**Journal of Information Processing Vol.22 No.4 1–7 (Oct. 2014)**

[DOI: 10.2197/ipsjjip.22.1]

**Regular Paper**

# An On-The-Fly Algorithm for Conditional Weighted Pushdown Systems

Hua Vy Le Thanh[1,a)] Xin Li[2,b)]

Received: February 17, 2014, Accepted: June 5, 2014

**Abstract:** Pushdown systems (PDSs) are well-understood as abstract models of recursive sequential programs, and weighted pushdown systems (WPDSs) are a general framework for solving certain meet-over-all-path problems in program analysis. Conditional WPDSs (CWPDSs) further extend WPDSs to enhance the expressiveness of WPDSs, in which each transition is guarded by a regular language over the stack that specifies conditions under which a transition rule can be applied. CWPDSs or its instance are shown to have wide applications in analysis of objected-oriented programs, access rights analysis, etc. Model checking CWPDSs was shown to be reduced to model checking WPDSs, and an offline algorithm was given that translates CWPSs to WPDSs by synchronizing the underlying PDS and finite state automata accepting regular conditions. The translation, however, can cause an exponential blow-up of the system. This paper presents an on-the-fly model checking algorithm for CWPDSs that synchronizes the computing machineries on-demand while computing post-images of regular configurations. We developed an on-the-fly model checker for CWPDSs and apply it to models generated from the reachability analysis of the HTML5 parser specification. Our preliminary experiments show that, the on-the-fly algorithm drastically outperforms the offline algorithm regarding both practical space and time efficiency.

**Keywords:** Conditional Weighted Pushdown System, Model Checking

## 1. Introduction

Pushdown systems (PDSs) are well understood as abstract models of sequential programs with recursive procedures. By encoding programs as pushdown systems, procedure calls and returns are guaranteed to be correctly paired. Weighted pushdown systems (WPDSs) [4] extend PDSs, by associating a value with each transition, and are a general framework for solving certain meet-over-all-path problems in program analysis. Program analysis yielded by WPDSs are context-sensitive in terms of valid paths. Li et al [1] further extended WPDSs to Conditional Weighted Pushdown Systems (CWPDSs), in which each transition is guarded by a regular language that specifies conditions under which the transition can be applied. The extension is motivated by the observation that, WPDSs are not precise enough to model objected-oriented programs like Java in program analysis.

CWPDSs are more expressive than WPDSs, and have wider applications. A points-to analysis algorithm was designed in the framework of CWPDSs, which enjoys context-sensitivity beyond valid paths regarding heap abstraction, call graph construction, and heap access [1]. Besides modelling objected-oriented program features, it can yield algorithms for generating stack-based access control policies for programs that employ stack-based access control mechanism, such as Java or and Microsoft CLR (common language runtime) [7]. Minamide et al [2] adopted conditional pushdown systems (CPDSs), an instance of CWPDSs, to model the HTML5 parser specification, and performed reachability analysis over it. They proposed a new algorithm for reachability analysis of CPDSs and successfully found several compatibility problems of web browsers with HTML5 specifications.

It was shown that the model checking problem on CWPDSs can be reduced to that on WPDSs, and an offline algorithm was given, which translates CWPSs to WPDSs by synchronizing the underlying pushdown system and finite state automata built for regular conditions [1]. The translation, however, can cause an exponential blow-up of the system, and prohibits the offline algorithm from practical applications. The original offline algorithm for CWPDSs generated the entire state space, whereas only part of the space needs exploring to answer a model checking enquiry.

This paper makes the following contributions:

- We present an on-the-fly model checking algorithm for CWPDSs, by interleaving saturation procedures for computing post-images with regular condition checking, and synchronizing the underlying pushdown systems and finite state automata accepting regular conditions on-demand. We also provide concrete algorithms for those steps of model checking CWPDSs that are only defined in the original paper.

[1] University of Science - Ho Chi Minh City.
[2] Corresponding author. The University of Tokyo.
[a)] thanhvy@gmail.com
[b)] li-xin@kb.is.s.u-tokyo.ac.jp

- We implemented the on-the-fly model checking algorithm in terms of computing post-images, and evaluated the algorithm with one of the large examples analysed in reachability analysis of HTML5 parser specification. Our preliminary experiments showed that the on-the-fly algorithm drastically outperforms the offline algorithm regarding both space and time efficiency in practice.

