Li X, Gardy P, Deng YX *et al.* Reachability of patterned conditional pushdown systems. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 35(6): 1295–1311 Nov. 2020. DOI 10.1007/s11390-020-0541-z

Reachability of Patterned Conditional Pushdown Systems

Xin Li[1], Patrick Gardy[1], Yu-Xin Deng[1,*], and Hiroyuki Seki[2]

[1]*Shanghai Key Laboratory of Trustworthy Computing, East China Normal University, Shanghai 200062, China*
[2]*Graduate School of Informatics, Nagoya University, Nagoya 464-8601, Japan*

E-mail: xinli@sei.ecnu.edu.cn; patrick.gardy@gmail.com; yxdeng@sei.ecnu.edu.cn; seki@i.nagoya-u.ac.jp

Received April 11, 2020; revised October 9, 2020.

Abstract Conditional pushdown systems (CPDSs) extend pushdown systems by associating each transition rule with a regular language over the stack alphabet. The goal is to model program verification problems that need to examine the runtime call stack of programs. Examples include security property checking of programs with stack inspection, compatibility checking of HTML5 parser specifications, etc. Esparza *et al.* proved that the reachability problem of CPDSs is EXPTIME-complete, which prevents the existence of an algorithm tractable for all instances in general. Driven by the practical applications of CPDSs, we study the reachability of patterned CPDS (pCPDS) that is a practically important subclass of CPDS, in which each transition rule carries a regular expression obeying certain patterns. First, we present new saturation algorithms for solving state and configuration reachability of pCPDSs. The algorithms exhibit the exponential-time complexity in the size of atomic patterns in the worst case. Next, we show that the reachability of pCPDSs carrying simple patterns is solvable in fixed-parameter polynomial time and space. This answers the question on whether there exist tractable reachability analysis algorithms of CPDSs tailored for those practical instances that admit efficient solutions such as stack inspection without exception handling. We have evaluated the proposed approach, and our experiments show that the pattern-driven algorithm steadily scales on pCPDSs with simple patterns.

Keywords conditional pushdown system, pattern, reachability, saturation algorithm

1 Introduction

Pushdown systems (PDSs) are widely used as abstract models for sequential programs with recursion. A PDS is formed by a finite set of states, a finite yet unbounded stack, and transition rules. The stack can be used to store the calling contexts of programs, whereby method calls and returns are correctly matched when encoding the programs as PDSs in program analysis and verification. Thanks to efficient saturation-based algorithms for computing the post-images and pre-images of (possibly infinite) regular sets of system states[1], quite a few pushdown model checking tools have been developed, notably Moped[1], jMoped[2], SLAM[3], Weighted PDS[4], PuMoC[5], PDSolver[6], etc., and these tools have been successfully applied to analyze, test and verify Boolean, Java, C/C++, and Erlang programs.

1.1 Motivation

PDSs have been extended in various ways to model concurrent programs[7,8], real-time system behaviors[9,10], etc. A line of research extends PDSs with the power of manipulating the stack, driven by practical analysis and verification problems that need to investigate or modify the runtime call stack of programs. To check global security properties of programs with stack inspection, Esparza *et al.* introduced PDSs with checkpoints[11] as well as model checking algorithms. The model was later reformulated as conditional PDSs (CPDSs) in order to precisely model

Regular Paper

Special Section on Software Systems 2020

This work was supported by the National Natural Science Foundation of China under Grant Nos. 61802126, 61672229, 61832015 and 62072176, the Ministry of Science and Technology of the People's Republic of China under Grant No. 2018YFC0830400, and Shanghai Pujiang Program under Grant No. 17PJ1402200.

*Corresponding Author

©Institute of Computing Technology, Chinese Academy of Sciences 2020

object-oriented features and perform context-sensitive points-to analysis for Java [12]. CPDSs extend PDSs by associating each transition rule with a regular language over the stack alphabet (often represented by a regular expression), such that a transition rule can be used in pushdown computation if the current stack word is accepted by the specified regular language. In recent years a few more interesting applications of CPDSs have been studied [13, 14]. Unfortunately, the reachability problem of CPDSs is computationally intractable in general. In [11], Esparza *et al.* proved that the reachability problem of PDSs with checkpoints, even for those having three states and no negative rules①, is EXPTIME-complete. It follows that the reachability problem of CPDSs is also EXPTIME-complete, and thus one could only hope that application instances of CPDSs in practice are not pathological.

Nevertheless, we observe that there are practical instances of CPDSs that admit tractable solutions. A notable example is security property verification of programs with stack inspection that has been extensively studied. By inserting Java API $checkPermission(p)$ in the program, runtime access control will be triggered during the program execution when the API is invoked, and the current call stack is inspected on whether the methods in the stack hold the expected permission p. In [15], Nitta *et al.* studied a subclass of programs with stack inspection where exception handling is excluded when access control fails. They showed that the class can be solved efficiently and sometime linear in the size of a program, given that the regular language involved in the analysis is represented by deterministic finite automata (DFA). Another example is context-sensitive points-to analysis for Java that has been actively studied over decades. Notably, such control-flow analysis (CFA) problems are characterized by mutual-dependency between data-flow and control-flow. There are several polynomial-time formalizations of the analysis problem, e.g., Bravenboer and Smaragdakis presented a declarative k-CFA [16]② Java points-to analysis algorithm in Datalog that can only express polynomial-time algorithms [17]. Also, extensive empirical studies show that most existing points-to analysis algorithms scale when $k \leqslant 2$ [18]. As aforementioned, a context-sensitive points-to analysis algorithm is presented for Java which is yielded as reachability analysis of CPDSs [12]. Therefore, there is a gap between the intractability results on CPDSs and tractable instances of the model.

1.2 Related Work

In addition to aforementioned earlier studies, there has been a great deal of recent work on extending PDSs with the power of manipulating the stack. Minamide and Mori formalized the HTML5 parser specification in an imperative programming language with commands for manipulating the entire stack, and reduced the compatibility checking of parser specifications to the reachability analysis of CPDSs [13]. They detected several compatibility issues in the implementations of web browsers with the complex HTML5 parser specifications. Vy and Li [19] presented an on-the-fly model checking algorithm for weighted CPDSs, by interleaving saturation algorithms with regular condition checking in a demand-driven manner. By experimenting with the application of HTML5 compatibility checking, they showed that the on-the-fly algorithm drastically outperforms the offline algorithm [19]. The problem of pushdown flow analysis with abstract garbage collection is discussed in the framework of CPDSs, yet the problem is not directly solved by using CPDSs due to the dynamic nature of garbage collection [14]. Abdulla *et al.* introduced discrete timed PDSs that combine PDSs and timed automata where stack symbols are extended with the notion of ages and can be incremented by transitions [9]. As a generalization of these extensions, Uezato and Minamide [20] introduced PDSs with transductions (TrPDSs) that associate each transition rule with a transduction whereby the entire stack content can be modified. Since the model is Turing-complete in general, they considered finite TrPDSs for which the closure of transductions appearing in the transitions of a TrPDS is finite and the reachability problem is decidable [20]. Later, Song *et al.* proposed new saturation algorithms for efficient reachability analysis of TrPDSs [21]. The aforementioned algorithms can be directly or indirectly applied to solve the reachability problems of CPDSs, and generally fall into two categories. The algorithms in [11,12,19] take a detour to the reachability analysis of PDSs and suffer from the potential exponential blowup of the resulting PDSs. In contrast, the algorithms in [13,21] directly extend the saturation procedure of PDSs by augmenting $\mathcal{P}$-automata with regular lookahead or transductions.

①In the setting of CPDSs, negative rules are transitions that are applied when the current stack does not satisfy the regular condition.

②Roughly speaking, it refers to the calling contexts that are abstracted and distinguished in terms of the last k call sites visited.

1.3 Key Ideas

By examining the existing applications of CPDSs, we observe that they carry regular expressions that obey certain patterns. A pattern can be seen as a Boolean combination of atomic patterns formed by set union, intersection, and negation operators, where an atomic pattern is a regular expression in the form of A^* or $A^*\gamma_1 \dots \gamma_k\Gamma^*$, where Γ is the stack alphabet, $A \subseteq \Gamma$, $\gamma_i \in \Gamma$ for each $1 \leqslant i \leqslant k$, and $*$ is Kleene star. For example, the access control policy for stack inspection upon calling *checkPermission*(p) can be expressed by the following regular expression:

$$M_p{}^* \left(m_1 + \cdots + m_n\right) \Gamma^* + M_p{}^*,$$

where M_p denotes the set of methods possessing the required permission p, and each m_i $(1 \leqslant i \leqslant n)$ denotes a privileged method. It says that stack inspection succeeds if all methods in the stack possess the required permission p until a privileged method is found, and aborts the execution otherwise. If exception handling is further considered, then the negated format of the regular expression above is also considered. It indicates how the thread of control in the program changes when the access control policy is violated.

Let us consider another example. To check the compatibility of HTML5 parser specifications with their implementations in web browsers, a specification language is modelled as CPDSs and the specifications are expressed by regular expressions such as the following:

$$A^*\mathtt{LiRp}\Gamma^* \ \& \ ! \ \mathtt{P}\Gamma^*,$$

where $A = \{\mathtt{Div,Optgroup,Option,Ruby}\} \subseteq \Gamma$ and $\mathtt{Li},\ \mathtt{Rp},\ \mathtt{P} \in \Gamma$ are all HTML elements and modelled as stack symbols in the analysis. We refer interested readers to [13] for more details.

Therefore, we propose an approach to reachability analysis driven by these patterns. To avoid an immediate exponential blowup by taking a detour to the reachability analysis of PDSs, we take an approach that directly extends the saturation rules of $\mathcal{P}$-automata. Instead of inspecting the exact stack contents for applying CPDS transition rules, we keep tracking the set of atomic patterns (denoted by $[\![\omega]\!]$) that are satisfied by the current stack word $\omega \in \Gamma^*$. Recall that a pattern carried by each transition rule can be seen as a Boolean combination of atomic patterns. Then checking whether a stack word ω belongs to some pattern R amounts to determining whether R can be satisfied by setting each atomic pattern in R to be true if it is from $[\![\omega]\!]$ and false otherwise.

For this purpose, we introduce a new notion called signatures of the stack contents which play a central role in our approach, as given formally in Definition 4 (Subsection 4.2). Generally, the signature of a stack word $\omega \in \Gamma^*$ contains the set of atomic patterns that ω satisfies (i.e., $[\![\omega]\!]$), as well as extra information needed for tracking and generating these atomic patterns. Then we introduce Sig-$\mathcal{P}$-automata (denoted by $\mathcal{B}$) as an update version of ordinary $\mathcal{P}$-automata (Subsection 4.3), where each state q is augmented and paired with the signature of any language accepted by $\mathcal{B}$ that is read forward from state q. Then the classic saturation rules for ordinary $\mathcal{P}$-automata can be adapted to the context of Sig-$\mathcal{P}$-automata (Subsection 4.4). We determine whether a CPDS transition rule carrying some pattern R can be used for saturating a Sig-$\mathcal{P}$-automaton, by simply checking whether R can be satisfied by the set of atomic patterns carried by the state.

1.4 Contributions

Comparing these approaches and observations above, we introduce a practically important subclass of CPDS called Patterned CPDS (pCPDS), driven by practical instances of CPDS, and make a comprehensive study on the reachability problem of pCPDS. Each transition rule in pCPDSs carries a regular expression in certain shape called patterns, and these patterns are further categorized as simple if they are formed by basic or atomic patterns and union operators, and complete, otherwise. Despite of syntactic constraints, pCPDS is expressive to model the existing applications of CPDSs discovered in practice.

Then we propose new algorithms to solve the state and configuration reachability problems of pCPDSs. To avoid the immediate exponential blowup caused by translating the reachability problem of CPDSs to that of PDS, the new algorithms directly extend the saturation rules for ordinary PDS to the setting of pCPDSs (Section 4). Although the algorithms still exhibit an exponential time and space complexity in the size of atomic patterns in the worst case, we show that the reachability problem of pCPDSs that carry simple patterns is solvable in fixed-parameter polynomial time and space. The result is in accordance with the existing tractable solutions tailored for the aforementioned applications of CPDSs. When the set of target configurations is finitely patterned as always found in practice, our pattern-driven approach permits further optimizations (Section 5). We have implemented and evaluated the proposed reachability checking algorithms. Our preliminary experiments show that the pattern-driven approach scales much better than the on-the-fly reachability checking algorithm for pCPDSs with simple patterns [19].

