head $\langle p, \gamma \rangle$

3. For each reachable repeating head $\langle p, \gamma \rangle$ in the set $Rep^+(\mathcal{BP})$, compute an abstraction

$$I_{\langle p,\gamma\rangle} = \alpha(\mathcal{L}_{\mathcal{BP}}(\{c_0\}, \langle p, \gamma\Gamma^* \rangle))$$

of the set of traces leading from the initial configuration to the first occurrence of the repeating head; this yields part (1) of the accepting traces.
4. Compute an abstraction

$$R_{\langle p,\gamma\rangle} = \alpha(\mathcal{L}_{\mathcal{BP}}(\langle p, \gamma\Gamma^* \rangle, \langle p, \gamma\Gamma^* \rangle))/\{\bar{1}\}$$

of the set of non-trivial traces between two occurrences of the repeating head; this yields part (2) of the accepting traces. Note that $(R_{\langle p,\gamma\rangle})^\omega$ is finite and can be computed as well, because the abstract domain is finite-chain.
5. Compute a finite-chain abstraction

$$T_{\langle p,\gamma\rangle} = I_{\langle p,\gamma\rangle} \odot (R_{\langle p,\gamma\rangle})^\omega$$

of the set of accepting traces visiting the repeating head $\langle p, \gamma \rangle$ infinitely often, as shown in Figure 2.
6. Consider the finite sum

$$T = \bigoplus_{\langle p,\gamma\rangle \in Rep^+(\mathcal{BP})} T_{\langle p,\gamma\rangle}$$

as an abstraction in the finite-chain domain of the set of all accepting traces of $\mathcal{BP}$.

As a consequence, the following theorem holds:

Theorem 5 *Given a BPDA $\mathcal{BP}$, we can compute a finite-chain abstraction of its set of accepting traces $\mathcal{L}(\mathcal{BP})$.*

5.2 Abstracting the single-indexed model checking problem for CPDSs

Let $\mathcal{CP} = (\mathcal{P}_1, \ldots, \mathcal{P}_n)$ be a CPDS and $\varphi = (\psi_1, \ldots, \psi_n)$ a single-indexed LTL formula. Our goal is to compute an over-approximation of the undecidable single-indexed LTL model checking problem, i.e. compute an over-approximation of the set of runs of $\mathcal{CP}$ verifying φ; if this over-approximation is empty, so is the actual set of runs verifying φ, and the answer to the existential single-indexed model checking problem is negative.

Our intuition is, for each component $\mathcal{P}_i$, to abstract the set of paths verifying ψ_i, then examine the emptiness of the intersection of these abstractions. However, in a global run of a CPDS, the execution paths of the pushdown components are interleaved. Synchronization signals between two threads j and k may occur in the global run but will not appear in local runs of the i-th pushdown component. We cannot therefore study the paths of a pushdown system $\mathcal{P}_i$ in isolation from the other components.

Projecting global runs. For each pushdown component $\mathcal{P}_i = (P_i, Lab_i \cup \{\tau\}, \Gamma, \Delta_i, c_0^i)$ of a CPDS $\mathcal{CP} = (\mathcal{P}_1, \ldots, \mathcal{P}_n)$, we introduce a new PDS $\mathcal{P}'_i = (P_i, Act, \Gamma, \Delta'_i, c_0^i)$ such that:

- $\Delta_i \subseteq \Delta'_i$; the new PDS $\mathcal{P}'_i$ extends the pushdown component $\mathcal{P}_i$.
- For all $p \in P_i$, $\gamma \in \Gamma$, and $x \in \{\tau\} \cup \bigcup_{j \neq i \neq k} Lab_{j,k}$, $(p, \gamma) \xrightarrow{x} (p, \gamma) \in \Delta'_i$. We add to each control state self-loops labelled either by the internal action τ or by every synchronization signal in $Lab_{j,k}$ between pair of different processes j and k for all $j \neq i \neq k$.

The PDS $\mathcal{P}'_i$ can either simulate $\mathcal{P}_i$ or self-loop and output any possible signal emitted by other processes, be it an internal action or a synchronization between two other processes.

Given two regular sets of configurations C' and C of $\mathcal{P}_i$, and the language $\mathcal{L}_{\mathcal{P}_i}(C', C)$ of traces leading from C' to C, we have

$$\mathcal{L}_{\mathcal{P}'_i}(C', C) = \mathcal{L}_{\mathcal{P}_i}(C', C) \sqcup\!\sqcup (\{\tau\} \cup \bigcup_{j \neq i \neq k} Lab_{j,k})^*$$

where $\sqcup\!\sqcup$ is the interleaving operator: we shuffle the paths of $\mathcal{P}_i$ with internal and synchronization actions of other threads.

As a consequence, it allows us to over-approximate projections of the set of global runs:

Lemma 2 *Let g be a global run of the CPDS $\mathcal{CP}$. Then g^i is a run of $\mathcal{P}'_i$.*

Intuitively, we can prove recursively that, at every step of g^i, we can either use an existing transition of $\mathcal{P}_i$ if this step of the global run g involved the i-th component, or simulate any action performed by other threads with one of $\mathcal{P}'_i$'s own self-loops.

Application to the model checking problem. Let g be a global run of $\mathcal{CP}$ and $\varphi = (\psi_1, \ldots, \psi_n)$ a single-indexed LTL formula. Then $g \models_\nu \varphi$ if and only if for each $i = 1, \ldots, n$, $g^i \models_\nu \psi_i$.

By Lemma 2, if $\forall r \in Runs_\omega(\mathcal{P}'_i)$, $r \not\models_\nu \psi_i$, then $g^i \not\models_\nu \psi_i$. By Theorem 3, there exists a BPDA $\mathcal{BP}_i$ such that $\mathcal{L}(\mathcal{BP}_i)$ is the set of all traces of runs of $\mathcal{P}'_i$ verifying ψ_i. $\mathcal{L}(\mathcal{BP}_i)$ is empty if and only if $\forall r \in Runs_\omega(\mathcal{P}'_i)$, $r \not\models \psi_i$.

The projections $g^1, \ldots, g^n$ of a global run g have the same trace as the original run g. As a consequence, even if we can find in the automata $\mathcal{P}'_1, \ldots, \mathcal{P}'_n$ runs accepting the LTL formulas $\psi_1, \ldots, \psi_n$, they can't be projections of a same accepting global run if they don't share the same trace.

We can use this property as a discriminating criterium for the existential single-indexed LTL model checking problem: we compute an abstraction of the set of accepting traces $\alpha(\mathcal{L}(\mathcal{BP}_i))$ of each BPDA $\mathcal{BP}_i$ then check if these abstractions share a common trace.

Theorem 6 *If $\alpha(\mathcal{L}(\mathcal{BP}_1)) \sqcap \ldots \sqcap \alpha(\mathcal{L}(\mathcal{BP}_n)) = \bot$, then there is no global run of $\mathcal{CP}$ accepting the single-indexed LTL formula φ.*

Proof (Theorem 6) Let us assume that there exists a global run g of $\mathcal{CP}$ such that $g \models \varphi$. Let g^i be its projection on the component $\mathcal{P}_i$. g^i is a run of $\mathcal{P}'{}_i$ by Lemma 2, hence, its trace t^i should belong to the over-approximation $\beta(\alpha(\mathcal{L}(\mathcal{BP}_i))$ of the set of traces of $\mathcal{BP}_i$.

