



# LTL Model-Checking for Communicating Concurrent Programs

Adrien Pommellet<sup>1</sup>(✉) and Tayssir Touili<sup>2</sup>

<sup>1</sup> LIPN and Université Paris-Diderot, Paris, France  
pommellet@irif.fr

<sup>2</sup> LIPN, CNRS, and Université Paris 13, Villetaneuse, France

**Abstract.** Communicating Pushdown Systems (CPDSs) can be used to model multi-threaded programs with recursive procedure calls and synchronisation by rendez-vous between parallel threads. While the reachability problem for this particular class of automata is undecidable, it can be tackled using an algebraic framework for computing abstractions of context-free languages. In this paper, we combine this framework with an automata-based approach in order to approximate an answer to the model-checking problem for Linear Temporal Logic (LTL) on CPDSs: we show that, given a single-indexed LTL formula, we can accurately tell if a CPDS does not follow this formula. Finally, we show how this method can be used to find race conditions in concurrent programs.

## 1 Introduction

The use of parallel programs has grown in popularity in the past fifteen years, but these remain 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* (also known as 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* are a natural model for programs with sequential, recursive procedure calls [6]. 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, as each thread is modelled as a PDS, and can synchronize by rendez-vous with other threads. 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 execution paths language, using a framework based on Kleene algebras.

Solving the model-checking problem of LTL for CPDSs would therefore be a worthy addition to the existing verification techniques. However, this problem is unfortunately undecidable. Our contributions in this paper are the following:

- we define the semantics of *single-indexed* LTL formulas for CPDS, that is, formulas of the form  $\varphi = (\psi_1, \dots, \psi_n)$ , where each LTL sub-formula  $\psi_i$  must hold for the  $i$ -th process with regards to the synchronized CPDS semantics;
- we 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 PDS developed by Esparza et al. in [6];
- we use this abstraction on isolated pushdown components to compute an over-approximation of the model-checking problem of single-indexed LTL for CPDSs;
- we apply this abstraction framework to detect race conditions in a toy example.

**Related Work.** *Multi-Stack Pushdown Systems* (MPDSs) are pushdown systems with two or more stacks that can be used to model synchronized parallel programs. In [8] Qadeer et al. solve the model-checking problem of LTL given a context-bounding constraint on runs, where context is an uninterrupted sequence of actions by 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 shown by La Torre et al. in [11]. 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 with an 2ETIME upper bound. These models depend on a 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 systems that can spawn new threads, also modeled as pushdown systems. Song et al. described in [10] 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. Hence, a single-indexed formula on a DPN is of the form  $\varphi = (\psi_1, \dots, \psi_n)$ , where each component  $\psi_i$  is a formula that must hold for the  $i$ -th thread pattern. While CPDSs can't model thread spawns, but DPNs do not feature synchronization between threads, an important aspect of concurrent programs.

*Synchronized dynamic pushdown networks* (DPNs) were later introduced by Pommellet et al. in [7]. 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.** In Sect. 2 of this paper, we remind the reader of the definition of *communicating pushdown systems* (CPDSs). In Sect. 3, we define the *single-indexed linear-time temporal logic* for CPDSs. We describe in Sect. 4 the abstraction framework designed by Bouajjani et al. in [3] in order to over-approximate the set of execution paths of a PDS. In Sect. 5, as a main contribution of this paper, we introduce an abstract model-checking algorithm. We apply this algorithm to an example in Sect. 6. Finally, we show our conclusion in Sect. 7.

## 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  $\mathcal{P} = (P, Act, \Gamma, \Delta, c_0)$  where  $P$  is a finite set of control states,  $Act$  a finite input alphabet, also called the set of actions,  $\Gamma$  a finite stack alphabet,  $\Delta \subseteq P \times \Gamma \times Act \times P \times \Gamma^*$  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  $\Delta \subseteq P \times \Gamma \times Act \times P \times \Gamma^{\leq 2}$ .

**Configurations of PDSs.** 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 configurations of  $\mathcal{P}$ . 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  $\Gamma$  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  $\mathcal{A}$ .

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

**Definition 2 (Bouajjani et al. [2]).** Let  $\mathcal{P} = (P, Act, \Gamma, \Delta, c_0)$  be a pushdown system. A  $\mathcal{P}$ -automaton  $\mathcal{A} = (Q, \Gamma, \delta, I, F)$  is a finite automaton on the stack alphabet  $\Gamma$  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.