2 Preliminaries

Definition 1 (CPDS [11,12]). *A conditional pushdown system (CPDS) $\mathcal{P}_c$ is a 4-tuple $(P, \Gamma, \mathcal{C}, \Delta_c)$, where P is a finite set of control states, Γ is a finite stack alphabet, $\mathcal{C}$ is a finite set of regular languages over Γ, and $\Delta_c \subseteq P \times \Gamma \times \mathcal{C} \times P \times \Gamma^*$ is a finite set of transition rules. We write $\langle p, \gamma\rangle \overset{L}{\hookrightarrow} \langle q, \omega\rangle$ if $(p, \gamma, L, q, \omega) \in \Delta_c$. A configuration is a pair $\langle q, \omega\rangle \in P \times \Gamma^*$ of control state and stack content. A binary relation $\Rightarrow_c$ on configurations is defined by letting $\langle p, \gamma\omega'\rangle \Rightarrow_c \langle q, \omega\omega'\rangle$ for all $\omega' \in \Gamma^*$ if $\langle p, \gamma\rangle \overset{L}{\hookrightarrow} \langle q, \omega\rangle$ and $\omega' \in L$. Let $\Rightarrow_c^*$ be the reflexive and transitive closure of $\Rightarrow_c$. Given a set of configurations $S \subseteq P \times \Gamma^*$, we define $Pre^*(S) = \{s' \in P \times \Gamma^* \mid \exists s \in S : s' \Rightarrow_c^* s\}$ and $Post^*(S) = \{s' \in P \times \Gamma^* \mid \exists s \in S : s \Rightarrow_c^* s'\}$ to be the (possibly infinite) set of predecessors and successors of S, respectively. We denote by $pre^*(S)$ and $post^*(S)$ the set of predecessors and successors of S computed by the underlying unconditioned PDS, respectively.*

If $L = \Gamma^*$ for a transition rule $(p, \gamma, L, q, \omega) \in \Delta_c$, then this rule can be used under any condition when deriving the computation relation $\Rightarrow_c^*$ and it becomes an ordinary transition rule in PDSs. Thus PDSs are special instances of CPDSs where each transition rule carries the regular expression Γ^*. Following the tradition of PDSs, we assume a normalized form of CPDSs where $|\omega| \leqslant 2$ for each rule $\langle p, \gamma\rangle \overset{L}{\hookrightarrow} \langle q, \omega\rangle$. The fundamental reachability problem of CPDSs asks given a CPDS, whether a given configuration $\langle p', \omega'\rangle$ is reachable from another given configuration $\langle p, \omega\rangle$, i.e., does $\langle p, \omega\rangle \Rightarrow_c^* \langle p', \omega'\rangle$ hold?

Theorem 1 [11]. *The reachability problem of CPDSs is EXPTIME-complete given that the regular language associated with each transition rule in CPDSs is represented by a deterministic finite automaton.*

By Theorem 1, there do not exist computationally tractable algorithms in general for tackling the reachability problems of CPDSs. In [11], Esparza *et al.* presented an algorithm that translates a CPDS $\mathcal{P}_\mathcal{C}$ to a corresponding PDS whereby the reachability problem of CPDSs is reduced to that of PDSs. Let $\mathcal{C} = \{L_1, \ldots, L_n\}$. For every L_i ($i \in [1..n]$), let A_i be the DFA that recognizes L_i^R — the reverse of L_i. Let $\Pi_{i\in[1..n]}A_i = (States, \Gamma, \delta, s_0, F)$ be the cartesian product automaton $A_1 \times \ldots \times A_n$. Note that, $\Pi_{i\in[1..n]}A_i$ is also a DFA since each A_i is a DFA. Below we recall the translation of CPDSs to PDSs, of which the key insight is to synchronize the two machineries of $\Pi_{i\in[1..n]}A_i$ and the underlying pushdown system of $\mathcal{P}_\mathcal{C}$. For any transition rule in Δ_c on the left-hand side and for each state $\mathbf{r} \in States$, one would obtain the new transition rules of the resulting PDSs on the right-hand side:

$$\begin{array}{lcl}
\langle p, \gamma\rangle \overset{L}{\hookrightarrow} \langle q, \varepsilon\rangle & & \langle p, (\gamma, \boldsymbol{r})\rangle \hookrightarrow \langle q, \varepsilon\rangle \\
\langle p, \gamma\rangle \overset{L}{\hookrightarrow} \langle q, \gamma'\rangle & \Longrightarrow & \langle p, (\gamma, \boldsymbol{r})\rangle \hookrightarrow \langle q, (\gamma', \boldsymbol{r})\rangle \\
\langle p, \gamma\rangle \overset{L}{\hookrightarrow} \langle q, \gamma'\gamma''\rangle & & \langle p, (\gamma, \boldsymbol{r})\rangle \hookrightarrow \langle q, (\gamma', \boldsymbol{t})(\gamma'', \boldsymbol{r})\rangle
\end{array}$$

where $\delta(\boldsymbol{r}, \gamma'') = \boldsymbol{t}$. Then efficient algorithms of the pushdown model checking can be applied to solving the reachability problems of CPDSs. The size of the resulting pushdown system is $|\Delta_c| \times |States|$ since the product automaton $\Pi_{i\in[1..n]}A_i$ is deterministic. One may optimize the translation above by constructing a minimized DFA for each regular expression L_i. Then the product automaton will be built upon a smaller set of distinguished DFA denoted by $\mathcal{M}$ ($|\mathcal{M}| \leqslant n$) in which $\mathcal{L}(A) \neq \mathcal{L}(A')$ for any A, $A' \in \mathcal{M}$. Let $N = \max_{A\in\mathcal{M}} |A|$. Then $|States| = O(N^{|\mathcal{M}|})$ can be exponentially large in the number of distinguished regular conditions. Therefore, the size of the resulting PDS could explode right after the first step.

At the heart of efficient pushdown model checkers are saturation-based algorithms over $\mathcal{P}$-automata. Consider a PDS $\mathcal{P} = (P, \Gamma, \Delta)$. Let $\Gamma_\varepsilon = \Gamma \cup \{\varepsilon\}$. A $\mathcal{P}$-automaton $\mathcal{A}$ for $\mathcal{P}$ is a non-deterministic finite automaton (NFA) $(Q, \Gamma_\varepsilon, \rightarrow, P, F)$ that finitely represents a possibly infinite set of configurations, where Q is the set of states, $\rightarrow \subseteq Q \times \Gamma_\varepsilon \times Q$ is the set of transitions, and P and F are the sets of initial and final states, respectively. A configuration $\langle p, \omega\rangle$ is accepted by $\mathcal{A}$ if ω is accepted by A from the initial state p. A set C of configurations is accepted by $\mathcal{A}$ iff $\mathcal{L}(\mathcal{A}) = C$. A set C of configurations is regular if it is accepted by some $\mathcal{P}$-automaton. Given a $\mathcal{P}$-automaton $\mathcal{A}$ representing some regular set of configurations, by iteratively applying saturation rules and enlarging $\mathcal{A}$, the saturation methods build upon termination either $post^*(\mathcal{A})$ — the sets of reachable configurations from $\mathcal{A}$ in $\mathcal{P}$, or $pre^*(\mathcal{A})$ — the set of configurations that can reach $\mathcal{A}$ in $\mathcal{P}$. We consider regular sets of source configurations and target configurations in $\mathcal{P}$ represented in two $\mathcal{P}$-automata $\mathcal{A}_S$

and $\mathcal{A}_T$, respectively. To solve the reachability problem between source and target configurations, one way is to saturate $\mathcal{A}_S$ into $post^*(\mathcal{A}_S)$, cross $\mathcal{A}_T$ with $post^*(\mathcal{A}_S)$, and solve the emptiness problem of the resulting automaton. As aforementioned, the resulting PDS translated from CPDS would explode even though the saturation procedure for PDSs above is efficient.

3 Patterned Conditional Pushdown Systems

3.1 Problems to Be Addressed

Definition 2 (Pattern). *An atomic pattern denoted by β is a regular expression in the form of A^* or $A^*\gamma_1 \ldots \gamma_k\Gamma^*$ where $A \subseteq \Gamma$ and $\gamma_1, \ldots, \gamma_k \in \Gamma$ $(k > 0)$. The set of atomic patterns is denoted by $\mathfrak{P}$. Further, let $\mathfrak{P}_1(\subseteq \mathfrak{P})$ be the set of atomic patterns in the form of A^*, and let $\mathfrak{P}_2(\subseteq \mathfrak{P})$ be the set of atomic patterns in the form of $A^*\gamma_1 \ldots \gamma_k\Gamma^*$. A pattern R is a regular expression formed by atomic patterns. We define two kinds of simple and complete patterns as follows*:

$$(Simple)\ \mathsf{Spat}_{\mathfrak{P}} \ni R ::= \beta \mid \beta + R$$
$$(Complete)\ \mathsf{Cpat}_{\mathfrak{P}} \ni R ::= \beta \mid !R \mid R + R \mid R\&R.$$

The operator "!" denotes negation (or complementation), "&" denotes intersection, and " + " denotes union. Obviously, $\mathsf{Spat}_{\mathfrak{P}} \subseteq \mathsf{Cpat}_{\mathfrak{P}}$. Alternatively, we write Pat *for the set of complete patterns assuming $\mathfrak{P}$ is given without ambiguity. We write $\mathcal{L}(R)$ for the regular language that R represents.*

Parameterized by $\mathfrak{P}$, a simple pattern can be seen as a disjunction of patterns in $\mathfrak{P}$. Similarly, a complete pattern can be seen as Boolean combination of patterns in $\mathfrak{P}$, if we regard operators !, & and + as logic negation, conjunction and disjunction, respectively. In the sequel, we omit parameter $\mathfrak{P}$ when it is clear from the context.

Definition 3 (Patterned CPDS). *Assuming a set of atomic patterns $\mathfrak{P}$, a patterned CPDS (pCPDS) (parameterized by $\mathfrak{P}$) $\mathcal{P}_c\langle\mathfrak{P}\rangle$ is a CPDS $(P, \Gamma, \mathcal{C}, \Delta_c)$ where $\mathcal{C} = \{R_1, \ldots, R_m\}$ $(\subseteq \mathsf{Cpat}_{\mathfrak{P}})$ is a finite set of patterns. A patterned configuration (p, R) is a regular set $\{\langle p, \omega\rangle \mid \omega \in R\}$ of configurations for a control location $p \in P$ and a pattern $R \in \mathsf{Cpat}_{\mathfrak{P}}$. We trivially extend the definition of simple and complete patterns to simple and complete pCPDS depending on the type of patterns involved in $\mathcal{C}$. Let K be the maximal k appearing in atomic patterns of shape $A^*\gamma_1 \ldots \gamma_k\Gamma^*$ in $\mathcal{C}$.*

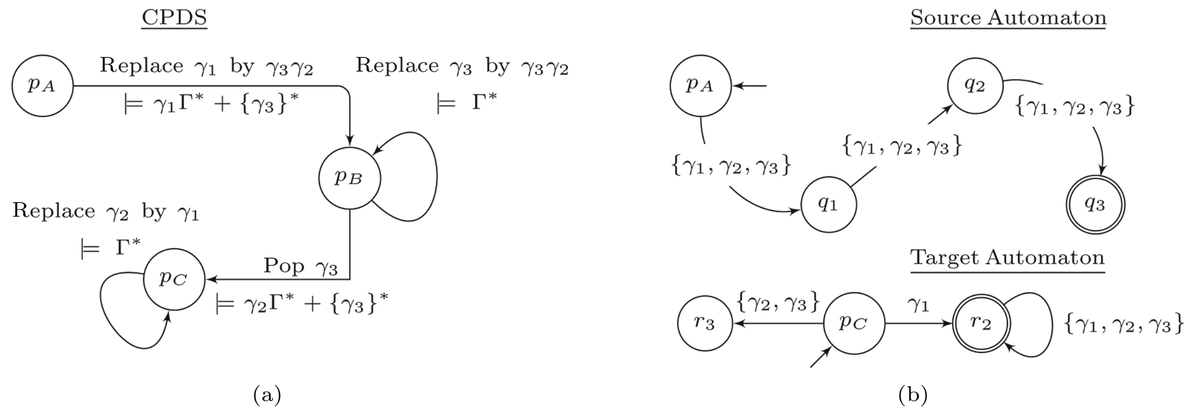
For the rest of the paper, we assume given a complete CPDS $\mathcal{P}_\mathcal{C} = (P, \Gamma, \mathcal{C}, \Delta_c)$, a regular set of source configurations $S \subseteq P \times \Gamma^*$, and a regular set of target configurations $T \subseteq P \times \Gamma^*$. The following problems of CPDSs are to be addressed.

• State reachability is a given control location $p \in P$ forward (resp. backward) reachable from S, i.e., does there exist $\omega \in \Gamma^*$ such that $\langle p, \omega\rangle \in Post^*(S)$ (resp. $\langle p,\ \omega\rangle \in Pre^*(S)$)?

• Configuration reachability is T reachable from S, i.e., does $T \cap Post^*(S) \neq \emptyset$, or equivalently, does $S \cap Pre^*(T) \neq \emptyset$ hold?

Also, we observe that in practice the source and target configurations S and T above are often a finite union of patterned configurations. As discussed later (Subsection 4.4), if we assume that either S or T satisfies this assumption, then we are able to simplify the last step in our new reachability algorithm.