However, the projections $g^1, \ldots, g^n$ of g all have the same trace as g, hence, $t^1 = \ldots = t^n = t$. As a consequence

$$t \in \beta(\alpha(\mathcal{L}(\mathcal{BP}_1)) \cap \ldots \cap \beta(\alpha(\mathcal{L}(\mathcal{BP}_n))$$

This set is therefore not empty, hence, by definition of Galois connections,

$$\alpha(\mathcal{L}(\mathcal{BP}_1)) \sqcap \ldots \sqcap \alpha(\mathcal{L}(\mathcal{BP}_n)) \neq \bot$$

and there is a contradiction. □

We can decide in a finite-chain domain if

$$\alpha(\mathcal{L}(\mathcal{BP}_1)) \sqcap \ldots \sqcap \alpha(\mathcal{L}(\mathcal{BP}_n)) \neq \bot$$

Hence, Theorem 6 can be used to approximate the existential single-indexed model-checking problem.

5.3 The universal single-indexed model checking problem

We now try to apply our abstraction framework to the universal model checking problem:

Definition 9 (Universal model checking) Given a CPDS $\mathcal{CP} = (\mathcal{P}_1, \ldots, \mathcal{P}_n)$ of $\mathcal{CP}$, a simple valuation on pushdown components ν, and a single-indexed LTL formula φ, $\mathcal{CP} \models_\nu \varphi$ if and only if for each global run g of $\mathcal{CP}$, $g \models_\nu \varphi$. Determining whether this property hold is called the *universal single-indexed model checking* problem.

Unlike LTL formulas, we don't know if the negation of a single-indexed LTL formula is also a single-indexed LTL formula. However, the following property holds:

$$\begin{aligned}
&\neg(\mathcal{CP} \models_\nu \varphi)\\
&= \neg(\forall g \in Runs_\omega(\mathcal{CP}), \forall i \in \{1,\dots,n\}), g_i \models_\nu \psi_i)\\
&= \exists g \in Runs_\omega(\mathcal{CP}), \exists i \in \{1,\dots,n\}, g_i \not\models_\nu \psi_i\\
&= \exists i \in \{1,\dots,n\}, \exists g \in Runs_\omega(\mathcal{CP}), g_i \models_\nu \neg\psi_i\\
&= \bigvee_{i\in\{1,\dots,n\}} \exists g \in Runs_\omega(\mathcal{CP}), g_i \models_\nu \neg\psi_i\\
&= \bigvee_{i\in\{1,\dots,n\}} \exists g \in Runs_\omega(\mathcal{CP}), g \models_\nu \varphi_i
\end{aligned}$$

Where φ_i is a single-indexed LTL formula such that its i-th component is the LTL formula $\neg\psi_i$ and all the others are $True$.

As a consequence, at least one single-indexed existential model checking property "$\exists g \in Runs_\omega(\mathcal{CP})$, $g \models_\nu \varphi_i$" holds true if and only if $\mathcal{CP} \not\models_\nu \varphi$. Conversely, all the existential properties are wrong if and only if $\mathcal{CP} \models_\nu \varphi$.

5.4 A summary of our abstraction procedures

In order to approximate the existential single-indexed model checking problem

"Given a CPDS $\mathcal{CP} = (\mathcal{P}_1,\dots,\mathcal{P}_n)$, a single-indexed LTL formula $\varphi = (\psi_1,\dots,\psi_n)$, and a simple valuation on pushdown components ν, is there a global run g of $\mathcal{CP}$ such that $g \models_\nu \varphi$?"

we apply the following procedure:

1. For each component $\mathcal{P}_i$, compute the altered PDS $\mathcal{P}'_i$.
2. Then compute the product BPDA $\mathcal{BP}_i$ of $\mathcal{P}'_i$ and ψ_i according to Theorem 3
3. Compute the abstraction $\alpha(\mathcal{L}(\mathcal{BP}_i))$.
4. Check that $\alpha(\mathcal{L}(\mathcal{BP}_1)) \sqcap \ldots \sqcap \alpha(\mathcal{L}(\mathcal{BP}_n)) = \bot$.
5. If this is true, then by Theorem 6, the answer to the existential single-indexed model checking problem is negative.
6. However, if this is not the case, we can't draw any immediate conclusion. We may use the abstract counter-example to help us find an actual run of $\mathcal{CP}$ that verifies φ. We may also refine the Kleene abstraction by increasing its order if it's a prefix abstraction.

Note that, as an optimization, we can perform an emptiness check on the BPDA $\mathcal{BP}_i$ before abstracting its language $\alpha(\mathcal{L}(\mathcal{BP}_i))$. If $\mathcal{L}(\mathcal{BP}_i) = \emptyset$, then there is obviously no global run g such that $g_i \models_\nu \psi_i$ and the answer to the existential problem is negative. By doing so, we avoid computing the abstraction $\alpha(\mathcal{L}(\mathcal{BP}_i))$ that may lead to spurious counter-examples if it were not empty.

However, if we want to solve the universal single-indexed model checking problem

"Is it true that if g is a global run of $\mathcal{CP}$ then $g \models_\nu \varphi$?"

we apply instead this procedure:

1. For each component ψ_i of φ, approximate the existential model checking of φ_i (as defined in subsection 5.3)) on $\mathcal{CP}$, using the procedure described above.
2. If, for every sub-problem

   "$\exists g \in Runs_\omega(\mathcal{CP}), g \models_\nu \varphi_i$"

   the procedure returns a negative answer, then the answer to the universal single-indexed model checking of φ is positive.
3. However, if this is not the case, we can't draw any immediate conclusion. We may use the abstract counter-example to help us find an actual run of $\mathcal{CP}$ that does not verify φ. We may also refine the Kleene abstraction by increasing its order if it's a prefix abstraction.

We can apply this abstraction framework to *safety* - ensuring that an unsafe state is never reached - or *liveness* - checking that a desirable state can always be eventually reached - properties. By design, our procedure is better at providing a positive answer to these problems, as a definitive negative answer would require us to compute an infinite run as a counter-example, something we are not yet capable of (even as a semi-decision procedure). It is unlikely such a semi-decision procedure is even possible, as the actual set we are approximating is an intersection of ω-context free languages.

6 Model Checking Stutter-invariant LTL Formulas on CPDSs

In the previous section, we focused on single-indexed LTL model checking for CPDSs. The valuation function was applied on a thread per thread basis: only one configuration in the set of configurations of pushdown components was evaluated at a time. It can be argued that such a definition is not natural, as local properties are applied to projections of a global run.

Hence, we should consider the LTL model checking problem on CPDSs where the valuation function takes a global configuration in the set $Conf_{\mathcal{P}_1} \times \ldots \times Conf_{\mathcal{P}_n}$ as an input. This problem is obviously undecidable, but we show that it can be abstracted for the sub-class of *stutter-invariant* LTL formulas.

6.1 Generic model checking for CPDSs

Given a CPDS $\mathcal{CP} = (\mathcal{P}_1, \ldots, \mathcal{P}_n)$, let $\nu : Conf_{\mathcal{CP}} \to 2^{AP}$ be a valuation function on global configurations. It is said to be *simple* if

$$\forall w_1, \ldots, w_n, w'_1, \ldots, w'_n \in \Gamma^*$$
$$\forall p_1 \in P_1, \ldots, p_n \in P_n, \forall \gamma_1, \ldots, \gamma_n \in \Gamma,$$

we have

$$\nu(\langle p_1, \gamma_1 w_1\rangle, \ldots, \langle p_n, \gamma_n w_n\rangle)$$
$$= \nu(\langle p_1, \gamma_1 w'_1\rangle, \ldots, \langle p_n, \gamma_n w'_n\rangle)$$

Intuitively, a simple valuation on global configurations is equivalent to a function

$$\nu : (P_1 \times \Gamma) \cup \ldots \cup (P_n \times \Gamma) \to 2^{AP}$$

that only depends on the control state and the top stack symbol of each component. To each global run g of $\mathcal{CP}$, we can match a path $\nu(g) \in Paths$.

We will now consider the general model checking problem, similar to the simpler PDS case:

Definition 10 (LTL semantics) Given a CPDS $\mathcal{CP}$, an infinite global run g of $\mathcal{CP}$, a LTL formula φ, and a simple valuation on global configurations ν, $g \models_\nu \varphi$ if and only if $\nu(g) \models \varphi$. Finding such a global run g is called the existential model checking problem.