Let  $\rightarrow_{\mathcal{A}}$  be the transition relation inferred from  $\delta$ . 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  $L(\mathcal{A}) \subseteq Conf_{\mathcal{P}}$  be the set of configurations accepted by  $\mathcal{A}$ . Intuitively, a  $\mathcal{P}$ -automaton is a finite automaton whose edges are labelled by stack symbols of  $\mathcal{P}$  and whose initial states represent the states of  $\mathcal{P}$ . The following lemma holds:

**Lemma 1 (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  $L(\mathcal{A}) = \mathcal{C}$ .

**The Reachability Relation.** For each  $a \in Act$ , we define the transition relation  $\xrightarrow{a}_{\mathcal{P}}$  on configurations as follows: if  $(p, \gamma) \xrightarrow{a} (p', w) \in \Delta$ , for each  $w' \in \Gamma^*$ ,  $\langle p, \gamma w' \rangle \xrightarrow{a}_{\mathcal{P}} \langle p', w w' \rangle$ . From these relations, we can then infer the *immediate successor* relation  $\rightarrow_{\mathcal{P}} = \bigcup_{a \in Act} \xrightarrow{a}_{\mathcal{P}}$ . The *reachability* relation  $\rightarrow_{\mathcal{P}}^*$  is the reflexive and transitive closure of the immediate successor relation  $\rightarrow_{\mathcal{P}}$ .

A *run*  $r$  is a sequence of configurations  $r = (r_i)_{i \geq 0}$  such that  $r_0 = c_0$ , and  $\forall i \geq 0, r_i \xrightarrow{a_i}_{\mathcal{P}} r_{i+1}$ . The sequence  $(a_i)_{i \geq 0}$  of actions is then said to be the *trace* of  $r$ . Traces and runs may be finite or infinite. Let  $Runs_{\omega}(\mathcal{P})$  (resp.  $Runs(\mathcal{P})$ ) be the set of all infinite (resp. finite) runs of  $\mathcal{P}$ . We define  $Traces_{\omega}(\mathcal{P})$  and  $Traces(\mathcal{P})$  in a similar manner.

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.

It has been shown in [2] that the set of predecessors  $pre^*(\mathcal{P}, \mathcal{C})$  is regular and can be computed by applying a saturation procedure:

**Theorem 1 (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  $pre^*(\mathcal{C})$ .*

## 2.2 The Model and Its Semantics

We now model each thread in a concurrent program as a PDS:

**Definition 3 (Bouajjani et al. [3]).** *A communicating pushdown system (CPDS) is a tuple  $\mathcal{CP} = (\mathcal{P}_1, \dots, \mathcal{P}_n)$  of pushdown systems sharing the same input alphabet  $Act$  and the same stack alphabet  $\Gamma$ .*

From then on, we assume that the set  $Act$  contains a special action  $\tau$  that represents internal actions, and that other letters in  $Lab = Act \setminus \{\tau\}$  model synchronization signals. We assume that each pair  $(\mathcal{P}_i, \mathcal{P}_j)$  of pushdown systems in the network uses a dedicated set of signals  $Lab_{i,j}$  disjoint from the other sets of signals.

A *global configuration* of  $\mathcal{CP}$  is a tuple  $g = (c_1, \dots, c_n)$  of configurations of  $\mathcal{P}_1, \dots, \mathcal{P}_n$ . Let  $Conf_{\mathcal{CP}}$  be the set of global configurations of the CPDS  $\mathcal{CP}$ . The *global starting configuration* of  $\mathcal{CP}$  is the tuple  $g_0 = (c_0^1, \dots, c_0^n)$  where  $c_0^i$  is the starting configuration of  $\mathcal{P}_i$ . We define a transition relation  $\xrightarrow{a}_{\mathcal{CP}}$  on global configurations as follows:

- $(c_1, \dots, c_n) \xrightarrow{\tau}_{\mathcal{CP}} (c'_1, \dots, c'_n)$  if there is  $i$  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;
- $(c_1, \dots, c_n) \xrightarrow{a}_{\mathcal{CP}} (c'_1, \dots, c'_n)$  if there are  $i$  and  $j$ ,  $i \neq j$  such that  $c_i \xrightarrow{a}_{\mathcal{P}_i} c'_i$ ,  $c_j \xrightarrow{a}_{\mathcal{P}_j} c'_j$  and  $c_k = c'_k$  for all  $k \neq i \neq j$ ; two synchronized processes perform a simultaneous action.

We define runs and traces with regards to this transition relation in a manner similar to PDSs. Given a global run  $g$ , we define  $g^i$  as its projection on its  $i$ -th component  $Conf_{\mathcal{P}_i}$ .

### 2.3 From a Program to a CPDS Model

We can assume that the program is given 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  $y$ .