Example 1. Fig.1 shows the running example to be used for illustrating our approach. Consider the pCPDS $\mathcal{P}_c$ with three control states $\{p_A, p_B, p_C\}$ and the stack alphabet $\Gamma = \{\gamma_1, \gamma_2, \gamma_3\}$. The set Δ_c of transitions is shown in Fig.1(a). In a transition, the first line repre-


CPDS
p_A
p_B
p_C
Replace γ_1 by $\gamma_3\gamma_2$
$\models \gamma_1\Gamma^* + \{\gamma_3\}^*$
Replace γ_3 by $\gamma_3\gamma_2$
$\models \Gamma^*$
Replace γ_2 by γ_1
$\models \Gamma^*$
Pop γ_3
$\models \gamma_2\Gamma^* + \{\gamma_3\}^*$
(a)
Source Automaton
p_A
q_1
q_2
q_3
$\{\gamma_1, \gamma_2, \gamma_3\}$
$\{\gamma_1, \gamma_2, \gamma_3\}$
$\{\gamma_1, \gamma_2, \gamma_3\}$
Target Automaton
r_3
p_C
r_2
$\{\gamma_2, \gamma_3\}$
γ_1
$\{\gamma_1, \gamma_2, \gamma_3\}$
(b)
Fig.1. (a) pCPDS p_C. (b) p_C's source and target $\mathcal{P}$-automata.

sents the stack operation and the second line following $\models$ represents the regular condition. The source and the target $\mathcal{P}$-automata are given in Fig.1(b) where the initial states are marked with an incoming arrow.

3.2 Application: Points-to Analysis for Java

Most practical applications of pCPDSs fall under the cloud of either simple pCPDSs or complete pCPDSs. For instance, the compatibility checking of HTML5 parser specifications is an instance of complete CPDSs [13]. The security property checking of programs with stack inspection and without exception handling is an instance of simple pCPDSs with $k = 1$ [15]. Below we illustrate the encoding of Java points-to analysis as the reachability analysis of simple pCPDSs.

Due to object-oriented features like polymorphism and late binding, the call graphs of programming languages like Java would not be decided at compile time in general. For example, to resolve the method invocation $x.f()$ in Java, Java virtual machine (JVM) relies on the runtime type of the object that x refers to. Therefore, points-to analysis and precise call graph construction algorithms are mutually-dependent on each other. An approach is presented in [12] to yield a context-sensitive Java points-to analysis by modelling it as a model checking problem of weighted CPDSs. Since the possibly infinite set of heap objects is abstracted to be finite in terms of their allocation sites, the problem can be equivalently modelled as model checking problems of CPDSs. An example is given below that is borrowed and simplified from Fig.2 in [12]. We show that a context-sensitive points-to analysis (or equivalently call graph construction) algorithm can be yielded as the reachability analysis of simple pCPDSs.

Fig.2 shows a code snippet that defines two classes A and B such that B inherits from A. In the main method, two dynamic objects of A and B are created and passed to a method *foo* for performing some operations. One may be concerned with issues such as whether variables c and d are aliased, or whether downcasts at line 23 and 26 are safe. The questions boil down to whether c and d refer to an integer object and a string object, respectively. To answer the questions, we model points-to

```
 1  public class A {
 2    Object f;
 3    public Object set() {
 4      this.f = new Integer(5);
 5      return this.f;
 6    }
 7  }
 8
 9  public class B extends A {
10    public Object set() {
11      this.f = new String();
12      return this.f;
13    }
14  }
15
16  public class Main {
17    public static Object foo(A x) {
18      return x.set();
19    }
20    public static void main(String[] args) {
21      A a = new A();
22      Object c = foo(a);
23      Integer i = (Integer) c;
24      A b = new B();
25      Object d = foo(b);
26      String s = (String) d;
27      ......
28    }
29  }
```

(a)

1) Part of Transitions for $A.set()$

$r_{4\text{-}1} : \langle \Lambda,\ \texttt{A.set} \rangle \hookrightarrow \langle o_4,\ \texttt{A.set} \rangle$

$r_{4\text{-}2} : \langle o_4,\ \texttt{A.set} \rangle \hookrightarrow \langle f_{\texttt{A.set}},\ \texttt{A.set} \rangle$

$r_5 : \langle f_{\texttt{A.set}},\ \texttt{A.set} \rangle \hookrightarrow \langle ret_{\texttt{A.set}},\ \texttt{A.set} \rangle$

$r_{13} : \langle ret_{\texttt{A.set}},\ \texttt{A.set} \rangle \hookrightarrow \langle ret_{\texttt{A.set}},\ \varepsilon \rangle$

2) Part of Transitions for $foo()$

$r_{18\text{-}1} : \langle \Lambda,\ \texttt{foo} \rangle \overset{R_1}{\hookrightarrow} \langle \Lambda,\ \texttt{A.set l}_{18} \rangle$

$r_{18\text{-}2} : \langle \Lambda,\ \texttt{foo} \rangle \overset{R_2}{\hookrightarrow} \langle \Lambda,\ \texttt{B.set l}_{18} \rangle$

$r_{18\text{-}3} : \langle ret_{\texttt{A.set}},\ \texttt{foo} \rangle \hookrightarrow \langle ret_{\texttt{foo}},\ \texttt{foo} \rangle$

$r_{18\text{-}4} : \langle ret_{\texttt{B.set}},\ \texttt{foo} \rangle \hookrightarrow \langle ret_{\texttt{foo}},\ \texttt{foo} \rangle$

$r_{19} : \langle ret_{\texttt{foo}},\ \texttt{foo} \rangle \hookrightarrow \langle ret_{\texttt{foo}},\ \varepsilon \rangle$

3) Part of Transitions for $main()$

$r_{21\text{-}1} : \langle \Lambda,\ \texttt{main} \rangle \hookrightarrow \langle o_{21},\ \texttt{main} \rangle$

$r_{21\text{-}2} : \langle o_{21},\ \texttt{main} \rangle \hookrightarrow \langle a,\ \texttt{main} \rangle$

$r_{22\text{-}1} : \langle a,\ \texttt{main} \rangle \hookrightarrow \langle x,\ \texttt{foo l}_{22} \rangle$

$r_{22\text{-}2} : \langle \Lambda,\ \texttt{main} \rangle \hookrightarrow \langle \Lambda,\ \texttt{foo l}_{22} \rangle$

$r_{22\text{-}3} : \langle ret_{\texttt{foo}},\ \texttt{l}_{22} \rangle \hookrightarrow \langle c,\ \texttt{main} \rangle$

$r_{23} : \langle c,\ \texttt{main} \rangle \hookrightarrow \langle i,\ \texttt{main} \rangle$

(b)

Fig.2. (a) Java code snippet for illustrating the application of simple pCPDSs to context-sensitive points-to analysis, with (b) its encoding as conditional pushdown transition rules given, where $R_1 = \texttt{l}_{22}\Gamma^*$ and $R_2 = \texttt{l}_{25}\Gamma^*$, and the regular condition carried by transition rules is omitted if it is Γ^*; each transition rule is numbered by $r_{i\text{-}j}$ where i is the line number of the encoded program and j numbers all resulting transitions for the i-th line; Λ denotes the dynamic heap environment; o_i denotes the abstract heap object allocated at the i-th line; $ret_{\texttt{func}}$ represents the distinguished return variable of method *func*.

analysis as model checking problems of simple pCPDSs. As given in Fig.2(b), the rule $r_{4\text{-}1}$ mimics creating object o_4 on the heap, and the rule $r_{4\text{-}2}$ models that o_4 flows to the reference variable $this.f$ in $A.set()$. The rule r_{13} models the moment when the program execution exits the scope of $A.set()$ and the method is popped out. The encoding of $main()$ is similar except that extra transition rule $r_{22\text{-}2}$ is introduced to model the thread of program control changes at line 22. Considering the call graph is constructed on-the-fly, the encoding of $foo()$ relies on the runtime type of x to decide the method invocation at line 18.

Further, let $pta(x, \texttt{cxt})$ denote the set of objects pointed-to by x under the calling contexts cxt. Here, a calling context of x in some method m is defined as $\texttt{m l}_\texttt{n} \ \ldots \ \texttt{l}_1$, where $\texttt{l}_1, \ldots, \texttt{l}_\texttt{n}$ is the sequence of call sites (in terms of line number) for method invocations leading to the current method m. It can be solved as the following reachability problem of CPDSs:

$$pta(x, \texttt{cxt}) \overset{def}{=} \{o \mid \exists\omega \in \Gamma^*.\langle \Lambda,\ \texttt{main}\rangle \Rightarrow_c^* \langle o,\ \omega\rangle \Rightarrow_c^* \langle x,\ \texttt{cxt}\rangle\}.$$

That is, $pta(x, \texttt{cxt})$ gives the set of objects that are backward reachable from $\{\langle x,\ \texttt{cxt}\rangle\}$. Now we are ready to query the points-to sets of x in $foo()$ whereby $r_{18\text{-}1}$ and $r_{18\text{-}2}$ are encoded. Because $pta(x, \texttt{foo l}_{22}) = \{o_4\}$, $A.set()$ is called under this calling context and we associate the regular condition $R_1 = \texttt{l}_{22}\Gamma^*$ with the rule $r_{18\text{-}1}$. Similarly, we associate the regular condition $R_2 = \texttt{l}_{25}\Gamma^*$ with rule $r_{18\text{-}2}$. Note that, the analysis will confuse the two variables c and d if regular conditions are removed from the transition system.

In general, suppose that x points to an object of type A under a calling context cxt, and consider Shivers' k-CFA context-sensitivities. A transition rule encoding this method invocation will carry a union of regular expressions in the form of $l_k \ldots l_1 \Gamma^*$, where l_i $(1 \leqslant i \leqslant k)$ denotes the last k call sites before calling $A.f()$ for a given k. It says that the stack has to comply with the calling contexts of the receiver object on which the method is invoked. This is an instance of simple pCPDSs for a given parameter k.

4 Saturation Algorithms for pCPDSs

4.1 Overall Approach

$\mathcal{P}$-automata $\mathcal{A}_S$ and $\mathcal{A}_T$ are for recognizing source and target configurations S and T, respectively. Both forward and backward saturation can be used to solve the reachability problem of pCPDS. We use the forward saturation procedure to describe our approach. The backward saturation algorithm works similarly. The reachability problems of pCPDSs are solved by our method in three major steps as follows.

Step 1. An algorithm is presented. It takes as input the $\mathcal{P}$-automaton $\mathcal{A}_S$ ($\mathcal{A}$ for short if no confusion would occur) and builds up a Sig-$\mathcal{P}$-automaton $\mathcal{B}$ with $\mathcal{L}(\mathcal{B}) = S$ (see Algorithm 1 in Subsection 4.3).

Step 2. Saturation rules are adapted to Sig-$\mathcal{P}$-automata and iteratively applied to saturate $\mathcal{B}$ computed in the first step. Upon convergence, it outputs a Sig-$\mathcal{P}$-automaton $\mathcal{B}_{post^*}$ with $\mathcal{L}(\mathcal{B}_{post^*}) = Post^*(S)$ (see Subsection 4.4).

Step 3. We can solve the reachability problems.

• *State Reachability.* A control state p is forward reachable from S iff there exists an initial state p in $\mathcal{B}_{post^*}$.

• *Configuration Reachability.* T is reachable from S iff $\mathcal{B}_{post^*} \cap \mathcal{A}_T \neq \emptyset$ and unreachable otherwise.

In particular, if the target configuration $T = \bigcup_{i\in[1..n]}(p_i, R_i)$ is a finite union of patterned configurations, the last step for checking configuration reachability above can be further simplified: T is reachable from S iff there exist some $(p, R) \in T$ and an initial state p in $\mathcal{B}_{post^*}$ carrying a set J of atomic patterns such that R can be satisfied by J.

4.2 Generating Signatures of Stack Words

In this subsection we describe how to compute signatures of the stack words. Below we first define signatures for dealing with atomic patterns in the shape of A^* and $A^*\gamma_1 \ldots \gamma_k\Gamma^*$ (Definition 4). To generate atomic patterns of shape $A^*\gamma_1 \ldots \gamma_k\Gamma^*$, we need to keep track of the middle word letters $\gamma_1 \ldots \gamma_k$ as the stack grows. This forces us to augment a $\mathcal{P}$-automata state with not only atomic patterns satisfied by the stack but also the longest prefix of the stack within the length K.

Definition 4 (Signature). *Given a stack word $\omega \in \Gamma^*$, we write $\omega_{\leqslant K}$ for the longest prefix of ω within the length K. The signature $(J, v) \subseteq \mathfrak{P} \times \Gamma^{\leqslant K}$ of ω is a pair with $J = [\![\omega]\!]$ and $v = \omega_{\leqslant K}$. We write* Sig *for the set of signatures.*

The cornerstone of our analysis is to generate the signature of a stack word $\omega \in \Gamma^*$. We propose an inductive approach by the following function:

$$SigOf \ :: \ \Gamma^* \mapsto \mathsf{Sig}$$

$$SigOf(\omega) = \begin{cases} (\mathfrak{P}_1, \varepsilon), & \text{if } \omega = \varepsilon, \\ Update(SigOf(\omega'), \gamma), & \text{o.w. } \omega = \gamma\omega'. \end{cases}$$

The first case is the base case for $\omega = \varepsilon$. We can conclude that $[\![\varepsilon]\!] = \mathfrak{P}_1$ since ε belongs to any atomic pattern in the form of A^* $(\in \mathfrak{P}_1)$, and $\varepsilon_{\leqslant K} = \varepsilon$. Note that, ε is not accepted by another type of atomic patterns in the form of $A^*\gamma_1 \dots \gamma_k\Gamma^*$ $(k > 0)$ $(\in \mathfrak{P}_2)$. The second case is the inductive case for $\omega = \gamma\omega'$ that depends on an *Update* function for generating the signature of ω taking as inputs γ and the signature of ω'. The function computes the new signature of the stack after γ is pushed onto the top.