This problem is obviously undecidable, being more general than the already undecidable reachability problem.

6.2 Stutter-invariant LTL formulas

Intuitively, stutter-invariant formulas are such that adding or removing repetitions in a path does not change its evaluation for such formulas. Hence, sequences that differ only in the amount of stuttering can be considered equivalent when checking stutter-invariant properties.

Definition 11 (Stutter-invariance) A LTL formula φ is said to be *stutter-invariant* if, $\forall (\pi_i)_{\geq 0} \in Paths$ and $\forall (n_i)_{i \geq 0} \in (\mathbb{N}^*)^{\mathbb{N}}$,

$$\pi_0^{n_0} \pi_1^{n_1} \ldots \models \varphi \text{ if and only if } \pi \models \varphi$$

where $\pi_0^{n_0} \pi_1^{n_1} \ldots$ stands for the infinite sequence where the i-th value of π is repeated n_i times. The paths $\pi_0^{n_0} \pi_1^{n_1} \ldots$ and π are said to be *stutter-equivalent.*

As proven by Peled et al. in [10], LTL formulas without the operator *next* (X) are stutter-invariant and, conversely, any stutter-invariant LTL property is equivalent to an LTL $\setminus$ (X) formula. The *next* operator is seldom used in concurrent model checking: hence, abstracting the stutter-invariant LTL model checking problem for CPDSs would be a worthy addition to existing verification techniques.

Note that the set of stutter-invariant formulas is stable by negation. As a consequence, the universal and existential stutter-invariant LTL model checking problems are equivalent:

$$(\forall r, r \models \varphi) \Leftrightarrow \neg(\exists r, r \models \neg\varphi)$$

6.3 Abstract model checking for stutter-invariant LTL

In this sub-section, we show how our abstraction framework for single-indexed LTL formulas can be applied to stutter-invariant model checking:

Theorem 7 *Let $\mathcal{CP}$ be a CPDS, ν a simple valuation on global configurations, and φ, a stutter-invariant LTL formula. Then there exists a CPDS $\mathcal{CP}'$, a simple valuation on configurations of pushdown components ν', and a single-indexed formula ψ such that there exists a global run g of $\mathcal{CP}$ verifying $g \models_\nu \varphi$ if and only if there exists a global run g' of $\mathcal{CP}'$ verifying $g' \models_{\nu'} \psi$.*

As a consequence, if there is no global run g' of $\mathcal{CP}'$ such that $g' \models_{\nu'} \psi$, then there is no global run of $\mathcal{CP}$ such that $g \models_\nu \varphi$. Our abstraction framework on $\mathcal{CP}'$ can provide us with a negative answer to the existential stutter-invariant model checking problem (or, similarly, with a positive answer to the universal problem).

Reduction to the single-indexed case. Intuitively, we will add to the pushdown components of $\mathcal{CP}$ a new PDS called the controller, whose purpose will be to store in its control state the current states and stack symbols of the other components so that it can be used as an input for the simple valuation function ν. To do so, we will modify the pushdown components so that after each transition they must send a status update to the controller.

As a consequence, the valuations of the actual global configurations of the CPDS $\mathcal{CP}$ and of the top configurations stored in the controller's memory will be similar, although the extra transitions added by the synchronization process between the controller and the other pushdown components will cause some stuttering. A global run of $\mathcal{CP}$ can therefore be used to design a new global run in the modified CPDS such that its projection on the controller will be equivalent as far as the verification of stutter-invariant properties is concerned.

Proof (Theorem 7) Without loss of generality, we assume that the control states sets $(P_i)_{i=1\ldots n}$ are disjoint, that each pushdown component $\mathcal{P}_i$ has a bottom stack symbol $\perp$ in Γ that can never be popped and a starting configuration of the form $c_0^i = \langle s_0^i, \perp \rangle$.

Modifying the pushdown components. For each pushdown component $\mathcal{P}_i = (P_i, \{\tau\} \cup Lab_i, \Gamma, \Delta_i, c_0^i)$, we introduce a new PDS $\mathcal{P}'_i = (P'_i, \Sigma_i, \Gamma, \Delta'_i, c_0^i)$ such that:

- Let P_i^j, $j \in \{1, \ldots, n\}$ be disjoint copies of P_i. Then
$$P'_i = P_i \cup P_i^1 \cup \ldots \cup P_i^n$$
To each state $p \in P$, we match a state $p^j \in P_i^j$.
- We add new synchronization signals between the controller and the pushdown components:
$$\Sigma_i = \{\tau\} \cup Lab_i \cup (P_i \times \Gamma \times \{1, \ldots, n\})$$
- $\forall (p, \gamma) \xrightarrow{a} (q, w) \in \Delta_i$ such that $a \in Lab_{i,j}$, we have
$$(p, \gamma) \xrightarrow{a} (q^j, w) \in \Delta'_i$$
Unlike Δ_i, instead of moving straight to q, the PDS goes through an intermediate state q^j.
- $\forall (p, \gamma) \xrightarrow{\tau} (q, w) \in \Delta_i$ we have
$$(p, \gamma) \xrightarrow{\tau} (q^i, w) \in \Delta'_i$$
The PDS goes through an intermediate state q^i instead of moving straight to q.
- $\forall p \in P_i$ and $\gamma \in \Gamma$, $\forall j \in \{1, \ldots, n\}$, we have
$$(p^j, \gamma) \xrightarrow{(p,\gamma,j)} (p, \gamma) \in \Delta'_i$$
When the PDS is in an intermediate state p^j, it moves to an actual state $p \in P_i$ and sends to the controller $\mathcal{C}$ a signal stating its current state, its top stack symbol (there is always one since we can't pop the bottom of the stack), and the index of the pushdown component it previously synchronized with (i stands for internal transitions of $\mathcal{P}_i$) so that the controller can update its configuration.

Designing the controller. We define a new pushdown component $\mathcal{C} = (C, \Sigma, \Gamma, \Delta_c)$ called the *controller* such that:

- Let the set of control states be
$$C = (P_1 \times \Gamma) \times (P_2 \times \Gamma) \times \ldots \times (P_n \times \Gamma) \times Buffer$$
where
$$Buffer = ((P_1 \cup \ldots \cup P_n) \times \Gamma \times \{1, ..., n\}) \cup \{Empty\}$$
The set of control states stores the state and top stack symbols of the n other components, but also features a buffer that can either store a state and top stack symbol of a pushdown component or wait in an *Empty* state.
- Let the input alphabet be
$$\Sigma = (P_1 \cup \ldots \cup P_n) \times \Gamma \times \{1, \ldots, n\}$$
The input alphabet can be used to receive the state and top stack symbols of another pushdown component, as well as the index of the pushdown component it last synchronized with (an internal transition is considered as a synchronization of a component $\mathcal{P}'_i$ with itself).
- For each pushdown component $\mathcal{P}_i$,
$$\forall p_1 \in P_1, \ldots, \forall p_n \in P_n, \forall q_i \in P_i, \\ \forall \gamma_1, \ldots, \gamma_n, \gamma'_i \in \Gamma$$
we add to Δ_c the transition
$$((\langle p_1, \gamma_1 \rangle, \ldots, \langle p_i, \gamma_i \rangle, \ldots, \langle p_n, \gamma_n \rangle, Empty), \perp) \\ \xrightarrow{(q_i, \gamma'_i, i)} \\ ((\langle p_1, \gamma_1 \rangle, \ldots, \langle q_i, \gamma'_i \rangle, \ldots, \langle p_n, \gamma_n \rangle, Empty), \perp)$$
If the buffer is empty and the controller receives a signal triggered by an internal transition of a pushdown component, then it can directly update its state according to the information sent.
- For each pushdown component $\mathcal{P}_i$, $\forall j \neq i$,
$$\forall p_1 \in P_1, \ldots, \forall p_n \in P_n, \forall q_i \in P_i, \\ \forall \gamma_1, \ldots, \gamma_n, \gamma'_i \in \Gamma$$
we add to Δ_c the transition
$$((\langle p_1, \gamma_1 \rangle, \ldots, \langle p_i, \gamma_i \rangle, \ldots, \langle p_n, \gamma_n \rangle, Empty), \perp) \\ \xrightarrow{(q_i, \gamma'_i, j)} \\ ((\langle p_1, \gamma_1 \rangle, \ldots, \langle p_i, \gamma_i \rangle, \ldots, \langle p_n, \gamma_n \rangle, \langle q_i, \gamma_i, j \rangle), \perp)$$
If the buffer is empty and the controller receives a signal triggered by a synchronized transition of a pushdown component, then the data sent is stored in the buffer and the controller waits for the signal of the other component that took part in the synchronized transition.
- For each pushdown component $\mathcal{P}_i$, $\forall j \neq i$,
$$\forall p_1 \in P_1, \ldots, \forall p_n \in P_n, \forall q_i \in P_i, \forall q_j \in P_j, \\ \forall \gamma_1, \ldots, \gamma_n, \gamma'_i, \gamma'_j \in \Gamma$$
we add to Δ_c the transition
$$((\langle p_1, \gamma_1 \rangle, \ldots, \langle p_i, \gamma_i \rangle, \ldots, \langle p_j, \gamma_j \rangle, \ldots, \langle p_n, \gamma_n \rangle, \\ \langle q_j, \gamma'_j, i \rangle), \perp) \\ \xrightarrow{(q_i, \gamma'_i, j)} \\ ((\langle p_1, \gamma_1 \rangle, \ldots, \langle q_i, \gamma'_i \rangle, \ldots, \langle q_j, \gamma'_j \rangle, \ldots, \langle p_n, \gamma_n \rangle, \\ Empty), \perp)$$
If the buffer is full, then the controller updates its states according to the information sent by the expected pushdown component as well as the data stored in the buffer.

We introduce a buffer in order to handle synchronized transitions between two pushdown components: if, in a global run, a synchronized transition is such that a CPDS moves from configuration

$$(\langle p_1, \gamma_1 w_1\rangle, \langle p_2, \gamma_2 w_2\rangle)$$

to

$$(\langle p_1', \gamma_1' w_1\rangle, \langle p_2', \gamma_2' w_2\rangle)$$

then the representation of the pushdown components in the controller should move from

$$(\langle p_1, \gamma_1\rangle, \langle p_2, \gamma_2\rangle)$$

to

$$(\langle p_1', \gamma_1'\rangle, \langle p_2', \gamma_2'\rangle)$$

without going through the partially updated configuration

$$(\langle p_1', \gamma_1'\rangle, \langle p_2, \gamma_2\rangle)$$

that is never reached by the original CPDS. To avoid this issue, the first synchronization signal will be stored in the buffer, then both configurations are updated at the same time, using the buffer and the next synchronization signal.

Example 6 (Handling internal transitions) The run r_i

$$\langle p_i, \gamma_i w\rangle \xrightarrow{\tau} \langle q_i^i, \gamma_i' w\rangle \xrightarrow{(q_i, \gamma_i', i)} \langle q_i, \gamma_i' w\rangle$$

of the pushdown component $\mathcal{P}_i'$ can synchronize with the run r

$$(\ldots, \langle p_i, \gamma_i\rangle, \ldots, \mathit{Empty}) \xrightarrow{(q_i, \gamma_i', i)} (\ldots, \langle q_i, \gamma_i'\rangle, \ldots, \mathit{Empty})$$

of the controller.

Example 7 (Handling synchronized transitions) Consider the run r_i

$$\langle p_i, \gamma_i w_i\rangle \xrightarrow{a} \langle q_i^j, \gamma_i' w_i\rangle \xrightarrow{(q_i, \gamma_i', j)} \langle q_i, \gamma_i' w_i\rangle$$

and the run r_j

$$\langle p_j, \gamma_j w_j\rangle \xrightarrow{a} \langle q_j^i, \gamma_j' w_j\rangle \xrightarrow{(q_j, \gamma_j', i)} \langle q_j, \gamma_j' w_j\rangle$$

of the pushdown components $\mathcal{P}_i'$ and $\mathcal{P}_j'$ that can synchronize with the run r

$$(\ldots, \langle p_i, \gamma_i\rangle, \ldots, \langle p_j, \gamma_j\rangle, \ldots, \mathit{Empty})$$
$$\xrightarrow{(q_i, \gamma_i', j)}$$
$$(\ldots, \langle p_i, \gamma_i\rangle, \ldots, \langle p_j, \gamma_j\rangle, \ldots, \langle q_i, \gamma_i', j\rangle)$$
$$\xrightarrow{(q_j, \gamma_j', i)}$$
$$(\ldots, \langle q_i, \gamma_i'\rangle, \ldots, \langle q_j, \gamma_j'\rangle, \ldots, \mathit{Empty})$$

of the controller.

The valuation function. We define a new simple valuation function ν' for the pushdown component on the set

$$(P_1 \times \Gamma) \cup \ldots \cup (P_n \times \Gamma) \cup ((P_1 \times \Gamma) \times \ldots \times (P_n \times \Gamma) \times \mathit{Buffer})$$

- $\nu'((x, y)) = \nu(x)$ if

  $$x \in (P_1 \times \Gamma) \times (P_2 \times \Gamma) \times \ldots \times (P_n \times \Gamma)$$

  and $y \in \mathit{Buffer}$; the valuation function on the controller is equivalent to the simple valuation of global configurations of $\mathcal{CP}$, and ignores the value of the buffer.
- $\nu'(x) = \mathit{True}$ if

  $$x \in (P_1 \times \Gamma) \cup \ldots \cup (P_n \times \Gamma)$$

  The valuation function on the n other pushdown components is not relevant and always true.

The CPDS and the single-indexed LTL formula. Let us consider the CPDS with $n+1$ components

$$\mathcal{CP}' = (\mathcal{C}, \mathcal{P}_1', \ldots, \mathcal{P}_n')$$

on the alphabet

$$\mathit{Act}' = \mathit{Act} \cup (P_1 \cup \ldots \cup P_n) \times \Gamma \times \{1, \ldots, n\}$$

and the single-indexed formula

$$\psi = (\varphi, \mathit{True}, \ldots, \mathit{True})$$

Our goal is to simulate (with some repetitions) in the controller component of $\mathcal{CP}'$ paths matched to global runs of the original CPDS $\mathcal{CP}$, and check the stutter-invariant formula φ there.

Computing a new global run. Let us assume that a global run g of $\mathcal{CP}$ verifying the stutter-invariant LTL formula φ exists. We will define inductively a global run g' on $\mathcal{CP}'$, starting from the initial configuration

$$((\langle s_0^1, \bot\rangle, \ldots, \langle s_0^n, \bot\rangle, \mathit{Empty}), c_0^1, \ldots, c_0^n)$$

For each transition of the form

$$(\langle p_1, \gamma_1 w_1\rangle, \ldots, \langle p_n, \gamma_n w_n\rangle) \xrightarrow{x}_{\mathcal{CP}} (\langle q_1, \gamma_1' w_1'\rangle, \ldots, \langle q_n, \gamma_n' w_n'\rangle)$$

belonging to g, we will design a sequence of transitions of g' such that, given

$$v = \nu((\langle p_1, \gamma_1 w_1\rangle, \ldots, \langle p_n, \gamma_n w_n\rangle))$$

and

$$v' = \nu((\langle q_1, \gamma_1' w_1'\rangle, \ldots, \langle q_n, \gamma_n' w_n'\rangle))$$

the valuation ν' on the controller $\mathcal{C}$ goes through a sequence of the form $v^\lambda v'^\mu$. Moreover, g' should otherwise be similar to g w.r.t. the other n pushdown components, interactions with the controller component notwithstanding.