Without loss of generality, we assume that threads share no global variables and instead 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  $ch!(x)$ , value  $x$  is sent through channel  $ch$ . With a receive statement  $ch?(y)$ , the value received through channel  $ch$  is bound to variable  $y$ . We also consider that both local and global variables may only take a finite number of values.

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

The set  $Act$  contains an internal action  $\tau$  and an action  $ch(n)$  for each channel  $ch$  and value  $n$  that can be carried through it.  $Lab$  is a disjoint union of sets  $Lab_{i,j}$  corresponding to synchronization actions of the form  $ch_{i,j}(v)$  from a system  $\mathcal{P}_i$  to another system  $\mathcal{P}_j$ .

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  $g_1$  and  $g_2$  (resp.  $l_1$  and  $l_2$ ) are the valuations of global (resp. local) variables before and after the execution of the statement:

- if  $s$  is an assignment, it is represented by rules of the form  $(g_1, (n_1, l_1)) \xrightarrow{\tau} (g_2, (n_2, l_2))$ ; assigning new values to variables in  $g_1$  and  $l_1$  results in new valuations  $g_2$  and  $l_2$ ;
- if  $s$  is a procedure call, it is represented by rules of the form  $(g_1, (n_1, l_1)) \xrightarrow{\tau} (g_2, (f_0, l_0)(n_2, l_2))$ , 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 by rules of the form  $(g_1, (n_1, l_1)) \xrightarrow{\tau} (g_2, \varepsilon)$ ; we simulate returns of values by introducing an additional global variable and assigning the return value to it in the valuation  $g_2$ ;
- if  $s$  is an assignment  $ch?(y)$  of a value  $x$  carried through a channel  $c$  to a variable  $y$ , it is represented by rules of the form  $(g_1, (n_1, l_1)) \xrightarrow{ch(v)} (g_2, (n_2, l_2))$  where  $g_1$  and  $g_2$  (resp.  $l_1$  and  $l_2$ ) are such that assigning the value  $v$  to the variable  $y$  in  $g_1$  (resp.  $l_1$ ) results in the new valuations  $g_2$  (resp.  $l_2$ );
- if  $s$  is an output  $ch!(x)$  through a channel  $c$  of the value  $x$  of a variable  $y$ , it is represented by rules of the form  $(g_1, (n_1, l_1)) \xrightarrow{ch(v)} (g_2, (n_2, l_2))$  such that the variable  $y$  has value  $x$  in either  $g_1$  or  $l_1$ .

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

### 3 Model-Checking LTL on CPDSs

#### 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  $\rho = (\rho_i)_{i \geq 0}$  in the set  $Paths = (2^{AP})^\omega$ .

**Definition 4 (LTL).** *The set of LTL formulas is given by the following grammar:*

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

$\perp$  stands for the predicate ‘always true’.  $X$  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 5 (Semantics of LTL).** *Let  $\varphi$  be a LTL formula,  $\rho \in Paths$ , and  $i \in \mathbb{N}$ . We define inductively the semantics of the relation  $\rho, i \models \varphi$ :*

$$\begin{aligned} \rho, i \models \rho \text{ where } \rho \in AP &\Leftrightarrow \rho \in \rho_i \\ \rho, i \models X \varphi &\Leftrightarrow \rho, i + 1 \models \varphi \\ \rho, i \models \varphi \text{ U } \psi &\Leftrightarrow \exists j \geq i \text{ such that } \rho, j \models \psi \text{ and} \\ &\forall k \in \{i, \dots, j - 1\}, \rho, k \models \varphi \end{aligned}$$

as well as the obvious interpretation of the boolean operators.

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

#### 3.2 LTL Model-Checking for PDSs

We recall in this Section the model-checking problem for PDSs and the automata-theoretic framework introduced in [2, 6].

Let  $\nu : Conf_{\mathcal{P}} \rightarrow 2^{AP}$  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  $\nu : P \times \Gamma \rightarrow 2^{AP}$  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  $\nu$ . We write that  $r \models_{\nu} \varphi$  if  $\nu(r), 0 \models \varphi$ . The *model-checking* problem is defined as follows:

**Definition 6 (The model-checking problem).** *Given a LTL formula  $\varphi$ , a PDS  $\mathcal{P}$  with a starting configuration  $c_0$ , and a simple valuation  $\nu$  on configurations of  $\mathcal{P}$ , the model-checking problem consists in determining whether  $\exists r \in \text{Runs}_\omega(\mathcal{P}), r \models_\nu \varphi$ .*

In order to solve this problem, we consider the following class of automata:

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

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

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

**Theorem 2.** *Given a PDS  $\mathcal{P}$  and a LTL formula  $\varphi$ , there exists a BPDS  $\mathcal{BP}$  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  $r \models_\nu \varphi$ .*