Recall that the signature of ω is a pair (J, v) with $J = [\![\omega]\!]$ and $v = \omega_{\leqslant K}$. It is easy to update the prefix v, and the core task is to update atomic patterns $[\![\omega]\!]$ based on $[\![\omega']\!]$ and γ. The atomic patterns satisfied by ω from $\mathfrak{P}$ would be either 1) inherited from $[\![\omega']\!]$, or 2) newly-added after reading the symbol γ. The two cases are handled through the following functions, respectively.

$$\begin{aligned}
&Inh \ :: \ 2^{\mathfrak{P}} \times \Gamma \mapsto \ 2^{\mathfrak{P}}\\
&Inh(J,\gamma) = J'\\
&\text{s.t.} \begin{cases}
\text{if } v(A \cup \{\gamma\})^* \in J, \text{ then } (A \cup \{\gamma\})^* \in J',\\
\text{if } (A \cup \{\gamma\})^*\gamma_1 \dots \gamma_k\Gamma^* \in J,\\
\qquad\qquad \text{then } (A \cup \{\gamma\})^*\gamma_1 \dots \gamma_k\Gamma^* \in J'.
\end{cases}
\end{aligned}$$

$$\begin{aligned}
&Gen \ :: \ \Gamma^{\leqslant K} \times \Gamma \mapsto \ 2^{\mathfrak{P}}\\
&Gen(v,\gamma) = J'\\
&\text{s.t. if } \exists 0 \leqslant j \leqslant K \ \text{ and } \exists A \subseteq \Gamma \text{ with } A^*\gamma v_{\leqslant j}\Gamma^* \in \mathfrak{P},\\
&\qquad\qquad\qquad\qquad \text{then } A^*v\gamma v_{\leqslant j}\Gamma^* \in J'.
\end{aligned}$$

Then we are ready to describe the update function as follows, and illustrate different cases handled by the function in Fig.3.

$$\begin{aligned}
&Update \ :: \ \mathsf{Sig} \times \Gamma \mapsto \ \mathsf{Sig}\\
&Update((J,v),\gamma) = \big(Inh(J,\gamma) \cup Gen(v,\gamma), v'\big)\\
&\qquad\qquad\qquad\qquad \text{s.t. } v' = (\gamma v)_{\leqslant K}.
\end{aligned}$$

Recall that $\mathcal{P}$-automata finitely represent the (regular set of) pushdown configurations, i.e., a set of pairs of control states and stack words. In Fig.3, we use a snapshot of some part of $\mathcal{P}$-automata to show how the update function works, where q and q' are automata states, and q' moves to q after reading the stack symbol γ_2. We would like to remark that, as shown in the figure, the function *Inh* always either preserves in the new signature of q an atomic pattern carried by q' or just abandons it. That is, it would never introduce new atomic patterns into the signature carried by q. In contrast, the function *Gen* checks if any new atomic patterns can be added to the signature of q after reading a new stack symbol. The correctness of our method for generating signatures is given in Lemma 1.

Lemma 1. *For any $\omega' \in \Gamma^*$, $SigOf(\omega') = ([\![\omega']\!], \omega'_{\leqslant K})$.*

Proof. By definition, if $\omega' = \varepsilon$, then $SigOf(\omega') = (\mathfrak{P}_1, \varepsilon)$ where $\mathfrak{P}_1 = [\![\varepsilon]\!]$ and $\varepsilon_{\leqslant K} = \varepsilon$. Otherwise, let $\omega' = \gamma\omega$ for $\gamma \in \Gamma$ and $\omega \in \Gamma^*$. We have the inductive hypothesis that $SigOf(\omega) = ([\![\omega]\!], \omega_{\leqslant K})$. Then it amounts to proving that $Update(([\![\omega]\!], \omega_{\leqslant K}), \gamma) = ([\![\gamma\omega]\!], (\gamma\omega)_{\leqslant K})$.

Let $(J', v') = Update(([\![\omega]\!], \omega_{\leqslant K}), \gamma)$. Then $v' \overset{\text{def}}{=} (\gamma\omega_{\leqslant K})_{\leqslant K}$ by definition.

First, we show that $v' = (\gamma\omega)_{\leqslant K}$. By case analysis of the length of ω:

• $|\omega| < K$: we have that $\omega_{\leqslant K} = \omega$ and $(\gamma\omega)_{\leqslant K} = (\gamma\omega_{\leqslant K})_{\leqslant K} = \gamma\omega$;

• $|\omega| \geqslant K$: let $\omega = \gamma_1 \dots \gamma_K \dots \gamma_n$ for some $n \geqslant K$. Then we have that $\omega_{\leqslant K} = \gamma_1 \dots \gamma_K$, and $(\gamma\omega)_{\leqslant K} = (\gamma\omega_{\leqslant K})_{\leqslant K} = \gamma\gamma_1 \dots \gamma_{K-1}$.

Then we show that $J' = [\![\gamma\omega]\!]$.

Considering a pattern $\beta \in J'$, we show that $\beta \in [\![\gamma\omega]\!]$, i.e., $J' \subseteq [\![\gamma\omega]\!]$. By definition of the update function,

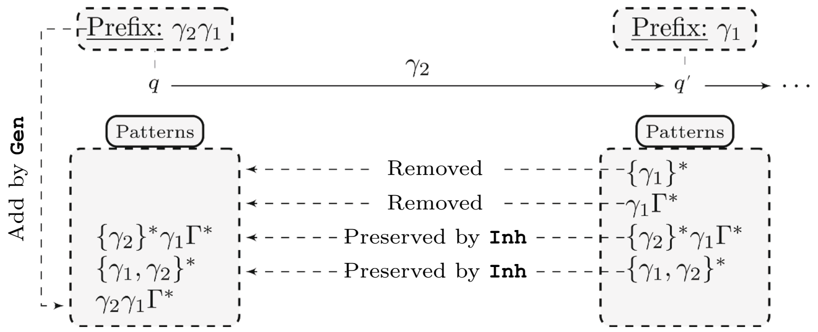

Prefix: $\gamma_2\gamma_1$
Prefix: γ_1
q
γ_2
q'
...
Add by Gen
Patterns
Patterns
Removed
Removed
Preserved by Inh
Preserved by Inh
$\{\gamma_2\}^*\gamma_1\Gamma^*$
$\{\gamma_1,\gamma_2\}^*$
$\gamma_2\gamma_1\Gamma^*$
$\{\gamma_1\}^*$
$\gamma_1\Gamma^*$
$\{\gamma_2\}^*\gamma_1\Gamma^*$
$\{\gamma_1,\gamma_2\}^*$

Fig.3. Illustrating different cases handled by the update function, where $K = 2$ and $\mathfrak{P}$ consists of the following basic patterns: $\{\gamma_1\}^*$, $\gamma_1\Gamma^*$, $\{\gamma_2\}^*\gamma_1\Gamma^*$, $\{\gamma_1, \gamma_2\}^*$, and $\gamma_2\gamma_1\Gamma^*$.

• either β is generated through the *Gen* function and has the shape of $A^*\gamma\omega_{\leqslant j}\Gamma^*$ for $0 \leqslant j \leqslant K$; trivially, ω satisfies $\omega_{\leqslant j}\Gamma^*$ and thus $\gamma\omega$ satisfies $A^*\gamma\omega_{\leqslant j}\Gamma^*$, i.e., $\beta \in [\![\gamma\omega]\!]$;

• or β was generated by *Inh* as follows:

1) if $\beta = (A \cup \{\gamma\})^*$ for some $A \subseteq \Gamma$ with $A^* \in [\![\omega]\!]$, then w satisfies A^* and $\gamma\omega$ satisfies $(\{\gamma\} \cup A)^*$, i.e., $\beta \in [\![\gamma\omega]\!]$;

2) if $\beta = (A \cup \{\gamma\})^*\gamma_1 \dots \gamma_k\Gamma^*$ for some $A \subseteq \Gamma$ and $\gamma_i \in \Gamma (1 \leqslant i \leqslant k)$ with $A^*\gamma_1 \dots \gamma_k\Gamma^* \in [\![\omega]\!]$, then ω satisfies $A^*\gamma_1 \dots \gamma_k\Gamma^*$ and thus $\gamma\omega$ satisfies β, i.e., $\beta \in [\![\gamma\omega]\!]$.

Considering a pattern $\beta \in [\![\gamma\omega]\!]$, we show that $\beta \in J'$, i.e., $[\![\gamma\omega]\!] \subseteq J'$. By case analysis of the shape of β, we have the followings.

• If $\beta = A^*$ for some $A \subseteq \Gamma$, then trivially, $\gamma \in A$ and also $A^* \in [\![\omega]\!]$. Then by definition of the *Inh* function, $A^* \in J'$.

• If $\beta = A^*\gamma_1 \dots \gamma_k\Gamma^*$ with $A \subseteq \Gamma$ and $\gamma_i \in \Gamma$ $(1 \leqslant i \leqslant K)$, then $\gamma\omega = v\gamma_1 \dots \gamma_k v'$ for some $v, v' \in \Gamma^*$ and the letters in v belong to A.

1) If $v = \varepsilon$, then $\beta = A^*\gamma\gamma_2 \dots \gamma_k\Gamma^*$, and any such kind of basic patterns would be generated by the *Gen* function, and thus $\beta \in J'$.

2) Otherwise, $v \neq \varepsilon$ implies that $\gamma_1 \dots \gamma_k$ is a subsequence of ω. Trivially, $\beta \in [\![\omega]\!]$. Then by the *Inh* function, $\beta \in J'$. □

4.3 Augmenting $\mathcal{P}$-Automata with Signatures

We introduce Sig-$\mathcal{P}$-automata where each state carries the signature of stack words accepted by the automaton reading forward from the state.

Definition 5 (Sig-$\mathcal{P}$-Automaton). *A Sig-$\mathcal{P}$-automaton $\mathcal{B}$ for a pCPDS $\mathcal{P}_{\mathcal{C}} = (P, \Gamma, \mathcal{C}, \Delta_c)$ is an NFA $(Q_{\mathcal{B}}, \Sigma, \to_{\mathcal{B}}, I_{\mathcal{B}}, F_{\mathcal{B}})$ where $Q_{\mathcal{B}} \subseteq Q \times \mathsf{Sig}$ with $P \subseteq Q$ for some finite set Q, $\Sigma = \Gamma \cup \{\varepsilon\}$ is the input alphabet, $\to_{\mathcal{B}} \subseteq Q_{\mathcal{B}} \times \Sigma \times Q_{\mathcal{B}}$ is the finite set of transitions, $I_{\mathcal{B}} \subseteq (P \times \mathsf{Sig}) \cap Q_{\mathcal{B}}$ is the finite set of initial states, and $F_B \subseteq Q_{\mathcal{B}}$ is the finite set of final states. The following property holds on each state $q = (s, (J, v))$ in $\mathcal{B}$:*

$$(J, v) \text{ is the signature of } \omega \text{ for any } \omega \text{in } \mathcal{L}(\mathcal{B}, q). \quad (1)$$

We introduce the transition relation $\xrightarrow{\omega}{}^*$ for Sig-$\mathcal{P}$-automata. It is the smallest set of relations satisfying 1) $q \xrightarrow{\varepsilon}{}^* q$; 2) $q \xrightarrow{\gamma}{}^* q'$ if $(q, \gamma, q') \in \to_{\mathcal{B}}$; 3) $q \xrightarrow{\gamma\omega}{}^* q'$ if $q \xrightarrow{\gamma}{}^* q''$ and $q'' \xrightarrow{\omega}{}^* q'$.