1. If $x = \tau$, then the system $\mathcal{CP}$ executes in a component $\mathcal{P}_i$ an internal action
$$\langle p_i, \gamma_i w_i \rangle \xrightarrow{\tau}_{\mathcal{P}_i} \langle q_i, \gamma_i' w_i' \rangle$$
and $\forall j \neq i$
$$\langle p_j, \gamma_j w_j \rangle = \langle q_j, \gamma_j' w_j' \rangle$$
We first use in $\mathcal{CP}'$ an internal transition
$$\langle p_i, \gamma_i w_i \rangle \xrightarrow{\tau}_{\mathcal{P}_i'} \langle q_i^i, \gamma_i' w_i' \rangle$$
of the component $\mathcal{P}_i'$.
It is then followed by a synchronized action between the transition
$$\langle q_i^i, \gamma_i' w_i' \rangle \xrightarrow{(q_i, \gamma_i', i)}_{\mathcal{P}_i'} \langle q_i, \gamma_i' w_i' \rangle$$
of the system $\mathcal{P}_i'$ and the transition
$$((\langle p_1, \gamma_1 \rangle, \ldots, \langle p_i, \gamma_i \rangle, \ldots, \langle p_n, \gamma_n \rangle, \mathit{Empty}), \bot) \xrightarrow{(q_i, \gamma_i', i)} ((\langle p_1, \gamma_1 \rangle, \ldots, \langle q_i, \gamma_i' \rangle, \ldots, \langle p_n, \gamma_n \rangle, \mathit{Empty}), \bot)$$
of the controller $\mathcal{C}$.
Intuitively, the i-th component performs the internal action then synchronizes with the controller to update the stack configurations. $\mathcal{CP}$ performed the internal action in one step, $\mathcal{CP}'$ in two. The latter path stutters once, as the controller's state does not change during the first step.
2. if $x = a$, $a \in \mathit{Lab}_{i,j}$, then the system $\mathcal{CP}$ executes a synchronized action between two components $\mathcal{P}_i$ and $\mathcal{P}_j$, but otherwise, $\forall k \neq j, i$,
$$\langle p_k, \gamma_k w_k \rangle = \langle q_k, \gamma_k' w_k' \rangle$$
We first perform in $\mathcal{CP}$' a synchronized action between the transition
$$\langle p_i, \gamma_i w_i \rangle \xrightarrow{a}_{\mathcal{P}_i'} \langle q_i^j, \gamma_i' w_i' \rangle$$
of the component $\mathcal{P}_i'$ and the transition
$$\langle p_j, \gamma_j w_j \rangle \xrightarrow{a}_{\mathcal{P}_j'} \langle q_j^i, \gamma_j' w_j' \rangle$$
of the component $\mathcal{P}_j'$.
Then we perform a synchronized action between the transition
$$\langle q_i^j, \gamma_i' w_i' \rangle \xrightarrow{(q_i, \gamma_i', j)}_{\mathcal{P}_i'} \langle q_i, \gamma_i' w_i' \rangle$$
of the system $\mathcal{P}_i'$ and the transition
$$((\langle p_1, \gamma_1 \rangle, \ldots, \langle p_i, \gamma_i \rangle, \ldots, \langle p_n, \gamma_n \rangle, \mathit{Empty}), \bot) \xrightarrow{(q_i, \gamma_i', j)} ((\langle p_1, \gamma_1 \rangle, \ldots, \langle p_i, \gamma_i \rangle, \ldots, \langle p_n, \gamma_n \rangle, \langle q_i, \gamma_i', j \rangle), \bot)$$
of the controller $\mathcal{C}$.
It is finally followed by a synchronized action between the transition
$$\langle q_j^i, \gamma_j' w_j' \rangle \xrightarrow{(q_j, \gamma_j', i)}_{\mathcal{P}_j'} \langle q_j, \gamma_j' w_j' \rangle$$
of $\mathcal{P}_j'$ and the transition
$$((\langle p_1, \gamma_1 \rangle, \ldots, \langle p_i, \gamma_i \rangle, \ldots, \langle p_j, \gamma_j \rangle, \ldots, \langle p_n, \gamma_n \rangle, \langle q_i, \gamma_i', j \rangle), \bot) \xrightarrow{(q_j, \gamma_j', i)} ((\langle p_1, \gamma_1 \rangle, \ldots, \langle p_i', \gamma_i' \rangle, \ldots, \langle p_j', \gamma_j' \rangle, \ldots, \langle p_n, \gamma_n \rangle, \mathit{Empty}), \bot)$$
of the controller $\mathcal{C}$.
We perform a synchronized action between $\mathcal{P}_i$ and $\mathcal{P}_j$, then we update the controller by synchronizing $\mathcal{C}$ and $\mathcal{P}_i$, storing in the buffer the updated state to avoid a partially updated (hence incorrect) configuration, then we eventually synchronize $\mathcal{C}$ and $\mathcal{P}_j$ while emptying the buffer.

When the valuation path g of $\mathcal{CP}$ goes through the values vv', the valuation path g' of the controller in $\mathcal{CP}'$ goes through the values $v^2 v'$ in case 1, and through the values $v^3 v'$ in case 2. Since φ is stutter-invariant and holds for g, then it must hold for the projection of g' on the controller $\mathcal{C}$, hence $g' \models_{\nu'} \psi$.

From $\mathcal{CP}'$ to $\mathcal{CP}$. Let g' be a global run of $\mathcal{CP}'$. By design, if we remove any synchronized transition involving the controller from g' and replace every transition of the form $(p, \gamma) \xrightarrow{x} (q^j, w) \in \Delta_i'$ belonging to $\mathcal{P}_i'$ by its equivalent $(p, \gamma) \xrightarrow{x} (q, w) \in \Delta_i$ in $\mathcal{P}_i$, we end up with a global run g of $\mathcal{CP}$ such that $\nu(g)$ and $\nu'(g')$ are stutter-equivalent. Hence, $\nu(g) \models \varphi$ if and only if $\nu'(g') \models \psi$. □

7 Application to race conditions

A race condition is an issue peculiar to multi-threaded programs that happens when events do not occur in the order the programmer intended, such as concurrent operations on a shared memory location. In this section, we show a toy example of a race condition in a CPDS that can be detected thanks to our abstraction.

This example is inspired by the works in [6,9,15] on a Windows NT Bluetooth driver.

7.1 The CPDS model

We consider a network composed of three processes: one of these handles memory allocation, and the two others processes can synchronize with it in order to use memory to fulfil requests. These processes are the following:

MEMORY: handles the amount of free memory available; this amount decreases when another process uses memory; the process will send different signals depending on whether there is free memory left or not;

CONSUME: can arbitrarily use the free memory handled by the previous process;

REQUEST: has a stack of requests to fulfil, and will use memory to do so.

If MEMORY runs out of free memory and another process try to use some nonetheless, MEMORY will reach an error state.

Each process can be modelled by a PDS as follows:

7.1.1 The process MEMORY.

Let m and m_e be its two states. Its stack alphabet is $\{\gamma, \bot\}$. The number of γ's in the stack corresponds to the amount of memory available to other threads, a single γ being enough to handle a single request. This process will pop a γ from its stack if it receives a signal *use*. As an internal action, it can also push a γ on its stack (allocating memory) if there is no free memory left. It can send a signal *on* to other threads if there is at least one γ on the stack, and will send *off* otherwise. If it receives a *use* signal but there is no γ on the stack, it will instead move to the error state m_e.