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  $\text{Rep}(\mathcal{BP})$  be the finite set of repeating heads of  $\mathcal{BP}$ . The following lemma characterizes accepting runs with regards to repeating heads:

**Lemma 2.**  *$r$  is an accepting run of a BPDS  $\mathcal{BP}$  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.*

### 3.3 Single-Indexed LTL for CPDSs

Let  $\mathcal{CP} = (\mathcal{P}_1, \dots, \mathcal{P}_n)$  be a CPDS,  $\nu$  a simple valuation function on  $\text{Conf}_{\mathcal{P}_1} \cup \dots \cup \text{Conf}_{\mathcal{P}_n}$ , and for  $i = 1, \dots, n$ , let  $\psi_i$  be a LTL formula. The formula  $\varphi = (\psi_1, \dots, \psi_n)$  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 model-checking).** *Given a CPDS  $\mathcal{CP} = (\mathcal{P}_1, \dots, \mathcal{P}_n)$ , a global run  $g$  of  $\mathcal{CP}$ , and a single-indexed LTL formula  $\varphi = (\psi_1, \dots, \psi_n)$ ,  $g \models_\nu \varphi$  if and only if for each  $i = 1, \dots, n$ ,  $g^i \models_\nu \psi_i$ . Finding such a global run  $g$  is called the model-checking problem.*

Intuitively, each PDS  $\mathcal{P}_i$  in the CPDS satisfies formula  $f_i$ , but does so while synchronizing with the others PDSs. If the model-checking problem for CPDSs were decidable, so would be the reachability problem. However, since the latter is obviously undecidable, the former is as well.

We therefore seek to get at least an approximate answer to this problem. The issue with CPDSs is the following: for each  $i = 1, \dots, n$ , there may be a run  $r$  of the PDS  $\mathcal{P}_i$  satisfying a formula  $\psi_i$ , but a global, synchronized run on the CPDS  $(\mathcal{P}_1, \dots, \mathcal{P}_n)$  satisfying  $\varphi = (\psi_1, \dots, \psi_n)$  may not exist.

## 4 An Abstraction Framework for Traces

We seek to approximate global runs of CPDSs. To do so, we want to abstract traces of their pushdown components. We remind here the mathematical framework presented by Bouajjani et al. in [3] in order to abstract the language  $\mathcal{L}_{\mathcal{P}}(\{c_0\}, C) = \{t \in \text{Traces}(\mathcal{P}) \mid \exists c \in C, c_0 \xrightarrow{t}^* c\}$  of traces of a PDS  $\mathcal{P}$  leading from the starting configuration  $c_0$  to a regular set of configurations  $C$ .

### 4.1 Abstractions and Galois Connections

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

Our abstraction of  $\mathcal{L}$  requires a lattice  $E = (D, \leq, \sqcup, \sqcap, \perp, \top)$ , from now on called the *abstract lattice*, where  $D$  is a set called the abstract domain, as well as a pair of mappings  $(\alpha, \beta)$  called a *Galois connection*, where  $\alpha : 2^{\text{Act}^*} \rightarrow D$  and  $\beta : D \rightarrow 2^{\text{Act}^*}$  are such that  $\forall x \in 2^{\text{Act}^*}, \forall y \in D, \alpha(x) \leq y \Leftrightarrow x \subseteq \beta(y)$ .

$\forall L \in \mathcal{L}$ , given a Galois connection  $(\alpha, \beta)$ , we have  $L \subseteq \beta(\alpha(L))$ . Hence, the Galois connection can be used to over-approximate a language such as  $\mathcal{L}_{\mathcal{P}}(\{c_0\}, C)$ .

Moreover, it is easy to see that  $\forall L_1, \forall L_2 \in \mathcal{L}, \alpha(L_1) \sqcap \alpha(L_2) = \perp$  if and only if  $\beta(\alpha(L_1)) \cap \beta(\alpha(L_2)) = \emptyset$ . We can therefore check the emptiness of intersections of over-approximations directly in the abstract domain.

### 4.2 Kleene Abstractions

As defined in [3], an abstract lattice  $E = (D, \leq, \sqcup, \sqcap, \perp, \top)$  is said to be compatible with a Kleene algebra  $K = (A, \oplus, \odot, \bar{0}, \bar{1})$  if  $D = A$ ,  $x \leq y \Leftrightarrow x \oplus y = y$ ,  $\perp = \bar{0}$  and  $\sqcup = \oplus$ .