**Related Work**

Esparza et al introduced solutions for model checking LTL on pushdown systems with regular valuation. As an application, they introduced a formal model called pushdown systems with checkpoints, and showed that the reachability problem of the model is EXPTIME-complete. The underlying CPDSs of CWPDSs can be regarded as an instance of the model, and also shares the EXPTIME-hardness for reachability checking. So our on-the-fly algorithm for CWPDs is designed to hopefully be practical by exploring the state space relevant to model checking enquiries.

Minamide et al. adopted CPDSs as a formal model for reachability analysis of HTML5 parser specification and then checked compatibility of web browsers with specifications. The authors proposed a new algorithm for computing regular pre-images of CPDSs, by extending $\mathcal{P}$-automata with regular lookahead. Though the algorithm also has exponential complexity, it aimed to avoid the exponential blow-up of the system state space due to the translation. Preliminary experimental results in Section 4 indicate that our on-the-algorithm is also competitive in avoiding the exponential blow-up before reachability analysis.

**Organization** The remainder of the paper is organized as follows. Section 2 recalls CWPDSs and the off-line model checking algorithm. Section 3 presents an on-the-fly algorithm for CWPDSs, and an example that illustrates the procedure. Our empirical study is given in Section 4, which compares performance of the on-the-fly algorithm and the off-line algorithm. Section 5 concludes the paper.

## 2. Preliminary

### 2.1 Weighted Pushdown Systems

**Definition 2.1** A **pushdown system** $\mathcal{P}$ is $(Q, \Gamma, \Delta, q_0, \omega_0)$ where $Q$ is a finite set of control locations, $\Gamma$ is a finite stack alphabet, $\Delta \subseteq Q \times \Gamma \times Q \times \Gamma^*$ is a finite set of transitions, $q_0 \in Q$ and $\omega_0 \in \Gamma^*$ is the initial control location and stack contents, respectively. A transition $(p, \gamma, q, \omega) \in \Delta$ is written as $\langle p, \gamma\rangle \hookrightarrow \langle q, \omega\rangle$. A **configuration** is a pair $\langle q, \omega\rangle$ with $q \in Q$ and $\omega \in \Gamma^*$. A computation relation $\Rightarrow$ is defined on configurations, such that $\langle p, \gamma\omega'\rangle \Rightarrow \langle q, \omega\omega'\rangle$ for all $\omega' \in \Gamma^*$ if $\langle p, \gamma\rangle \hookrightarrow \langle q, \omega\rangle$, and the reflexive and transitive closure of $\Rightarrow$ is denoted by $\Rightarrow^*$. Given a set of configurations $C$, we define $pre^*(C) = \{c' \mid \exists c \in C : c' \Rightarrow^* c\}$ and $post^*(C) = \{c' \mid \exists c \in C : c \Rightarrow^* c'\}$ to be the (possibly infinite) set of pre- and post- images of $C$, respectively.

A pushdown system is a variant of pushdown automata without input alphabet. It is not used as language acceptors but a model of system behaviors.

**Definition 2.2** A **bounded idempotent semiring** $\mathcal{S}$ is $(D, \oplus, \otimes, \bar{0}, \bar{1})$, where $\bar{0}, \bar{1} \in D$, and

- $(D, \oplus)$ is a commutative monoid with $\bar{0}$ as its unit element, and $\oplus$ is idempotent, i.e., $a \oplus a = a$ for all $a \in D$;
- $(D, \otimes)$ is a monoid with $\bar{1}$ as the unit element;
- $\otimes$ distributes over $\oplus$, i.e., for all $a, b, c \in D$, $a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c)$ and $(b \oplus c) \otimes a = (b \otimes a) \oplus (c \otimes a)$ ;
- for all $a \in D, a \otimes \bar{0} = \bar{0} \otimes a = \bar{0}$;
- A partial ordering $\sqsubseteq$ is defined on $D$ such that $a \sqsubseteq b$ iff $a \oplus b = a$ for all $a, b \in D$, and there are no infinite descending chains in $D$.

**Definition 2.3** A **weighted pushdown system** $\mathcal{W}$ is $(\mathcal{P}, \mathcal{S}, f)$, where $\mathcal{P} = (Q, \Gamma, \Delta, p_0, \omega_0)$ is a pushdown system, $\mathcal{S} = (D, \oplus, \otimes, \bar{0}, \bar{1})$ is a bounded idempotent semiring, and $f : \Delta \to D$ is a weight assignment function.

By Def. 2.2, we have that $\bar{0}$ is the greatest element.