The process MEMORY is represented by the following PDS rules:

(r_1) $(m, \gamma) \xrightarrow{on} (m, \gamma)$; the process signals that there is still free memory left;

(r_2) $(m, \bot) \xrightarrow{off} (m, \bot)$; the process signals that there is no free memory left;

(r_3) $(m, \bot) \xrightarrow{\tau} (m, \gamma\bot)$; the process allocates memory;

(r_4) $(m, \gamma) \xrightarrow{use} (m, \varepsilon)$; the amount of free memory available decreases;

(r_5) $(m, \bot) \xrightarrow{use} (m_e, \bot)$; the process reaches its error state.

7.1.2 The process CONSUME.

Let c, c_{check}, and c_{done} be its three states and $\bot$ its only stack symbol. This process can check if there is any free memory left by exchanging a signal *on* with MEMORY, then consume one unit by sending a signal *use*.

CONSUME is represented by the following PDS rules:

(r_6) $(c, \bot) \xrightarrow{on} (c_{check}, \bot)$; the process checks if there is any memory left;

(r_7) $(c_{check}, \bot) \xrightarrow{use} (c_{done}, \bot)$; the process uses one unit of memory;

(r_8) $(c_{done}, \bot) \xrightarrow{\tau} (c, \bot)$; the process goes back to its initial waiting state.

7.1.3 The process REQUEST.

Let r and r_{check} be its two states. Its stack alphabet is $\{\gamma, \bot\}$. The number of γ's in the stack corresponds to the number of requests it must handle. As an internal action, it can receive a new request and push a γ symbol on its stack. This process can check if there is any free memory left by exchanging a signal *on* with MEMORY, then handle a request and consume one unit by sending a signal *use*, popping a symbol γ from its own stack.

The process REQUEST is represented by the following PDS rules:

(r_{9a}) $(r, \gamma) \xrightarrow{\tau} (r, \gamma\gamma)$; the process adds a new request;

(r_{9b}) $(r, \bot) \xrightarrow{\tau} (r, \gamma\bot)$; the process adds a new request;

(r_{10}) $(r, \gamma) \xrightarrow{on} (r_{check}, \gamma)$; the process checks if there is any free memory left;

(r_{11}) $(r_{check}, \gamma) \xrightarrow{use} (r, \varepsilon)$; the process handles a request while using one unit of memory.

7.2 Using a single-indexed LTL formula

Let P be the set of all states of the CPDS. We define $AP = P$ and a simple valuation ν such that for each stack symbol x and $p \in P$, $\nu(\langle p, x\rangle) = \{p\}$. We express a correct behaviour of the CPDS as the conjunction of the three following LTL formulas:

- $\psi_{MEMORY} = G(\neg m_e)$; the process MEMORY can't reach its error state;
- $\psi_{CONSUME} = GF(c)$; the process CONSUME will always go back to its waiting state c;
- $\psi_{REQUEST} = G(r_{check} \Rightarrow F(r))$; the process REQUEST, when it starts handling a request, must complete it and go back to its default state.

We then apply the prefix abstraction scheme introduced in Section 5.2 to the universal single-indexed model checking problem

$$(\mathit{MEMORY}, \mathit{CONSUME}, \mathit{REQUEST}) \models (\psi_{MEMORY}, \psi_{CONSUME}, \psi_{REQUEST})$$

Our algorithm finds a counter-example in seven steps. Intuitively, a race condition happens when both CONSUME and REQUEST try to use memory while MEMORY only has a single unit available.

7.3 An erroneous trace

We write $(r_i) \leftrightarrow (r_j)$ if we apply two rules that synchronize. We start from the initial configuration:

$$(\langle m, \bot\rangle, \langle c, \bot\rangle, \langle r, \bot\rangle)$$

(r_3) MEMORY allocates memory:

$$(\langle m, \gamma\bot\rangle, \langle c, \bot\rangle, \langle r, \bot\rangle)$$

$(r_6) \leftrightarrow (r_1)$ MEMORY sends *on* to CONSUME:

$$(\langle m, \gamma\bot\rangle, \langle c_{check}, \bot\rangle, \langle r, \bot\rangle)$$

(r_{9b}) REQUEST adds a new request:

$$(\langle m, \gamma\bot\rangle, \langle c_{check}, \bot\rangle, \langle r, \gamma\bot\rangle)$$

$(r_{10}) \leftrightarrow (r_1)$ MEMORY sends *on* to REQUEST:

$$(\langle m, \gamma\bot\rangle, \langle c_{check}, \bot\rangle, \langle r_{check}, \gamma\bot\rangle)$$

$(r_7) \leftrightarrow (r_4)$ CONSUME sends *use* to MEMORY and the latter process uses one unit of memory:

$$(\langle m, \bot\rangle, \langle c_{done}, \bot\rangle, \langle r_{check}, \gamma\bot\rangle)$$

(r_8) CONSUME goes back to default mode:

$$(\langle m, \bot\rangle, \langle c, \bot\rangle, \langle r_{check}, \gamma\bot\rangle)$$

$(r_{11}) \leftrightarrow (r_5)$: REQUEST sends *use* to MEMORY and the latter process reaches an error mode, violating ψ_{MEMORY}:

$$(\langle m_e, \bot\rangle, \langle c, \bot\rangle, \langle r, \bot\rangle)$$

Using a prefix abstraction of order 7, we can find an erroneous prefix $\tau \cdot on \cdot use \cdot \tau \cdot use \cdot \tau \cdot use$ in 7 steps. This prefix can be matched to a concrete, infinite global run with trace $\tau \cdot on \cdot use \cdot \tau \cdot use \cdot \tau \cdot use \cdot \tau^\omega$ that violates the single indexed-formula by manually extending the finite sequence of rules outlined above with an infinite loop of rule (r_{9a}).

8 Conclusion and Future Works

In this paper, we study the model checking problem of single-indexed and stutter-invariant LTL properties for CPDSs, which is unfortunately undecidable. We design an algorithm to abstract the model checking problem that relies on the automata-theoretic approach of [2, 7] and the Kleene abstraction framework of [3]. We then apply this technique to a toy example and find a race condition.

An automata-theoretic approach to the CTL model checking problem for PDSs has been introduced in [17]. It remains to be seen if the CTL model checking problem for CPDSs can be abstracted in a similar manner to LTL.

References

1. M. F. Atig. Model-checking of ordered multi-pushdown automata. *Logical Methods in Computer Science*, 8(3), 2012.
2. A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model-checking. In A. Mazurkiewicz and J. Winkowski, editors, *CONCUR '97: Concurrency Theory*, pages 135–150, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.
3. A. Bouajjani, J. Esparza, and T. Touili. A generic approach to the static analysis of concurrent programs with procedures. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '03, pages 62–73, New York, NY, USA, 2003. ACM.
4. A. Bouajjani, M. Müller-Olm, and T. Touili. Regular symbolic analysis of dynamic networks of pushdown systems. In M. Abadi and L. de Alfaro, editors, *CONCUR 2005 – Concurrency Theory*, pages 473–487, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.
5. D. Caucal. On the regular structure of prefix rewriting. *Theoretical Computer Science*, 106(1):61 – 86, 1992.
6. S. Chaki, E. Clarke, N. Kidd, T. Reps, and T. Touili. Verifying concurrent message-passing c programs with recursive calls. In H. Hermanns and J. Palsberg, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, ETAPS '06, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.
7. J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. Efficient algorithms for model checking pushdown systems. In *Proceedings of the 12th International Conference on Computer Aided Verification*, CAV '00, pages 232–247, London, UK, UK, 2000. Springer-Verlag.
8. Y. Kesten, Z. Manna, H. McGuire, and A. Pnueli. A decision algorithm for full propositional temporal logic. In C. Courcoubetis, editor, *Computer Aided Verification*, pages 97–109, Berlin, Heidelberg, 1993. Springer Berlin Heidelberg.
9. G. Patin, M. Sighireanu, and T. Touili. Spade: Verification of multithreaded dynamic and recursive programs. In W. Damm and H. Hermanns, editors, *Computer Aided Verification*, pages 254–257, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.
10. D. A. Peled and T. Wilke. Stutter-invariant temporal properties are expressible without the next-time operator. *Inf. Process. Lett.*, 63(5):243–246, 1997.
11. A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, SFCS '77, pages 46–57, Washington, DC, USA, 1977. IEEE Computer Society.
12. A. Pommellet and T. Touili. Static analysis of multithreaded recursive programs communicating via rendez-vous. In *APLAS*, volume 10695 of *Lecture Notes in Computer Science*, pages 235–254. Springer, 2017.
13. A. Pommellet and T. Touili. LTL model-checking for communicating concurrent programs. In *Verification and Evaluation of Computer and Communication Systems - 12th International Conference, VECoS 2018, Grenoble, France, September 26-28, 2018, Proceedings*, pages 150–165, 2018.
14. S. Qadeer and J. Rehof. Context-bounded model checking of concurrent software. In N. Halbwachs and L. D. Zuck, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 93–107, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.
15. S. Qadeer and D. Wu. Kiss: Keep it simple and sequential. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, PLDI '04, pages 14–24, New York, NY, USA, 2004. ACM.