The Kleene algebra  $K$  is an *Act-semiring* if it can be generated by  $\bar{0}$ ,  $\bar{1}$ , and elements of the form  $v_a \in A$ ,  $\forall a \in \text{Act}$ . A *Kleene abstraction* is an abstraction such that the abstract lattice  $E$  is compatible with the Kleene algebra and the Galois connection  $\alpha : 2^{\text{Act}^*} \rightarrow D$  and  $\beta : D \rightarrow 2^{\text{Act}^*}$  is defined by:

$$\begin{aligned} \alpha(L) &= \bigoplus_{a_1 \dots a_n \in L} v_{a_1} \odot \dots \odot v_{a_n} \\ \beta(x) &= \left\{ a_1 \dots a_n \in 2^{\text{Act}^*} \mid v_{a_1} \odot \dots \odot v_{a_n} \leq x \right\} \end{aligned}$$

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  $\mathcal{L}$ ,  $\bar{0}$  and  $\bar{1}$  to the empty language and  $\{\varepsilon\}$ ,  $v_a$  to the language  $\{a\}$ , the upper bound  $\top \in K$  to  $\text{Act}^*$ , 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. Moreover, we must have  $v_{\tau} = \bar{1}$ .



A *finite-chain* abstraction is such that the lattice  $(K, \oplus)$  has no infinite ascending chains. Prefix and suffix abstractions are such examples on the lattice  $2^W$ , where  $W(n) = \{w \in Act^* \mid |w| \leq n\}$  is the set of words of length smaller than  $n$ .

### 4.3 The Set of $K$ -Predecessors

Let  $\mathcal{P} = (P, Act, \Gamma, \Delta, c_0)$  be a PDS and  $K = (A, \oplus, \odot, \bar{0}, \bar{1})$  a Kleene algebra corresponding to a Kleene abstraction of the set  $Lab$ . We define inductively the set  $\Pi_K$  of path expressions as the smallest subset of  $K$  such that:

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

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

A  $K$ -configuration of  $\mathcal{P}$  is a pair  $(c, \pi)$  in  $Conf_{\mathcal{P}}^K = P \times \Gamma^* \times \Pi_K$ . 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  $(c', \pi)$ . 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  $pre_K^*(\mathcal{P}, C)$  of  $K$ -configurations  $(c, \pi)$  such that  $(c, \pi) \rightsquigarrow_{\mathcal{P}, K} (c', \bar{1})$  for  $c' \in C$ :

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

As we will see later, the abstract path expression  $\pi$  is meant to be the abstraction of an actual trace from  $c'$  to  $C$ .

### 4.4 $K$ -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 9 ( $K$ -automaton).** A  $K$ -automaton is a tuple  $\mathcal{A} = (Q, \Gamma, \delta, I, F)$  where  $Q$  is a finite set of control states,  $\delta \subseteq Q \times \Gamma \times K \times Q \times$  a finite set of transition rules,  $I = P$  the set of initial states, and  $F \subseteq Q$  the set of final states.

Intuitively, a  $\mathcal{P}$ -automaton can be seen as  $K$ -automaton whose transitions are all labelled by  $\bar{1}$ .

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

- $q \xrightarrow{(\epsilon, \bar{1})}_{\mathcal{A}} q'$  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  $q' \xrightarrow{(w', e')}_{\mathcal{A}} q''$ , then  $q \xrightarrow{(ww', e \odot e')}_{\mathcal{A}} q''$ .

We say that  $\mathcal{A}$  accepts a  $K$ -configuration  $\langle p, w \rangle, \pi$  if  $p \xrightarrow{(w,e)}_{\mathcal{A}} q$  for  $q \in F$  and some  $e \in K$  such that  $\pi \leq e$ . Let  $L_K(\mathcal{A})$  be the set of all configurations accepted by  $\mathcal{A}$ , and  $P_K(\mathcal{A}) = \{\pi \mid \exists c \in C, (c, \pi) \in L_K(\mathcal{A})\}$  the set of abstract traces matched to these configurations.

By labelling the  $\mathcal{P}$ -automaton  $\mathcal{A}_{pre^*}$  accepting  $pre^*(C)$  yielded by Theorem 1, the following theorem has been proven:

**Theorem 3 (Bouajjani et al. [3]).** *Let  $P$  be a PDS and  $\mathcal{A}$  a  $\mathcal{P}$ -automaton accepting a regular set of configurations  $C$ . Then we can compute a  $K$ -automaton  $\mathcal{A}_{pre_K^*}$  accepting the set  $pre_K^*(P, C)$ .*

From there, it is possible to compute the abstract trace language using the product automaton  $\mathcal{A}'$  between  $\mathcal{A}_{pre_K^*}$  and a  $\mathcal{P}$ -automaton accepting  $\langle c_0, \Gamma^* \rangle$ .