Let $\mathcal{A}$ be a $\mathcal{P}$-automaton that recognizes the regular set S of source configurations. Algorithm 1 takes as input $\mathcal{A}$, and outputs a Sig-$\mathcal{P}$-automaton $\mathcal{B}$ that recognizes S. The main idea is to construct $\mathcal{B}$ such that there exists a simulation relation between $\mathcal{A}$ and $\mathcal{B}$. To this end, we prepare a mapping l from the states in $\mathcal{A}$ to the power set of states in $\mathcal{B}$ (lines 3–7) which induces a backward simulation relation from $\mathcal{A}$ to $\mathcal{B}$. Initially, l relates the final states $F_{\mathcal{A}}$ of $\mathcal{A}$ to the final states $F_{\mathcal{B}}$ of $\mathcal{B}$ (lines 6 and 7). Notably, $F_{\mathcal{B}}$ is a pair of q_f and the signature $SigOf(\varepsilon)$. Starting with the final states $F_{\mathcal{A}}$ as the workset, the algorithm takes a state q from the workset and traverses $\mathcal{A}$ backwards following the reverse of each transition (p, γ, q) in $\mathcal{A}$ (lines 10 and 11), and constructs a corresponding transition (p', γ, q') in $\mathcal{B}$ with $p' = (p, I')$ and $I' = Update(I, \gamma)$, for each state $q' = (q, I)$ in $l(q)$ (lines 13–17). The state p will be added to the workset if l relates p to some newly-generated node p' in $\mathcal{B}$, and p' will be set as final states in $\mathcal{B}$ if p is final states in $\mathcal{A}$ (lines 18–22). The while loop proceeds until no more states can be processed in the workset. The algorithm maintains in *done* a set of

Algorithm 1. Building a Sig-$\mathcal{P}$-Automaton $\mathcal{B}$ Accepting a Regular Set of Configurations

```
Input: a $\mathcal{P}$-automaton $\mathcal{A} = (Q_{\mathcal{A}}, \Gamma, \to_{\mathcal{A}}, P, F_{\mathcal{A}})$
       accepting a regular set of configurations
       $S \subseteq P \times \Gamma^*$, where $\mathcal{A}$ has no transitions into $P$
       and no $\varepsilon$-transitions
Output: a Sig-$\mathcal{P}$-automaton $\mathcal{B} = (Q_{\mathcal{B}}, \Gamma, \to_{\mathcal{B}}, I_{\mathcal{B}}, F_{\mathcal{B}})$
        accepting $S$, where $\mathcal{B}$ has no transitions into $P$
        and no $\varepsilon$-transitions

1  $F_{\mathcal{B}} := \{(q_f, (\mathfrak{P}_{A^*}, \varepsilon)) \mid q_f \in F_{\mathcal{A}}\}$
2  $\to_{\mathcal{B}} := \emptyset$
   // Prepare a map $l : Q_{\mathcal{A}} \to 2^{Q_{\mathcal{B}}}$
3  $l := \lambda x.\emptyset$
4  for $q \in Q_{\mathcal{A}} \setminus F_{\mathcal{A}}$ do
5      $l(q) := \emptyset$
6  for $q \in F_{\mathcal{A}}$ do
7      $l(q) := F_{\mathcal{B}}$
   // Build the transitions of $\mathcal{B}$
8  $workset := F_{\mathcal{A}}$; $done := \emptyset$
9  while $workset \neq \emptyset$ do
10     Take + remove a state $q$ from $workset$
11     for $p \xrightarrow{\gamma}_{\mathcal{A}} q$ do
12         for $q' \in l(q)$ with $q' = (q, I)$ do
13             if $(p, \gamma, q') \notin done$ then
14                 $done := done \cup \{(p, \gamma, q')\}$
15                 $I' := Update(I, \gamma)$
16                 $p' := (p, I')$
17                 $\to_{\mathcal{B}} := \to_{\mathcal{B}} \cup (p', \gamma, q')$
18                 if $p' \notin l(p)$ then
19                     $l(p) := l(p) \cup \{p'\}$
20                     $workset := workset \cup \{p\}$
21                     if $p \in F_{\mathcal{A}}$ then
22                         $F_{\mathcal{B}} := F_{\mathcal{B}} \cup \{p'\}$
   // From the transitions, build states of $\mathcal{B}$
23 $Q_{\mathcal{B}} := \bigcup_{(p', \gamma, q') \in \to_{\mathcal{B}}} \{p', q'\}$
24 $I_{\mathcal{B}} := Q_{\mathcal{B}} \cap (P \times \mathsf{Sig})$
25 return $(Q_{\mathcal{B}}, \Gamma, \to_{\mathcal{B}}, I_{\mathcal{B}}, F_{\mathcal{B}})$
```

triplets, which have already been processed for making new transitions in $\mathcal{B}$, so as to ensure the termination of the algorithm (lines 13 and 14).

Next we show in Lemma 2 the correctness of Algorithm 1. There are three things to check. First, $\mathcal{B}$ has no transitions into initial states P and no ε-transitions. This is trivially satisfied by algorithm construction. Second, $\mathcal{B}$ satisfies property (1) and is a Sig-$\mathcal{P}$-automaton. This is a consequence of the algorithm construction and Lemma 1.

Lemma 2. *In Algorithm 1, $\mathcal{B}$ is a* Sig-$\mathcal{P}$*-automaton.*

Proof. We show that condition (1) holds at every step of the construction. We do this through an induction. The initial step, where there are no edges in $\mathcal{B}$, is trivially satisfied. Now in the induction step, we add an edge $(p', Update(\gamma, I)) \xrightarrow{\gamma}{}^{*} (p, I)$. Let $\mathcal{L}_{add} \subseteq \mathcal{L}(\mathcal{B}, p')$ be the set of words that were not accepted before the edge was added but accepted after. We only need to show that for any word $\omega \in \mathcal{L}_{add}$, $([\![\omega]\!], v) = Update(\gamma, I)$ for some $v \in \Gamma^{\leqslant K}$. This is ensured by Lemma 1. $\square$

The third and final thing to check is that $\mathcal{B}$ recognizes the same language as $\mathcal{A}$. To show this, we recall some established facts of simulation relations on finite automata. Let $\mathcal{B}_i = (Q_i, \Gamma_{\varepsilon,i}, \rightarrow_i, Init_i, F_i)$ be NFA for $i \in \{0, 1\}$. A simulation relation between $\mathcal{B}_0$ and $\mathcal{B}_1$ is a relation $\prec \subseteq Q_0 \times Q_1$ such that $q_0 \prec q_1$ if 1) $q_0 \in F_0$ implies that $q_1 \in F_1$, and 2) for any $q_0 \xrightarrow{\gamma}_0 p_0$, there exists $q_1 \xrightarrow{\gamma}_1 p_1$ such that $p_0 \prec p_1$. It is well-established that, given a simulation relation $\prec$ on $\mathcal{B}_0$ and $\mathcal{B}_1$, $q_0 \prec q_1$ implies that $\mathcal{L}(\mathcal{B}_0, q_0) \subseteq \mathcal{L}(\mathcal{B}_1, q_1)$. We say $\mathcal{B}_0$ is simulated by $\mathcal{B}_1$ if there exists a simulation relation $\prec \subseteq Q_0 \times Q_1$, such that, for any $q_0 \in Init_0$, there exists $q_1 \in Init_1$ with $q_0 \prec q_1$. It follows that $\mathcal{L}(\mathcal{B}_0) \subseteq \mathcal{L}(\mathcal{B}_1)$ if $\mathcal{B}_0$ is simulated by $\mathcal{B}_1$. Furthermore, we say $\mathcal{B}_0$ and $\mathcal{B}_1$ are simulation-equivalent, denoted by $\mathcal{B}_0 \sim \mathcal{B}_1$, if $\mathcal{B}_0$ is simulated by $\mathcal{B}_1$, and $\mathcal{B}_1$ is simulated by $\mathcal{B}_0$. We have $\mathcal{L}(\mathcal{B}_0) = \mathcal{L}(\mathcal{B}_1)$ if $\mathcal{B}_0 \sim \mathcal{B}_1$.

Given an NFA $\mathcal{B} = (Q, \Gamma, \rightarrow, Init, F)$, we define a new automaton $\overleftarrow{\mathcal{B}} = (Q, \Gamma, \leftarrow, F, Init)$, by reversing $\rightarrow$, i.e., $\leftarrow = \{(q, \gamma, p) \mid (p, \gamma, q) \in \rightarrow\}$, and by exchanging the initial and finial states of $\mathcal{B}$. Let ω^{-1} be the reverse of word $\omega \in \Gamma^*$. Then $\omega \in \mathcal{L}(\mathcal{B})$ iff $\omega^{-1} \in \mathcal{L}(\overleftarrow{\mathcal{B}})$.

Lemma 3. *Algorithm 1 has the following properties.*

1) *For $q \in Q_{\mathcal{A}}$, and $q' \in Q_{\mathcal{B}}$, we write $q \prec_{\mathcal{AB}} q'$ if $q' \in l(q)$. Then $\prec_{\mathcal{AB}}$ is a simulation relation and $\overleftarrow{\mathcal{A}}$ is simulated by $\overleftarrow{\mathcal{B}}$.*

2) *For $q \in Q_{\mathcal{A}}$, and $q' \in Q_{\mathcal{B}}$, we write $q' \prec_{\mathcal{BA}} q$ if $q' \in l(q)$. Then $\prec_{\mathcal{BA}}$ is a simulation relation and $\overleftarrow{\mathcal{B}}$ is simulated by $\overleftarrow{\mathcal{A}}$.*

Proof. We show that $\overleftarrow{\mathcal{A}}$ is simulated by $\overleftarrow{\mathcal{B}}$. It immediately follows the algorithm construction at lines 12–21 that, for any $p \xrightarrow{\gamma}_{\mathcal{A}} q$, and any $q' \in l(q)$, there exists $p' \xrightarrow{\gamma}_{\mathcal{B}} q'$ such that $p' \in l(p)$ and $p' = (p, I)$ for some I. Line 28 in Algorithm 1 relates initial states of $\overleftarrow{\mathcal{A}}$ and $\overleftarrow{\mathcal{B}}$. We have that $\prec_{\mathcal{AB}}$ is a simulation relation between $\overleftarrow{\mathcal{A}}$ and $\overleftarrow{\mathcal{B}}$. One can similarly conclude with 2) that $\overleftarrow{\mathcal{B}}$ is simulated by $\overleftarrow{\mathcal{A}}$ by the algorithm construction. $\square$

Corollary 1. *Considering Algorithm 1, $\mathcal{B}$ accepts the same language as $\mathcal{A}$.*

Theorem 2. *The time and space required by the $Update((J, v), \gamma)$ function is in $O(K \times |\mathfrak{P}|)$. Algorithm 1 takes $O(|\rightarrow_{\mathcal{A}}| \times |\mathsf{Sig}| \times K \times |\mathfrak{P}|)$ time and space and produces an automaton of size $O(|\rightarrow_{\mathcal{A}}| \times |\mathsf{Sig}|)$.*

Proof. The *Update* function only requires membership checks (membership in J for Inh and in $\mathfrak{P}$ for Inh considering prefixes in v) in linear time, and a small update of the prefix is also linear. Thus we get a linear complexity in K and $\mathfrak{P}$. Consider Algorithm 1, by choosing appropriate data structures, all the needed membership test, removal operations, etc., can take constant time. Further, we know that each triplet $(p, \gamma, q') \in done$ is processed only once in the algorithm. Thus line 14 is executed in $O(|\rightarrow_{\mathcal{A}}| \times |\mathsf{Sig}|)$. In total, this gives us an algorithm in $O(|\rightarrow_{\mathcal{A}}| \times |\mathsf{Sig}| \times K \times |\mathfrak{P}|)$ time and space. Note that the resulting automaton is of size linear in $\rightarrow_{\mathcal{A}}$ and Sig. $\square$

4.4 Saturation Rules for Sig-$\mathcal{P}$-Automata

4.4.1 Evaluating Signatures

Below we give an evaluation function that takes as input a pattern $R \in \mathsf{Pat}$ and a signature $(J, v) \in \mathsf{Sig}$, and determines whether R can be satisfied by J. The correctness is witnessed by Lemma 4.

$$Evaluate :: \mathsf{Pat} \times \mathsf{Sig} \mapsto \{\text{true}, \text{ false}\}$$

$$Evaluate(R, (J, v))$$

$$= \begin{cases} \text{true, if } R \text{ is satisfied by setting the basic patterns} \\ \quad \text{in } J \text{ to true and the ones not in } J \text{ to false,} \\ \text{false, otherwise.} \end{cases}$$

Lemma 4. *For any pattern $R \in$ Pat and any stack word $\omega \in \Gamma^*$, ω satisfies R iff $Evaluate(R, SigOf(\omega)) = true$.*

Proof. Suppose $\omega \in \mathcal{L}(R)$. By induction on the structure of R:

• case $R = \beta$, i.e., R is a basic pattern: then since $\omega \in \mathcal{L}(R)$, $R \in [\![\omega]\!]$ and $Evaluate(R, I) =$ true for any I of shape $([\![\omega]\!], v)$;

• case $R = !R'$: then $\omega \notin \mathcal{L}(!R')$ and by induction hypothesis $Evaluate(R', I) =$ false for any I of shape $([\![\omega]\!], v)$, thus $Evaluate(R, I) =$ true;

• case $R = R_1\&R_2$: by definition of $Evaluate$, $Evaluate(R_1\&R_2, I) = Evaluate(R_1, I) \wedge Evaluate(R_2, I)$, further, $\omega \in \mathcal{L}(R) \iff \omega \in \mathcal{L}(R_1)$ and $\omega \in \mathcal{L}(R_2)$; by induction hypothesis, $Evaluate(R_1, I) =$ true and $Evaluate(R_2, I) =$ true, and then $Evaluate(R, I) =$ true;

• case $R = R_1 + R_2$: similar to the conjunction.

Suppose that $Evaluate(R, I) =$ true where $I = ([\![\omega]\!], v)$ for some v. By induction on the structure of R,

• case $R = \beta$, i.e., R is a basic pattern: by definition of $[\![\omega]\!]$, it must hold that ω satisies R and thus $\omega \in \mathcal{L}(R)$;

• case $R = !R'$: then $Evaluate(R', I) =$ false and $\omega \notin \mathcal{L}(R')$, thus $\omega \in \mathcal{L}(R)$;

• case $R = R_1\&R_2$: we have $\texttt{eval}(R_1, I) =$ true and $Evaluate(R_2, I) =$ true, and by induction hypothesis, $\omega \in \mathcal{L}(R_1)$ and $\omega \in \mathcal{L}(R_2)$, which implies that $\omega \in \mathcal{L}(R)$;

• case $R = R_1 + R_2$: similar to the conjunction. □

Lemma 4 says that checking whether the current stack satisfies a regular condition R can be reduced to evaluating the signature of the stack and the condition. Since each state of Sig-$\mathcal{P}$-automata carries the signature of the stack contents reading forward from the state, we can directly adapt saturation rules of ordinary $\mathcal{P}$-automata to the context of Sig-$\mathcal{P}$-automata.