16. G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM Trans. Program. Lang. Syst.*, 22(2):416–430, Mar. 2000.
17. F. Song and T. Touili. Efficient CTL model-checking for pushdown systems. *Theoretical Computer Science*, 549:127 – 145, 2014.
18. F. Song and T. Touili. Model-checking dynamic pushdown networks. *Formal Aspects of Computing*, 27(2):397–421, Mar 2015.
19. F. Song and T. Touili. LTL model-checking for dynamic pushdown networks communicating via locks. *CoRR*, abs/1611.02528, 2016.
20. S. L. Torre, P. Madhusudan, and G. Parlato. A robust class of context-sensitive languages. In *22nd Annual IEEE Symposium on Logic in Computer Science (LICS 2007)*, pages 161–170, July 2007.

A Kleene Abstractions

We detail the mathematical framework introduced by Bouajjani et al. in [3] in order to abstract trace languages of PDSs.

A.1 Abstractions and Galois connections

Let $\mathcal{L} = (2^{Act^*}, \subseteq, \cup, \cap, \emptyset, Act^*)$ be the complete lattice of languages on *Act*.

Our abstraction of $\mathcal{L}$ requires a lattice $E = (D, \leq, \sqcup, \sqcap, \bot, \top)$, from now on called the *abstract lattice*, where D is a set called the abstract domain, as well as a pair of mappings (α, β) called a *Galois connection*, where $\alpha : 2^{Act^*} \to D$ and $\beta : D \to 2^{Act^*}$ are such that

$$\forall x \in 2^{Act^*}, \forall y \in D, \alpha(x) \leq y \Leftrightarrow x \subseteq \beta(y)$$

$\forall L \in \mathcal{L}$, given a Galois connection (α, β), we have $L \subseteq \beta(\alpha(L))$; the Galois connection can be used to over-approximate a language in 2^{Act^*} (e.g. the trace language of a PDS).

Moreover, it is easy to see that

$$\forall L_1, \forall L_2 \in \mathcal{L}, \alpha(L_1) \sqcap \alpha(L_2) = \bot$$

if and only if

$$\beta(\alpha(L_1)) \cap \beta(\alpha(L_2)) = \emptyset$$

A.2 Kleene algebras

We consider a special class of abstractions called *Kleene abstractions.*

An *idempotent semiring* is a structure $K = (A, \oplus, \odot, \overline{0}, \overline{1})$, where $\oplus$ is an associative, commutative, and idempotent ($a \oplus a = a$) operation such that A is closed under the infinite sum $\bigoplus$, and $\odot$ is an associative operation. $\overline{0}$ and $\overline{1}$ are neutral elements for $\oplus$ and $\odot$ respectively, $\overline{0}$ is an annihilator for $\odot$ ($a \odot \overline{0} = \overline{0} \odot a = \overline{0}$) and $\odot$ distributes over $\oplus$.

K is an *Act-semiring* if it can be generated by $\overline{0}$, $\overline{1}$, and elements of the form $v_a \in A$, $\forall a \in Act$. A semiring is said to be closed if $\oplus$ can be extended to an operator over countably infinite sets while keeping the same properties as $\oplus$.

We define $a^0 = \overline{1}$, $a^{n+1} = a \odot a^n$ and $a^* = \bigoplus_{n \geq 0} a^n$. Adding the $*$ operation to an idempotent closed *Act*-semiring K transforms it into a *Kleene algebra.*

A.3 Kleene abstractions

An abstract lattice $E = (D, \leq, \sqcup, \sqcap, \bot, \top)$ is said to be compatible with a Kleene algebra $K = (A, \oplus, \odot, \overline{0}, \overline{1})$ if $D = A$, $x \leq y \Leftrightarrow x \oplus y = y$, $\bot = \overline{0}$ and $\sqcup = \oplus$.

A *Kleene abstraction* is an abstraction such that the abstract lattice E is compatible with the Kleene algebra and the Galois connections $\alpha : 2^{Act^*} \to D$ and $\beta : D \to 2^{Act^*}$ are defined by:

$$\alpha(L) = \bigoplus_{a_1 \ldots a_n \in L} v_{a_1} \odot \ldots \odot v_{a_n}$$

$$\beta(x) = \{a_1 \ldots a_n \in Act^* \mid v_{a_1} \odot \ldots \odot v_{a_n} \leq x\}$$

Intuitively, a Kleene abstraction is such that the abstract operations $\oplus$, $\odot$, and $*$ can be matched to the union, the concatenation, and the Kleene closure of the languages of the lattice $\mathcal{L}$, $\overline{0}$ and $\overline{1}$ to the empty language and $\{\varepsilon\}$, v_a to the language $\{a\}$, the upper bound $\top \in K$ to Act^*, and the operation $\sqcap$ to the intersection of languages in the lattice $\mathcal{L}$.

In order to compute $\alpha(L)$ for a given language L, each word $a_1 \dots a_n$ in L is matched to its abstraction $v_{a_1} \odot \dots \odot v_{a_n}$, and we consider the sum of these abstractions.

A *finite-chain* abstraction is such that the lattice $(A, \leq)$ has no infinite ascending chains: as a consequence, infinite sequences of concatenations $\bigodot_{i \geq 0} v_{a_i}$ can be defined and effectively computed, and the abstraction framework can be applied to infinite traces in Act^ω. The prefix and first occurrence ordering abstractions defined previously are finite-chain.

A.4 The set of K-predecessors

Let $\mathcal{P} = (P, Act, \Gamma, \Delta, c_0)$ be a PDS and $K = (A, \oplus, \odot, \overline{0}, \overline{1})$ a Kleene algebra corresponding to a Kleene abstraction of the set Act.

We define inductively the set Π_K of *path expressions* as the smallest subset of K such that:

- $\overline{1} \in \Pi_K$;
- if $\pi \in \Pi_K$, then $\forall a \in Act$, $v_a \odot \pi \in \Pi_K$.

For a given path expression π, we define its length $|\pi|$ as the number of occurrences of simple elements of the form v_a in π.

A *K-configuration* of $\mathcal{P}$ is a pair (c, π) in $Conf^K_{\mathcal{P}} = P \times \Gamma^* \times \Pi_K$. We can extend the transition relation $\longrightarrow_{\mathcal{P}}$ to K-configurations with the following semantics: $\forall a \in Act$, if $c \xrightarrow{a}_{\mathcal{P}} c'$, then

$$\forall \pi \in \Pi_K, (c, v_a \odot \pi) \longrightarrow_{\mathcal{P},K} (c', \pi)$$

$(c, v_a \odot \pi)$ is said to be an immediate K-predecessor of (c', π). The reachability relation $\rightsquigarrow_{\mathcal{P},K}$ is the reflexive transitive closure of $\longrightarrow_{\mathcal{P},K}$.