## 5 Abstract Model-Checking of LTL for CPDSs

In this section, as a main contribution of this paper, we will introduce a semi-decision procedure for model-checking LTL on CPDSs.

### 5.1 Abstracting Accepting Traces of a BPDS

By Lemma 2, each accepting run of a BPDS  $\mathcal{BP}$  can be matched to a repeating head it visits infinitely often, and any run visiting a repeating head infinitely often is accepting. As a consequence, if we can for each repeating head compute (resp. abstract) the set of traces visiting it infinitely often, we can compute (resp. abstract) the set of accepting traces of the BPDS.

Let  $\langle p, \gamma \rangle \in Rep(\mathcal{BP})$  be a repeating head. An accepting trace visiting  $\langle p, \gamma \Gamma^* \rangle$  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.

In order to abstract the set of accepting traces visiting  $\langle p, \gamma \rangle$  infinitely often, we first compute the set  $pre_K^*(\mathcal{BP}, \langle p, \gamma \Gamma^* \rangle)$  of  $K$ -predecessors of configurations with this repeating head, using Theorem 3. Then, we consider the set  $pre_K^*(\mathcal{BP}, \langle p, \gamma \Gamma^* \rangle) \cap (c_0 \times \Pi_K)$  and check its emptiness.

It will be empty if the repeating head is not reachable from the starting configuration  $c_0$ . Otherwise, it will be equal to the product of  $c_0$  with an abstraction  $I_{\langle p, \gamma \rangle}$  of the set of traces from  $c_0$  to  $C$ . Therefore, the abstraction  $I_{\langle p, \gamma \rangle} = P_K(pre_K^*(\mathcal{BP}, \langle p, \gamma \Gamma^* \rangle) \cap (c_0^i \times \Pi_K))$  yields part (1) of our abstraction of the set of accepting traces of the BPDS.

Next, we want to abstract the set of paths between two occurrences of the repeating head. To do so, we use again the set  $pre_K^*(\mathcal{BP}, \langle p, \gamma \Gamma^* \rangle)$  of  $K$ -predecessors of configurations with a repeating head  $\langle p, \gamma \rangle$ . We consider its intersection  $pre_K^*(\mathcal{BP}, \langle p, \gamma \Gamma^* \rangle) \cap \langle p, \gamma \Gamma^* \rangle \times \Pi_K$  with the product of the set of configurations with a repeating head  $\langle p, \gamma \rangle$  with all path expressions.

This set of  $K$ -configurations abstracts traces between two configurations with the same repeating head  $\langle p, \gamma \Gamma^* \rangle$ . Therefore, the abstraction  $L_{\langle p, \gamma \rangle} = P_K(\text{pre}_K^*(\mathcal{BP}, \langle p, \gamma \Gamma^* \rangle) \cap \langle p, \gamma \Gamma^* \rangle \times \Pi_K)$  yields part **(2)** of our abstraction of the set of accepting traces of the BDPS.

In an accepting run of a BPDS, part **(1)** happens once, then **(2)** occurs infinitely often. Hence,  $R_{\langle p, \gamma \rangle} = I_{\langle p, \gamma \rangle} \odot (L_{\langle p, \gamma \rangle})^*$  is an abstraction of the set of accepting traces using the repeating head  $\langle p, \gamma \rangle$  infinitely often. This set can be computed in a finite-chain abstraction framework.

We can finally compute an abstraction  $R = \bigoplus_{\langle p, \gamma \rangle \in \text{Rep}(\mathcal{BP})} R_{\langle p, \gamma \rangle}$  of the set of all accepting traces of  $\mathcal{BP}$  by abstracting the set of accepting traces for each repeating head, then considering the finite sum of these sets.

## 5.2 Abstracting the Model-Checking Problem for CPDSs

Let  $\mathcal{CP} = (\mathcal{P}_1, \dots, \mathcal{P}_n)$  be a CPDS and  $\varphi = (\psi_1, \dots, \psi_n)$  a single-indexed LTL formula. We want to abstract the model-checking problem  $\mathcal{CP} \models \varphi$ . Our intuition is, for each component  $\mathcal{P}_i$ , to abstract the set of paths verifying  $\psi_i$ , then examine the emptiness of the intersection of these abstractions.

If  $n = 2$ , then for  $i = 1, 2$ , to each PDS  $\mathcal{P}_i$  and formula  $\psi_i$ , we match a BPDS  $\mathcal{BP}_i$  according to Theorem 2 and compute an abstraction of its sets of paths  $R^i$  as outlined previously.  $R^1 \sqcap R^2 = \perp$  implies that we can't find a trace that is accepting for both BPDSs, hence, there is no synchronized global run verifying  $\varphi$ .