4.4.2 Forward Saturation Rules

Given a Sig-$\mathcal{P}$-automaton $\mathcal{B}$ recognizing a regular set C of configurations, a Sig-$\mathcal{P}$-automaton $\mathcal{B}_{post^*}$ recognizing $Post^*(C)$ can be computed by iteratively applying the following saturation rules upon termination:

1) if $\langle p,\ \gamma\rangle \overset{R}{\hookrightarrow} \langle p',\ \varepsilon\rangle \in \Delta_c$, $(p, I_1) \xrightarrow{\gamma}{}^* (q, I_2)$ in $\mathcal{B}_{post^*}$, and $Evaluate(R, I_2) =$ true, add transition $((p', I_2), \varepsilon, (q, I_2))$ into $\mathcal{B}_{post^*}$;

2) if $\langle p, \gamma\rangle \overset{R}{\hookrightarrow} \langle p', \gamma'\rangle \in \Delta_c$, $(p, I_1) \xrightarrow{\gamma}{}^* (q, I_2)$ in $\mathcal{B}_{post^*}$, and $Evaluate(R, I_2) =$ true, add transitions $((p', I), \gamma', (q, I_2))$ into $\mathcal{B}_{post^*}$, where $I = Update(I_2, \gamma')$;

3) if $\langle p, \gamma\rangle \overset{R}{\hookrightarrow} \langle p', \gamma'\gamma''\rangle \in \Delta_c$, $(p, I_1) \xrightarrow{\gamma}{}^* (q, I_2)$ in $\mathcal{B}_{post^*}$, and $Evaluate(R, I_2) =$ true, add transitions $((p', I'), \gamma', (q_{p',\gamma'}, I))$ and $((q_{p',\gamma'}, I), \gamma'', (q, I_2))$ into $\mathcal{B}_{post^*}$, where $I = Update(I_2, \gamma'')$ and $I' = Update(I, \gamma')$.

Intuitively, suppose there is a transition rule $\langle p,\ \gamma\rangle \overset{R}{\hookrightarrow} \langle p',\ \omega\rangle \in \Delta_c$ and $(p, I_1) \xrightarrow{\gamma}{}^* (q, I_2)$ is already in $\mathcal{B}_{post^*}$. Since $Evaluate(R, I_2) =$ true, there exists $\omega' \in \mathcal{L}(\mathcal{B}_{post^*}, (q, I_2))$ with $\omega' \in R$ by the property of Sig-$\mathcal{P}$-automata. Then we know $\langle p,\ \gamma\omega'\rangle$ is accepted by $\mathcal{B}_{post^*}$, i.e., $\langle p,\ \gamma\omega'\rangle \in Post^*(C)$. Since the rule can be applied to $\langle p,\ \gamma\omega'\rangle$ here, $\langle p,\ \omega\omega'\rangle$ is the immediate successor of $\langle p,\ \gamma\omega'\rangle$, i.e., $\langle p,\ \omega\omega'\rangle \in Post^*(C)$. By adding the transition $(p', I) \xrightarrow{\omega}{}^* (q, I_2)$ into $\mathcal{B}_{post^*}$, the automaton accepts $\langle p,\ \omega\omega'\rangle$, where I is obtained by repeatedly applying the function $Update$ to ω letter-wise (once or twice as given in the rules above).

4.4.3 Backward Saturation Rules

Similarly, considering a Sig-$\mathcal{P}$-automaton $\mathcal{B}$ that recognizes a regular set C of configurations, the backward saturation rules for computing a Sig-$\mathcal{P}$-automaton $\mathcal{B}_{pre^*}$ that recognizes $Pre^*(C)$ are given as follows: if $\langle p, \gamma\rangle \overset{R}{\hookrightarrow} \langle p', \omega\rangle \in \Delta_c$, $(p', I_1) \xrightarrow{\omega}{}^* (q, I_2)$ in $\mathcal{B}_{pre^*}$, and $Evaluate(R, I_2) =$ true, we add transitions $((p, I), \gamma, (q, I_2))$ into $\mathcal{B}_{pre^*}$, where $I = Update(\gamma, I_2)$. Intuitively, suppose there is a transition rule $\langle p,\ \gamma\rangle \overset{R}{\hookrightarrow} \langle p',\ \omega\rangle \in \Delta_c$ and $(p', I_1) \xrightarrow{\omega}{}^* (q, I_2)$ is already in $\mathcal{B}_{pre^*}$. Since $Evaluate(R, I_2) =$ true, there exists $\omega' \in \mathcal{L}(\mathcal{B}_{pre^*}, (q, I_2))$ with $\omega' \in R$ by the property of Sig-$\mathcal{P}$-automata. Then we know $\langle p',\ \omega\omega'\rangle$ is accepted by $\mathcal{B}_{pre^*}$, i.e., $\langle p',\ \omega\omega'\rangle \in Pre^*(C)$. Since the rule can be applied to $\langle p',\ \omega\omega'\rangle$ here, $\langle p,\ \gamma\omega'\rangle$ is the immediate predecessor of $\langle p',\ \omega\omega'\rangle$, i.e., $\langle p,\ \gamma\omega'\rangle \in Pre^*(C)$. By adding the transition $(p, I) \xrightarrow{\gamma}{}^* (q, I_2)$ into $\mathcal{B}_{pre^*}$, the automaton accepts $\langle p,\ \gamma\omega'\rangle$, where I is obtained by applying $Update(I_2, \gamma)$.

4.4.4 Correctness Analysis

The proposed saturation rules for pCPDS directly extend the classic ones of PDSs. The correctness of the saturation process also naturally follows. We state the correctness theorems below by taking forward saturation as an example.

Theorem 3. *Considering a Sig-$\mathcal{P}$-automaton $\mathcal{B}$ that recognizes a regular set C of configurations, we can construct a Sig-$\mathcal{P}$-automaton $\mathcal{B}_{post^*}$ by applying the forward saturation rules with $\mathcal{L}(\mathcal{B}_{post^*}) = Post^*(C)$ upon termination.*

Proof. First, we show that $\mathcal{B}_{post^*}$ is a Sig-$\mathcal{P}$-automaton during the saturation process. This holds trivially when $\mathcal{B}_{post^*} = \mathcal{B}$ initially, and the property preserves each time when a new transition is added into the automaton. The proof for rules 1 and 2 is trivial for no new internal states are introduced. Considering rule 3, let $(q, I_2) \xrightarrow{\omega}{}^* q_f$ for some final state q_f hold in $\mathcal{B}_{post^*}$. I_2 is the signature of ω. By Lemma 1, I is the signature of $\gamma''\omega$. The property is proved.

Next, for any $\langle p,\ \omega\rangle \in Post^*(C)$, $\langle p',\ \omega'\rangle \Rightarrow_c^* \langle p,\ \omega\rangle$ for some $\langle p',\ \omega'\rangle \in \mathcal{L}(\mathcal{B}) = C$. It is not hard to prove that $p \xrightarrow{\omega}{}^* q_f$ for some final state q_f in $\mathcal{B}_{post^*}$. We omit the proof that is a straightforward extension of the proof of Lemma 3.3 in [1]. Therefore $\langle p,\ \omega\rangle$ is accepted by $\mathcal{B}_{post^*}$. Conversely, for any $\langle p,\ \omega\rangle \in \mathcal{L}(\mathcal{B}_{post^*})$, we can show that $\langle p',\ \omega'\rangle \Rightarrow_c^* \langle p,\ \omega\rangle$ for some $\langle p',\ \omega'\rangle$ such that $p' \xrightarrow{\omega'}{}^* q_f$ for some final state q_f holds in $\mathcal{B}$. We omit the proof that is a straightforward extension of the proof of Lemma 3.4 in [1]. That is, $\langle p,\ \omega\rangle \in Post^*(C)$. □

Finally, we show correctness of our overall analysis in Theorem 4.

Theorem 4. *Let $\mathcal{B}_{post^*}$ be a* Sig-$\mathcal{P}$*-automaton with $\mathcal{L}(\mathcal{B}_{post^*}) = Post^*(S)$. Then T is reachable from S iff $\mathcal{B}_{post^*} \cap \mathcal{A}_T \neq \emptyset$ and unreachable otherwise. If T is a finite union of patterned configurations, then T is reachable from S iff there exists (p, R) in T and an initial state (p, I) in $\mathcal{B}_{post^*}$ such that $Evaluate(R, I) = true$.*

Proof. Suppose there exists $(p, \omega) \in T \cap \mathcal{L}(\mathcal{B}_{post^*})$. Since $\mathcal{L}(\mathcal{B}_{post^*}) = Post^*(S)$, we get $(p, \omega) \in Post^*(S)$ and T is reachable from S. Conversely, suppose that T is reachable from S. Then there is a computation from a configuration in S to a configuration in T. Therefore by definition, $T \cap Post^*(S) \neq \emptyset$ and $\mathcal{A}_T \cap \mathcal{B}_{post^*} \neq \emptyset$.

Consider T is finitely patterned. If there exist (p, R) in T and an initial state (p, I) in $\mathcal{B}_{post^*}$ such that $Evaluate(R, I) =$ true, then let $(p, I) \xrightarrow{\omega}{}^* q_f$ hold for some final state q_f in $\mathcal{B}_{post^*}$ and ω satisfies R followed by $\langle p,\ \omega\rangle \in (p, R) \subseteq T$. Thus T is reachable from S. The reverse direction is similarly proved. □

Example 2. Consider example 1 and the Sig-$\mathcal{P}$-automaton $\mathcal{B}$ shown in Fig.4. The resulted automaton by applying the forward saturation process is given in Fig.5. After the addition of states and transitions, there is a state p_C augmented with the atomic pattern $\gamma_1\Gamma^*$. The target configuration $T = (p_C, \gamma_1\Gamma^*)$ is found reachable from the source configurations.

5 Complexity Analysis and Optimizations

The forward and backward saturation procedures can be efficiently implemented by straightforwardly extending the classic algorithms of ordinary PDSs, with replacing the underlying $\mathcal{P}$-automata with Sig-$\mathcal{P}$-automata and extra checking on regular conditions (see Figs.3.3 and 3.4 in [1]). To be self-contained, we give a forward saturation algorithm in Algorithm 2 and conclude the complexity results in Theorem 5.

Theorem 5. *Following [1], a forward saturation algorithm can be implemented in $O(|P| \times |\Delta_c| \times (n_1 + n_2) \times n_3 \times n_4 + |P| \times |\rightarrow_{\mathcal{B}}|)$ time and space, where $n_1 = |Q_{\mathcal{B}} \setminus P|$, $n_2 = |P| \times |\Gamma| \times |\mathsf{Sig}|$, $n_3 = K \times |\mathfrak{P}|$, n_4 is the time taken by the evaluation function that can be bounded by the size L of patterns, and $|\mathsf{Sig}|$ can be bounded by $O(2^{|\mathfrak{P}|} \times |\Gamma|^K)$.*

In the theorem, the simplified bound of signatures is obtained based on an implicit fact by the algorithm construction that, for any set of signatures I carried by states of a Sig-$\mathcal{P}$-automaton, $v = v'$ for

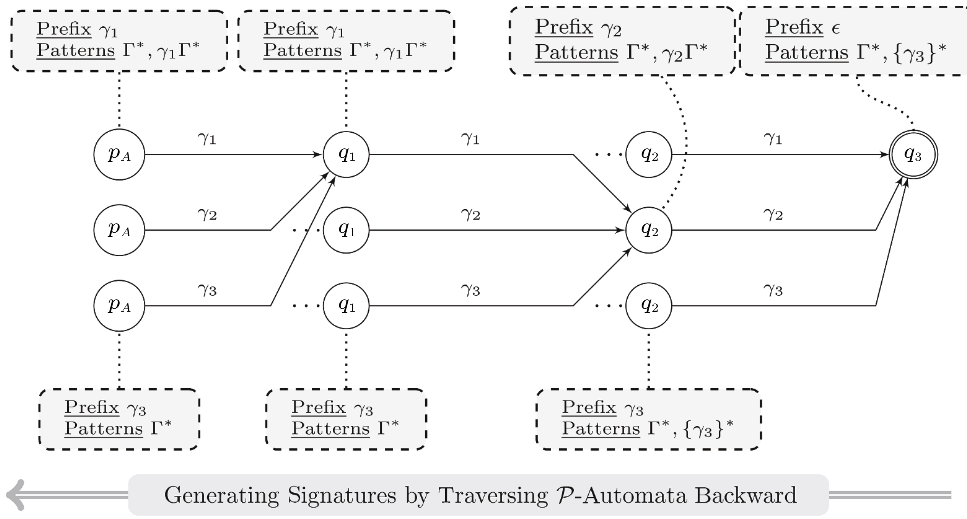

Prefix γ_1 Patterns $\Gamma^*, \gamma_1\Gamma^*$
Prefix γ_1 Patterns $\Gamma^*, \gamma_1\Gamma^*$
Prefix γ_2 Patterns $\Gamma^*, \gamma_2\Gamma^*$
Prefix ϵ Patterns $\Gamma^*, \{\gamma_3\}^*$
p_A
q_1
q_2
q_3
γ_1
γ_2
γ_3
Prefix γ_3 Patterns Γ^*
Prefix γ_3 Patterns Γ^*
Prefix γ_3 Patterns $\Gamma^*, \{\gamma_3\}^*$
Generating Signatures by Traversing $\mathcal{P}$-Automata Backward

Fig.4. Constructing a Sig-$\mathcal{P}$-automaton $\mathcal{B}$ that recognizes source configurations S in example 1, where $\mathfrak{P}$ consists of the following atomic patterns: Γ^*, $\{\gamma_3\}^*$, $\gamma_1\Gamma^*$, $\gamma_2\Gamma^*$.