Given a set of configurations C, we introduce the set of K-configurations $pre^*_K(\mathcal{P}, C)$:

$$pre^*_K(\mathcal{P}, C) = \{(c, \pi) \mid c \in pre^*(\mathcal{P}, C), \\ \pi \leq \alpha(\mathcal{L}_{\mathcal{P}}(\{c\}, C))$$

If $(c, \pi) \in pre^*_K(\mathcal{P}, C)$, then, by definition of the Kleene abstraction, $(c, \pi) \rightsquigarrow_{\mathcal{P},K} (c', \overline{1})$ for some $c' \in C$. Intuitively, the abstract path expression π is meant to be the abstraction of an actual trace from c' to C.

B Abstracting Traces of Pushdown Systems

We now present the automata-theoretic approach used by Bouajjani et al. in [3] to compute the set $pre^*_K(\mathcal{P}, C)$ in order to abstract runs of PDSs.

B.1 Representing regular sets of configurations

In order to represent regular sets of configurations, we consider the following structure:

Definition 12 (Bouajjani et al. [2]) Let $\mathcal{P} = (P, \Sigma, \Gamma, \Delta, c_0)$ be a pushdown system. A $\mathcal{P}$-automaton $\mathcal{A} = (Q, \Gamma, \delta, I, F)$ is a finite-state automaton on the stack alphabet Γ of $\mathcal{P}$ where Q is a set of states such that $P \subseteq Q$, $I = P$ the set of initial states, $F \subseteq Q$ the set of final states, and $\delta \subseteq Q \times \Gamma \cup \{\varepsilon\} \times Q$ a set of transitions.

Intuitively, a $\mathcal{P}$-automaton is a finite-state automaton whose edges are labelled by stack symbols of $\mathcal{P}$ and whose initial states represent the states of $\mathcal{P}$.

Let $\rightarrow_{\mathcal{A}}$ be the transition relation inferred from δ. We say that $\mathcal{A}$ *accepts* a configuration $\langle p, w \rangle$ if there is a path $p \xrightarrow{w}^*_{\mathcal{A}} f$ such that $f \in F$. Let $\mathcal{L}(\mathcal{A}) \subseteq Conf_{\mathcal{P}}$ be the set of configurations accepted by $\mathcal{A}$. Obviously, the following lemma holds:

Lemma 3 (Bouajjani et al. [2]) *A set of configurations $\mathcal{C}$ of a PDS $\mathcal{P}$ is regular if and only if there exists a $\mathcal{P}$-automaton $\mathcal{A}$ such that $\mathcal{L}(\mathcal{A}) = \mathcal{C}$.*

Let $\mathcal{C}$ be a regular set of configurations of a PDS $\mathcal{P} = (P, \Sigma, \Gamma, \Delta)$, and let $\mathcal{A}$ be a $\mathcal{P}$-automaton accepting $\mathcal{C}$. It has been proven by Didier Caucal in [5] that the sets $pre^*(\mathcal{P}, \mathcal{C})$ is regular. Moreover, it can be computed by applying a saturation procedure (where new transitions are incrementally added until a fixed point is reached) to a $\mathcal{P}$-automaton representing C:

Theorem 8 (Bouajjani et al. [2]) *Given a PDS $\mathcal{P}$ and a regular set of configurations $\mathcal{C}$, there exists a $\mathcal{P}$-automaton $\mathcal{A}_{pre^*}$ accepting $pre^*(\mathcal{C})$.*

B.2 K-automata

$\mathcal{P}$-automata are used to represent regular sets of configurations. They can be extended to K-automata in order to handle sets of K-configurations of a PDS $\mathcal{P}$.

Definition 13 (K-automata) A *K-automaton* matched to a PDS $\mathcal{P} = (P, \Sigma, \Gamma, \Delta)$ is a tuple $\mathcal{A} = (Q, \Gamma, \delta, I, F)$ where Q is a finite set of control states, $\delta \subseteq Q \times \Gamma \times K \times Q$ a finite set of transition rules, $I = P$ the set of initial states, and $F \subseteq Q$ the set of final states.

Intuitively, a K-automaton can be seen as a $\mathcal{P}$-automaton whose transitions labelled by stack symbols in Γ have been given an additional label in K. In a similar manner, $\mathcal{P}$-automaton can be seen as K-automaton whose transitions are all labelled by $\overline{1}$.

We define $\longrightarrow_{\mathcal{A}} \subseteq Q \times \Gamma^* \times K \times Q \times$ as the smallest transition relation satisfying:

- $q \xrightarrow{(\varepsilon, \overline{1})}_{\mathcal{A}} q$ for every $q \in Q$;
- if $(q, \gamma, e, q') \in \delta$, then $q \xrightarrow{(\gamma, e)}_{\mathcal{A}} q'$;
- if $q \xrightarrow{(w, e)}_{\mathcal{A}} q'$ and $q' \xrightarrow{(w', e')}_{\mathcal{A}} q''$ as well, then $q \xrightarrow{(ww', e \odot e')}_{\mathcal{A}} q''$.

We say that the K-automaton $\mathcal{A}$ accepts a K-configuration $(< p, w >, \pi)$ if $p \xrightarrow{(w, e)}_{\mathcal{A}} q$ for $q \in F$ and some $e \in K$ such that $\pi \leq e$.

Let $\mathcal{L}_K(\mathcal{A})$ be the set of all configurations accepted by $\mathcal{A}$, and $\mathcal{T}_K(\mathcal{A}) = \{\pi \mid \exists c \in C, (c, \pi) \in \mathcal{L}_K(\mathcal{A})\}$ the set of abstract traces matched to these configurations.

By adding extra labels in K to the $\mathcal{P}$-automaton $\mathcal{A}_{pre^*}$ accepting $pre^*(\mathcal{C})$ yielded by Theorem 1, we can compute a K-automaton accepting set $pre^*_K(\mathcal{P}, C)$ using an iterative fixpoint algorithm:

Theorem 9 (Bouajjani et al [3]) *Let $\mathcal{P} = (P, \Sigma, \Gamma, \Delta)$ be a PDS, $\mathcal{A}$ a $\mathcal{P}$-automaton accepting a regular set of configurations C, and K a Kleene algebra matched to a finite-chain Kleene abstraction α. Then we can compute a K-automaton $\mathcal{A}_{pre^*_K}$ matched to $\mathcal{P}$ accepting the set $pre^*_K(\mathcal{P}, C)$.*

Assuming the size of C is constant, the running time of this algorithm is in $O(l \cdot |\Delta|^3)$ operations.

B.3 Approximating the set of traces

Let $\mathcal{P}$ be a PDS and C, C' two regular sets of configurations of $\mathcal{P}$. Let K be a Kleene algebra matched to a finite-chain Kleene abstraction α. Using Theorem 9, we can compute a K-automaton $\mathcal{A}_{pre^*_K}$ matched to $\mathcal{P}$ accepting the set $pre^*_K(\mathcal{P}, C)$.

We can then construct a K-automaton $\mathcal{A}'_{pre^*_K}$ over $\Gamma \times K$ equal to the restriction of $\mathcal{A}_{pre^*_K}$ to configurations in C'. To do so, we consider the product between $\mathcal{A}_{pre^*_K}$ and a $\mathcal{P}$-automaton accepting the regular set of configurations C'.

Finally, $\alpha(\mathcal{L}_{\mathcal{P}}(C', C)) = \mathcal{T}_K(\mathcal{A}'_{pre^*_K})$ is an abstraction of the set $\mathcal{L}_{\mathcal{P}}(C', C)$ of traces of the PDS $\mathcal{P}$ leading from the regular set of configuration C' to another regular set of configurations C.

Obviously, the semantics of a PDS and a BPDA being similar, we can apply this framework to compute finite traces of BPDA as well.