However, in a global run of a CPDS, the execution paths of the pushdown components are interleaved. If the CPDS has more than two threads, synchronization signals with a third thread may occur in the global run but cannot be computed by abstracting runs of each BPDS on its own. We cannot therefore study the paths of a pushdown system  $\mathcal{P}_i$  in isolation from the other components. Without loss of generality, we assume that a partition of  $\text{Lab}$  such that  $\text{Lab} = \bigcup_{k \neq j} L_{k,j}$  and that transitions of the component  $\mathcal{P}_i$  can only be labelled by elements in  $\text{Lab}_i = \bigcup_{k \neq i} L_{i,k}$ . Intuitively, each pair  $(\mathcal{P}_i, \mathcal{P}_j)$  of pushdown components can only synchronize by using its own set of symbols  $\text{Lab}_i$ .

For each component  $\mathcal{P}_i$ , we then consider a new pushdown system  $\mathcal{P}'_i$  that extends  $\mathcal{P}_i$  with self-loops in each control state labelled by synchronization signals between pair of other processes in  $\text{Lab}_{j,k}$ ,  $j \neq k \neq i$ . The following lemma holds:

**Lemma 3.** *If  $g$  is a global run of  $\mathcal{CP}$ , then  $g_i$  is a run of  $\mathcal{P}'_i$ .*

We then want to abstract the set of paths of  $\mathcal{P}'_i$  verifying  $\psi_i$  for each  $i$  and consider the intersection of these abstractions. If it is indeed empty, the same property holds for the intersection of the actual sets of paths, and no global run satisfying  $\varphi$  exists in  $\mathcal{CP}$ .

To do so, to each PDS  $\mathcal{P}'_i$  and formula  $\psi_i$ , we match a BPDS  $\mathcal{BP}'_i$  according to Theorem 2. We then compute an abstraction  $R^i$  of the set of traces of the BPDS  $\mathcal{BP}'_i$ , as outlined in Sect. 5.1. The following theorem then holds:

**Theorem 4.** *If  $R^1 \sqcap \dots \sqcap R^n = \perp$ , then there is no global run of  $\mathcal{CP}$  accepting the single-indexed LTL formula  $\varphi$ .*

We can therefore over-approximate the model-checking problem for CPDSs.

### 5.3 Using Our Framework in a CEGAR Scheme

In a manner similar to the work of Chaki et al. in [5], we propose a semi-decision procedure that, in case of termination, answers exactly whether there exists a global run of a CPDS  $\mathcal{CP} = (\mathcal{P}_1, \dots, \mathcal{P}_n)$  satisfies a single-indexed LTL formula  $\varphi = (\psi_1, \dots, \psi_n)$ .

We introduce the following Counter-Example Guided Abstraction Refinement (CEGAR) scheme based on the finite-domain abstraction framework detailed previously, starting from  $n = 1$ .

**Abstraction:** for each PDS  $\mathcal{P}'_i$ , we compute an abstraction of the set of all accepting traces  $R^i$  of  $\mathcal{BP}'_i$  (the product between  $\mathcal{P}'_i$  and the Büchi automaton representing  $\psi_i$ ), using either the prefix or suffix abstraction of rank  $n$  introduced in [3];

**Verification:** we then check if  $R^1 \sqcap \dots \sqcap R^n = \perp$ ; if it is indeed true, then we conclude that no global run of  $\mathcal{CP}$  can satisfy  $\varphi$ ;

**Counter-Example Validation:** if there is such a global run, we then check if our abstraction introduced a spurious counter-example; if the counter-example is not spurious, then we conclude that there exists a global run of  $\mathcal{CP}$  satisfying  $\varphi$ ;

**Refinement:** if the counter-example was spurious, we go back to the first step, but use this time prefix and suffix abstractions of order  $n + 1$ .

If this procedure ends, we can decide the model-checking problem.

## 6 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 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.

### 6.1 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:

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

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;
- $(r_2) (m, \perp) \xrightarrow{off} (m, \perp)$ ; the process signals that there is no free memory left;
- $(r_3) (m, \perp) \xrightarrow{\tau} (m, \gamma\perp)$ ; the process allocates memory;
- $(r_4) (m, \gamma) \xrightarrow{use} (m, \varepsilon)$ ; the amount of free memory available decreases;
- $(r_5) (m, \perp) \xrightarrow{use} (m_e, \perp)$ ; the process reaches its error state.