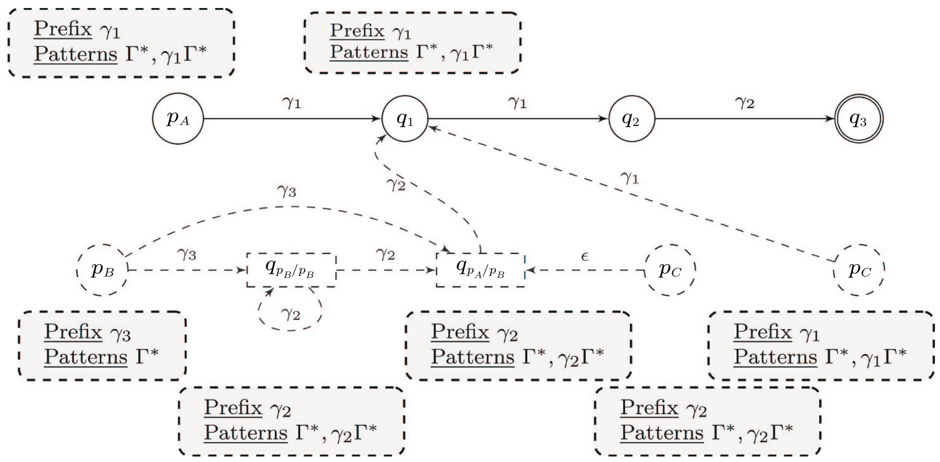


Fig.5. Sig-$\mathcal{P}$-automaton $\mathcal{B}_{post^*}$ with $\mathcal{L}(\mathcal{B}_{post^*}) = Post^*(S)$ for example 1.

any $(J, v), (J, v') \in I$. Thus we conclude $|\mathsf{Sig}| = O(|2^{|\mathfrak{P}|} \times |\Gamma|^K)$.

Algorithm 2. Computing a Sig-$\mathcal{P}$-Automaton $\mathcal{B}_{post^*}$ with $\mathcal{L}(\mathcal{B}_{post^*}) = Post^*(S)$

```
   Input: a $\mathsf{Sig}$-$\mathcal{P}$-automaton $\mathcal{B} = (Q_\mathcal{B}, \Gamma, \rightarrow_\mathcal{B}, Init_\mathcal{B}, F_\mathcal{B})$
          that recognizes $S$, where $\mathcal{B}$ has no transitions into
          the initial states and no $\varepsilon$-transitions
   Output: a $\mathsf{Sig}$-$\mathcal{P}$-automaton $\mathcal{B}_{post^*} =$
          $(Q_{\mathcal{B}_{post^*}}, \Gamma_\varepsilon, \rightarrow_{\mathcal{B}_{post^*}}, Init_{\mathcal{B}_{post^*}}, F_{\mathcal{B}_{post^*}})$ that
          recognizes $Post^*(S)$
1  $ws := \rightarrow_\mathcal{B} \cap (Init_\mathcal{B} \times \Gamma \times Q_\mathcal{B})$; $\rightarrow_{\mathcal{B}_{post^*}} := \rightarrow_\mathcal{B} \setminus ws$
   while $ws \neq \emptyset$ do
2      Take and remove a transition $t$ from $ws$ with
       $t = ((p, I_1), \gamma, (q_0, I_2))$
       if $t \notin \rightarrow_{\mathcal{B}_{post^*}}$ then
3          $\rightarrow_{\mathcal{B}_{post^*}} := \rightarrow_{\mathcal{B}_{post^*}} \cup \{t\}$
           if $\gamma \neq \varepsilon$ then
               for $\langle p, \gamma\rangle \overset{R}{\hookrightarrow} \langle p', \varepsilon\rangle \in \Delta_c$ do
                   if $Evaluate(R, I_2) =$ true then
4                      $ws := ws \cup \{((p', I_2), \varepsilon, (q_0, I_2))\}$
               for $\langle p, \gamma\rangle \overset{R}{\hookrightarrow} \langle p', \gamma'\rangle \in \Delta_c$ do
                   if $Evaluate(R, I_2) =$ true then
5                      $I = Update(\gamma', I_2)$
6                      $ws := ws \cup \{((p', I), \gamma', (q_0, I_2))\}$
               for $\langle p, \gamma\rangle \overset{R}{\hookrightarrow} \langle p', \gamma'\gamma''\rangle \in \Delta_c$ do
                   if $Evaluate(R, I_2) =$ true then
7                      $I = Update(\gamma'', I_2)$
8                      $I' = Update(\gamma', I)$
9                      $ws := ws \cup \{((p', I'), \gamma', (q_{p',\gamma'}, I))\}$
10                     $\rightarrow_{\mathcal{B}_{post^*}} :=$
                       $\rightarrow_{\mathcal{B}_{post^*}} \cup \{((q_{p',\gamma'}, I), \gamma'', (q_0, I_2))\}$
                       for $((p'', I), \varepsilon, (q_{p',\gamma'}, I))$ do
11                         $ws :=$
                           $ws \cup \{((p'', I), \gamma'', (q_0, I_2))\}$
           else
               for $((q_0, I_2), \gamma', (p'_0, I_3)) \in \rightarrow_{\mathcal{B}_{post^*}}$ do
12                 $ws := ws \cup \{((p, I_1), \gamma', (p'_0, I_3))\}$
13 $Q_{\mathcal{B}_{post^*}} := \bigcup_{(q,\gamma,q') \in \rightarrow_{\mathcal{B}_{post^*}}} \{q, q'\}$;
   $Init_{\mathcal{B}_{post^*}} := Q_{\mathcal{B}_{post^*}} \cap (P \times 2^{\mathsf{Sig}})$
14 return $(Q_{\mathcal{B}_{post^*}}, \Gamma_\varepsilon, \rightarrow_{\mathcal{B}_{post^*}}, Init_{\mathcal{B}_{post^*}}, F_{\mathcal{B}_{post^*}})$
```

Recall that the Sig-$\mathcal{P}$-automaton $\mathcal{B}$ created in Algorithm 1 is of size $O(|\rightarrow_\mathcal{A}| \times |\mathsf{Sig}|)$ and is created in time $O(|\rightarrow_\mathcal{A}| \times |\mathsf{Sig}| \times |\mathfrak{P}| \times K)$. This gives us an overall algorithm in $O(|P| \times |\Delta_c| \times (n_1 + n_2) \times K \times |\mathfrak{P}| \times L + |P| \times |\rightarrow_\mathcal{A}| \times |\mathsf{Sig}| + |\rightarrow_\mathcal{A}| \times |\mathsf{Sig}| \times |\mathfrak{P}| \times K)$ time and space, simplified as $O(|P| \times |\Delta_c| \times (|\rightarrow_\mathcal{A}| + |P| \times |\Gamma|) \times 2^{|\mathfrak{P}|} \times |\mathfrak{P}| \times L \times |\Gamma|^K \times K)$, as given in Table 1.

We move to the complexity analysis of simple pCPDSs. If a pattern R is simple then it is simply a disjunction of atomic patterns. Considering a simple pattern R, the satisfiability checking by $Evaluation(R, (J, v))$ is reduced to checking whether any basic pattern in J also appears in R. This observation enables us to further optimize our analysis algorithm: it will suffice for each state in Sig-$\mathcal{P}$-automaton to only carry a single atomic pattern that it admits. The modification on our algorithms is minimal: for any transition $((p, (J, v)), \gamma, (q, (J', v')))$ to be added into the Sig-$\mathcal{P}$-automaton in both Algorithm 1 and Algorithm 2, it will be divided as a set of transitions $\bigcup_{\beta \in J, \beta' \in J'} (p, (\beta, v)), \gamma, (q, (\beta', v'))$. The correctness of the algorithms is preserved, trivially. The time and space complexity becomes $O(|\mathfrak{P}|^2 \times (|\rightarrow_\mathcal{A}| \times |\mathsf{Sig}| \times |\mathfrak{P}| \times K))$ for Algorithm 1 and $O(|\mathfrak{P}|^2 \times (|P| \times |\Delta_c| \times (n_1 + n_2) \times n_3 \times L + |P| \times |\rightarrow_\mathcal{B}|))$ for Algorithm 2, where $|\mathsf{Sig}| = O(|\mathfrak{P}| \times |\Gamma|^K)$. The overall analysis takes $O(|P| \times |\Delta_c| \times (|\rightarrow_\mathcal{A}| + |P| \times |\Gamma|) \times |\mathfrak{P}|^4 \times L \times |\Gamma|^K \times K)$ time and space, as given in Table 1. The algorithm is now fixed-parameter polynomial in parameter K.

Finally, we summarize the complexity results of our algorithms in Table 1, taking into account time and space consumed in steps 1 and 2 of the approach that are dominating. As given in Table 1, our algorithm exhibits an exponential complexity in the size of atomic patterns for complete pCPDSs. Here L is the upper

Table 1. Complexity Results of Forward Saturation Algorithms for pCPDSs, Considering Time and Space Consumed in Steps 1 and 2

Algorithm	Complexity Result
Detour to reachability checking of PDSs[11]	$O\big(\lvert P\rvert \times \lvert\Delta_c\rvert \times \lvert States\rvert \times (\lvert Q_{\mathcal{A}} \setminus P\rvert + \lvert P\rvert \times \lvert\Gamma\rvert \times \lvert States\rvert) + \lvert P\rvert \times \lvert \rightarrow_{\mathcal{A}} \rvert\big)$
On-the-fly reachability checking of weighted CPDSs[19]	$O\big(\lvert P\rvert \times \lvert\Delta_c\rvert \times \lvert States\rvert \times (\lvert Q_{\mathcal{A}} \setminus P\rvert + \lvert P\rvert \times \lvert\Gamma\rvert \times \lvert States\rvert) + \lvert P\rvert \times \lvert \rightarrow_{\mathcal{A}} \rvert\big)$
Saturation algorithm for PDS with transductions[21]	$O\big(\lvert S\rvert \times (f\lvert\mathcal{T}\rvert) \times \lvert\Delta\rvert^3 \times \lvert\Gamma\rvert\big)$
Pattern-driven saturation algorithm (this work)	$O(\lvert P\rvert \times \lvert\Delta_c\rvert \times (\lvert \rightarrow_{\mathcal{A}} \rvert + \lvert P\rvert \times \lvert\Gamma\rvert) \times 2^{\lvert\mathfrak{P}\rvert} \times \lvert\mathfrak{P}\rvert \times L \times \lvert\Gamma\rvert^K \times K)$ (complete pCPDS) $O(\lvert P\rvert \times \lvert\Delta_c\rvert \times (\lvert \rightarrow_{\mathcal{A}} \rvert + \lvert P\rvert \times \lvert\Gamma\rvert) \times \lvert\mathfrak{P}\rvert^4 \times L \times \lvert\Gamma\rvert^K \times K)$ (simple pCPDS)

bound size of patterns (viewed as boolean formula of atomic patterns). For simple pCPDSs, our algorithms can be solved in fixed-parameter polynomial time and space. It coincides with those tractable instances of pCPDSs found in practice that have tailored efficient algorithms. For instance, one of the polynomial time algorithms for JDK stack inspection in [15] transforms a given CPDS A to an equivalent PDS A' in such a way that q is reachable in A if and only if $\langle q, prm\rangle$ is reachable in A' where prm is the set of permissions that q possesses. This corresponds to a special case of the subclass that the method in this paper works in polynomial time when $K = 1$ and $\mathfrak{P}$ is a constant. We also compare our analysis with other general algorithms for the reachability checking of CPDSs. The first row in Table 1 shows the complexity results for the original algorithm of CPDSs that takes a detour to reachability checking of PDSs[22] and its improved version, an on-the-fly reachability checking algorithm of weighted PDSs[19]. The number of states in the product automaton $|States|$ can grow exponentially large. The general algorithms of CPDSs in [11] and [19] would still exhibit an exponential time for simple pCPDSs. The second row shows the results for forward saturation algorithm designed for reachability analysis of PDSs with transductions, where $|S|$ is the number of states in the finite automaton extended with transductions that recognizes the source configurations, $\mathcal{T}$ is a finite set of transductions over Γ^*, and f is some computable function. The algorithm is also fixed-parameter tractable depending on the parameterized transductions

A possible optimization occurs when T is a finite union of patterned configurations. This is often the case we found in practical cases, when the intersection and emptiness checking in step 3 of our approach can be avoided, and is able to be replaced by a simple inspection against T and initial states of the saturated automaton (as stated in Theorem 4) in linear time. Last but not least, this optimized checking also enables us to terminate the saturation process early as soon as the target configurations are found reachable by a lightweight inspection of the initial states.