**The Process CONSUME.** Let  $c$ ,  $c_{check}$ , and  $c_{done}$  be its three states and  $\perp$  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:

- $(r_6) (c, \perp) \xrightarrow{on} (c_{check}, \perp)$ ; the process checks if there is any memory left;
- $(r_7) (c_{check}, \perp) \xrightarrow{use} (c_{done}, \perp)$ ; the process uses one unit of memory;
- $(r_8) (c_{done}, \perp) \xrightarrow{\tau} (c, \perp)$ ; the process goes back to its initial waiting state.

**The Process REQUEST.** Let  $r$  and  $r_{check}$  be its two states. Its stack alphabet is  $\{\gamma, \perp\}$ . The number of  $\gamma$ '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  $\gamma$  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  $\gamma$  from its own stack.

REQUEST is represented by the following PDS rules:

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

## 6.2 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  $\nu$  such that for each stack symbol  $x$  and  $p \in P$ ,  $\nu(\langle p, x \rangle) = \{p\}$ . We express the desirable 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 use a CEGAR scheme to check if there is a global run  $g$  such that the single-indexed formula  $(\psi_{MEMORY}, \psi_{CONSUME}, \psi_{REQUEST})$  does not hold for  $g$ . Our algorithm finds such 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.

## 6.3 An Erroneous Trace

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

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

$(r_3)$  MEMORY allocates memory:

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

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

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

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

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

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

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

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

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

$(r_8)$  CONSUME goes back to default mode:

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

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

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

This an erroneous execution path in 7 steps. We can find it using a prefix abstraction of order 7.

## 7 Conclusion and Future Works

In this paper, we study the model-checking problem of single-indexed 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, 6] 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 [9]. It remains to be seen if the CTL model-checking problem for CPDSs can be abstracted in a similar manner to LTL.

## References

1. Atig, M.F.: Model-checking of ordered multi-pushdown automata. *Log. Methods Comput. Sci.* **8**(3), (2012)
2. Bouajjani, A., Esparza, J., Maler, O.: Reachability analysis of pushdown automata: application to model-checking. In: Mazurkiewicz, A., Winkowski, J. (eds.) CONCUR 1997. LNCS, vol. 1243, pp. 135–150. Springer, Heidelberg (1997). [https://doi.org/10.1007/3-540-63141-0\\_10](https://doi.org/10.1007/3-540-63141-0_10)
3. Bouajjani, A., Esparza, J., Touili, T.: 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 2003, pp. 62–73, New York. ACM (2003)
4. Bouajjani, A., Müller-Olm, M., Touili, T.: Regular symbolic analysis of dynamic networks of pushdown systems. In: Abadi, M., de Alfaro, L. (eds.) CONCUR 2005. LNCS, vol. 3653, pp. 473–487. Springer, Heidelberg (2005). [https://doi.org/10.1007/11539452\\_36](https://doi.org/10.1007/11539452_36)
5. Chaki, S., Clarke, E., Kidd, N., Reps, T., Touili, T.: Verifying concurrent message-passing C programs with recursive calls. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 334–349. Springer, Heidelberg (2006). [https://doi.org/10.1007/11691372\\_22](https://doi.org/10.1007/11691372_22)

6. Esparza, J., Hansel, D., Rossmanith, P., Schwoon, S.: Efficient algorithms for model checking pushdown systems. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 232–247. Springer, Heidelberg (2000). [https://doi.org/10.1007/10722167\\_20](https://doi.org/10.1007/10722167_20)
7. Pommellet, A., Touili, T.: Static analysis of multithreaded recursive programs communicating via Rendez-Vous. In: Chang, B.-Y.E. (ed.) APLAS 2017. LNCS, vol. 10695, pp. 235–254. Springer, Cham (2017). [https://doi.org/10.1007/978-3-319-71237-6\\_12](https://doi.org/10.1007/978-3-319-71237-6_12)
8. Qadeer, S., Rehof, J.: Context-bounded model checking of concurrent software. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 93–107. Springer, Heidelberg (2005). [https://doi.org/10.1007/978-3-540-31980-1\\_7](https://doi.org/10.1007/978-3-540-31980-1_7)
9. Song, F., Touili, T.: Efficient CTL model-checking for pushdown systems. *Theor. Comput. Sci.* **549**, 127–145 (2014)
10. Song, F., Touili, T.: Model-checking dynamic pushdown networks. *Form. Asp. Comput.* **27**(2), 397–421 (2015)
11. La Torre, S., Madhusudan, P., Parlato, G.: A robust class of context-sensitive languages. In: 22nd Annual IEEE Symposium on Logic in Computer Science (LICS 2007), pp. 161–170, July 2007