6 Experiments

In this section, we report preliminary experimental results to evaluate the proposed approach. We have implemented a reachability checking tool for pCPDS in Java, based on the algorithms in Section 4 and Section 5. Our implementation extends the model checker jMoped[3] developed for reachability analysis of weighted PDS. The major goal of our experiments is to conduct empirical study on the efficiency and scalability of the proposed pattern-driven reachability analysis algorithm of pCPDS with simple patterns, which as we have shown is solvable in fixed-parameter polynomial time and space and is capable of modelling practical instances of CPDSs. All experiments were performed on a Mac OS X v.10.9.2 with 1.7 GHz Intel Core i7 processor, and 8 GB RAM.

Instead of restricting to particular applications, we conduct the evaluation from a general perspective with an example pCPDS that consists of non-trivial mutual-recursive pushdown transition rules (corresponding to mutual-recursive function calls in the programs) and carry various kinds of simple patterns enumerated from the given alphabet $\Gamma = \{\gamma_1, \dots, \gamma_n\}$. That is, the set of atomic patterns $\mathfrak{P}$ contains two sets of atomic patterns $\bigcup_{A\subseteq\Gamma} A^*$ and $\bigcup_{A\subseteq\Gamma,\gamma_1\dots\gamma_k\in\Gamma^*} A^*\gamma_1\dots\gamma_k\Gamma^*$ for some given $k > 0$. The simple patterns attached to transition rules are randomly-enumerated combinations of atomic patterns from $\mathfrak{P}$ formed by the union operator. The underlying pushdown system of our example is given in Fig.6(a). We also give the $\mathcal{P}$-automaton accepting $post^*(S)$ in Fig.6(b) computed by the transition rules in Fig.6(a). Each internal state in the automa-

[3]https://www7.in.tum.de/tools/jmoped/, Sept. 2020.

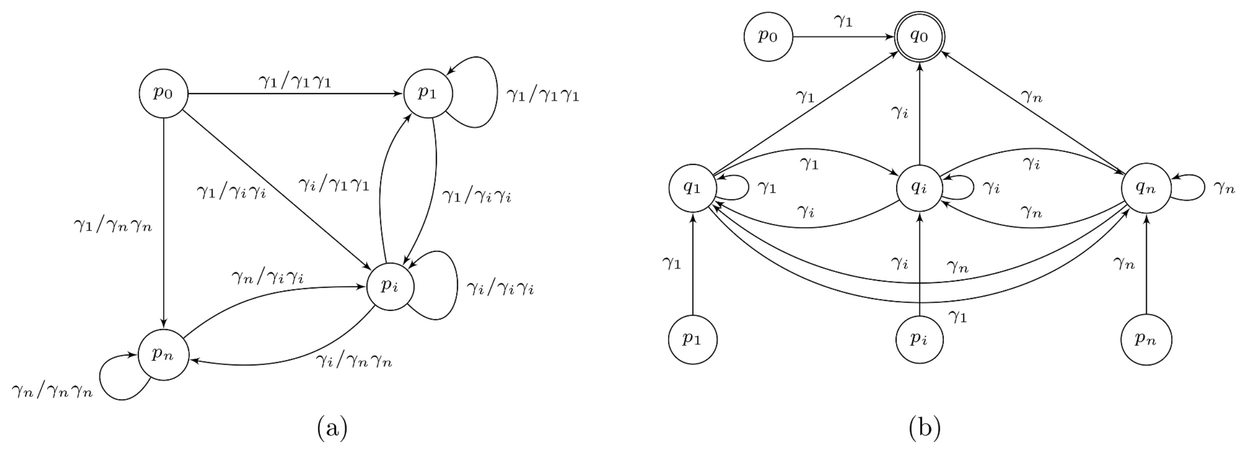

p_0
p_1
p_i
p_n
$\gamma_1/\gamma_1\gamma_1$
$\gamma_1/\gamma_i\gamma_i$
$\gamma_1/\gamma_n\gamma_n$
$\gamma_i/\gamma_1\gamma_1$
$\gamma_i/\gamma_i\gamma_i$
$\gamma_i/\gamma_n\gamma_n$
$\gamma_n/\gamma_i\gamma_i$
$\gamma_n/\gamma_n\gamma_n$
(a)
q_0
q_1
q_i
q_n
γ_1
γ_i
γ_n
(b)

Fig.6. Underlying PDS of (a) the example pCPDS, with (b) the $\mathcal{P}$-automaton accepting $post^*(S)$, where $S = \{\langle p_0, \gamma_1\rangle\}$ is the set of source configurations, p_i ($i \in [0..n]$) is control locations, q_0 is the final state, and q_i is indexed by (p_i, γ_i) for each $i \in [1..n]$ by the construction of algorithm. In (a), each transition $(p, \gamma/\gamma'\gamma'', p')$ says that, replace γ by $\gamma'\gamma''$ on top of the stack when changing the state from p to p'. Here the state reachability for control locations is of interest.

ton has incoming edges with reading any stack symbol γ_i ($i \in [1..n]$) in the alphabet. Since the computation of pCPDS synchronizes the two computing machineries of pushdown systems and condition automata, the underlying PDS designed in this way is capable of traversing the state space of the product automaton over condition automata when necessary.

In Table 2, we report results on the performance of our algorithm for computing the Sig-$\mathcal{P}$-automaton accepting $Post^*(S)$ that is the dominating step of the entire approach. Here $|\mathcal{C}|$ denotes the set of distinguished regular conditions. We have compared our analysis with the original reachability analysis algorithm[11] and the on-the-fly algorithm[19]. For the former, the original algorithm does not scale since the resulted PDS transformed from the target pCPDS explodes easily. For the latter, our approach scales much better as shown in Table 2. Given the fixed-parameter $K = 2$, #States and #Trans list the number of states and transitions in the saturated $\mathcal{P}$-automaton accepting $Post^*(S)$, respectively. As we increase the size of the alphabet, the on-the-fly algorithm rapidly deteriorates in performance since $|\Gamma| \geqslant 5$. In contrast, the pattern-driven algorithm can steadily scale to large example settings.

7 Conclusions

We studied the reachability problem of pCPDSs, a subclass of CPDSs carrying regular conditions obeying certain patterns. Despite of being syntactically constrained, pCPDSs are expressive enough to model the existing application instances of CPDSs in practice. By introducing a way of computing the so-called signatures of the stack contents, we directly extended the classic $\mathcal{P}$-automata techniques and saturation algorithms of PDSs to solve the reachability problem of pCPDSs. Our new algorithms avoid the immediate explosion caused by the approach that translates the problem to that of PDSs. Further, we identified a subclass called simple pCPDS for which there exist fixed-parameter polynomial solutions for reachability of pCPDSs in some para-

Table 2. Experimental Results on the Scalability of Our Approach for Computing Sig-$\mathcal{P}$-Automaton Accepting $cpost^*(S)$, Compared with an On-The-Fly Algorithm in [19]

pCPDS				Pattern-Driven Algorithm			On-the-Fly Algorithm[19]		
$\lvert\Gamma\rvert$	$\lvert\Delta_c\rvert$	$\lvert\mathfrak{P}\rvert$	$\lvert\mathcal{C}\rvert$	#States	#Trans	Time (s)	#States	#Trans	Time (s)
3	303	10	46	51	200	0.22	20	62	0.24
4	2 524	19	220	106	504	0.42	41	173	4.93
5	12 245	31	735	177	989	1.38	65	345	159.50
6	42 906	48	1 955	269	1 697	3.91	–	–	–
7	121 219	71	4 445	383	2 666	12.84	–	–	–
8	294 008	101	9 016	520	3 934	24.15	–	–	–

Note: $K = 2$, timeout is set to be 3 minutes. – indicates that the analysis is timed out.

meter K. Our preliminary empirical study showed that the proposed approach is promising to scale in practice for a given fixed parameter K. Our approach is parameterized by atomic patterns and could be applicable to other applications[22].

References

[1] Schwoon S. Model-checking pushdown systems [Ph.D. Thesis]. Department of Computer Science, Technische Universität München, 2002.

[2] Suwimonteerabuth D, Berger F, Schwoon S, Esparza J. jMoped: A test environment for Java programs. In *Proc. the 33rd International Conference on Computer-Aided Verification*, July 2017, pp.164-167.

[3] Ball T, Rajamani S K. The SLAM project: Debugging system software via static analysis. In *Proc. the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 2002, pp.1-3.

[4] Reps T W, Schwoon S, Jha S, Melski D. Weighted pushdown systems and their application to interprocedural dataflow analysis. *Sci. Comput. Program.*, 2005, 58(1/2): 206-263.

[5] Song F, Touili T. PuMoC: A CTL model-checker for sequential programs. In *Proc. the 27th IEEE/ACM International Conference on Automated Software Engineering*, September 2012, pp.346-349.

[6] Hague M, Ong C H L. Analysing mu-calculus properties of pushdown systems. In *Proc. the 17th International SPIN Workshop on Model Checking Software*, September 2010, pp.187-192.

[7] Bouajjani A, Müller-Olm M, Touili T. Regular symbolic analysis of dynamic networks of pushdown systems. In *Proc. the 16th International Conference on Concurrency Theory*, August 2005, pp.473-487.

[8] Cai X J, Ogawa M. Well-structured pushdown systems. In *Proc. the 24th International Conference on Concurrency Theory*, August 2013, pp.121-136.

[9] Abdulla P A, Atig M F, Stenman J. Dense-timed pushdown automata. In *Proc. the 27th Annual IEEE Symposium on Logic in Computer Science*, June 2012, pp.35-44.

[10] Abdulla P A, Atig M F, Stenman J. Computing optimal reachability costs in priced dense-timed pushdown automata. In *Proc. the 8th International Conference Language and Automata Theory and Applications*, March 2014, pp.62-75.

[11] Esparza J, Kucera A, Schwoon S. Model-checking LTL with regular valuations for pushdown systems. In *Proc. the 4th International Symposium on Theoretical Aspects of Computer Software*, October 2001, pp.316-339.

[12] Li X, Ogawa M. Conditional weighted pushdown systems and applications. In *Proc. the 2010 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, January 2010, pp.141-150.

[13] Minamide Y, Mori S. Reachability analysis of the HTML5 parser specification and its application to compatibility testing. In *Proc. the 18th International Symposium on Formal Methods*, August 2012, pp.293-307.

[14] Johnson J I, Sergey I, Earl C, Might M, van Horn D. Pushdown flow analysis with abstract garbage collection. *Journal of Functional Programming*, 2014, 24(2/3): 218-283.

[15] Nitta N, Takata Y, Seki H. An efficient security verification method for programs with stack inspection. In *Proc. the 8th ACM Conference on Computer and Communications Security*, November 2001, pp.68-77.

[16] Shivers O G. Control-flow analysis of higher order languages or taming lambda [Ph.D. Thesis]. Carnegie Mellon University, 1991.

[17] Bravenboer M, Smaragdakis Y. Strictly declarative specification of sophisticated points-to analyses. In *Proc. the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, October 2009, pp.243-262.

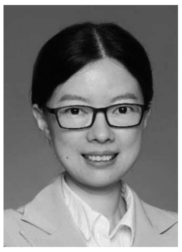
[18] Lhoták O, Hendren L. Context-sensitive points-to analysis: Is it worth it? In *Proc. the 15th International Conference on Compiler Construction*, March 2006, pp.47-64.

[19] Thanh H V L, Li X. An on-the-fly algorithm for conditional weighted pushdown systems. *Journal of Information Processing*, 2014, 22(4): 1-7.

[20] Uezato Y, Minamide Y. Pushdown systems with stack manipulation. In *Proc. the 11th International Symposium on Automated Technology for Verification and Analysis*, October 2013, pp.412-426.

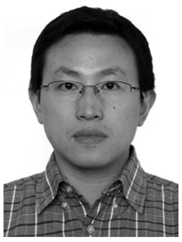
[21] Song F, Miao W K, Pu G G, Zhang M. On reachability analysis of pushdown systems with transductions: Application to Boolean programs with call-by-reference. In *Proc. the 26th International Conference on Concurrency Theory*, September 2015, pp.383-397.

[22] Esparza J, Ganty P. Complexity of pattern-based verification for multithreaded programs. In *Proc. the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 2011, pp.499-510.



Xin Li is a research associate professor in East China Normal University, Shanghai. She received her Ph.D. degree in information processing from Japan Advanced Institute of Science and Technology, Nomi, in 2007. Her research interests include formal methods and verification, especially about the theory and practice of model checking, and interdisciplinary machine learning research.

Patrick Gardy received his Ph.D. degree in computer science from University Paris-Saclay (France) in 2017, and afterwards was a post-doctorate at East China Normal University, Shanghai, when this paper was written. He now works in Railenium (France). His research interests focus on formal verification.



Yu-Xin Deng received his B.Eng. and M.Sc. degrees from Shanghai Jiao Tong University, Shanghai, in 1999 and 2002, respectively, and his Ph.D. degree in computer science from Ecole des Mines de Paris, Paris, in 2005. He is a professor in East China Normal University, Shanghai. His research interests include concurrency theory, especially about process calculi, and formal semantics of programming languages, as well as formal verification. He authored the book titled Semantics of Probabilistic Processes: An Operational Approach (Springer, 2015).

Hiroyuki Seki received his Ph.D. degree in computer science from Osaka University, Osaka, in 1987. He was an assistant professor, and later, an associate professor in Osaka University from 1987 to 1994. In 1994, he joined Nara Institute of Science and Technology, Nara, where he was a professor during 1996–2013. Currently, he is a professor in Nagoya University, Nagoya. His current research interests include formal language theory and formal approach to software development.