**Definition 2.4** Given a weighted pushdown system $\mathcal{W} = (\mathcal{P}, \mathcal{S}, f)$ where $\mathcal{P} = (Q, \Gamma, \Delta, p_0, \omega_0)$. Let $\sigma = [r_1, \ldots, r_k] \in \Delta^*$ denote a sequence of pushdown transition rules, and define $val(\sigma) = f(r_1) \otimes \cdots \otimes f(r_k)$. For any $c, c' \in Q \times \Gamma^*$, we denote by $path(c, c')$ the set of sequences of transition rules that transform configurations from $c$ into $c'$. Given regular sets of configurations $S, T \subseteq Q \times \Gamma^*$, the **meet-over-all-valid-path** (MOVP) problem computes

$$\mathsf{MOVP}(S, T) = \oplus\{val(\sigma) \mid \sigma \in path(s, t), s \in S, t \in T\}$$

WPDSs [4] solve the MOVP problem in program analysis, of which data domains comply with the bounded idempotent semiring. When applying WPDSs to program analysis, PDSs typically model (recursive) control flows of the program, weights encode transfer functions, $\otimes$ corresponds to the reverse of function composition, and $\oplus$ joins data flows.

### 2.2 Computing Post Images

**Definition 2.5** Given a PDS $\mathcal{P} = (Q, \Gamma, \Delta, q_0, \omega_0)$. A $\mathcal{P}$-automaton $\mathcal{A} = (Q', \Gamma, \rightarrow, Q, F)$ is a non-deterministic finite automaton, where $Q'$ is the finite set of states, $\Gamma$ is the input alphabet, $\rightarrow \subseteq Q' \times \Gamma \times Q'$ is the set of transitions, and $Q \subseteq Q'$ and $F \subseteq Q'$ are the set of initial and final states, respectively. We define $\rightarrow^* \subseteq Q' \times \Gamma^* \times Q'$ as the smallest relation such that

- $p \xrightarrow{\epsilon}{}^* p$ for any $p \in Q'$;
- $p \xrightarrow{\gamma}{}^* p'$ if $(p, \gamma, p') \in \rightarrow$;
- $p \xrightarrow{\omega\gamma}{}^* p'$ if $p \xrightarrow{\omega}{}^* p''$ and $p'' \xrightarrow{\gamma}{}^* p'$ for some $p'' \in Q'$.

A configuration $\langle p, \omega\rangle$ is accepted by $\mathcal{A}$ if $p \xrightarrow{\omega}{}^* q$ for some $q \in F$. A set $C$ of configurations is accepted by $\mathcal{A}$ if each $c \in C$ is accepted by $\mathcal{A}$. A set $C$ of configurations is regular if it is accepted by some $\mathcal{P}$-automaton, denoted by $\mathcal{A}_C$.

A key property of pushdown systems is that, a regular set of configurations is closed under forward and backward reachability. Therefore infinite configurations can be represented by finite means of $\mathcal{P}$-automata, and pre- and post-images can be computed by saturating the automaton.

We illustrate in Fig. 1 the saturation rules for computing $post^*(C)$ of a regular set $C$ of configurations. The saturation procedure takes as input the $\mathcal{P}$-automaton $\mathcal{A}_C$ that

recognizes $C$, and augments $\mathcal{A}_C$ with new edges and states by the saturation rules until convergence, and the stabilised automaton accepts $post^*(C)$. In the figure, solid edges and states reside in the current automaton, and dashed edges and states are newly added by saturation rules.

The MOVP problem tackled by WPDSs can be solved by saturating a *weighted* extension of $\mathcal{P}$-automata when computing pre- or post-images. Consider computing post-images as given in Algorithm 2 and ignore those underlined parts. Initially, all transitions in $\mathcal{A}_C$ are labelled with $\bar{1}$. The saturation procedure computes or updates the value for each transition when applying the saturation rule at line 18, 21, 27 and 30, and outputs a weighted automaton $\mathcal{A}_{post*(C)}$ of which each transition is associated with a value from $D$. The final result computed by the MOVP problem can be read-off from $\mathcal{A}_{post*(C)}$. Note that the time complexity is increased from the unweighted case by a factor no more than the maximal-length of descending chains in $D$ [4].

### 2.3 Conditional Weighted Pushdown Systems

**Definition 2.6** A **conditional pushdown system** (CPDS) $\mathcal{P}_c$ is a 6-tuple $(Q, \Gamma, \mathcal{C}, \Delta_c, q_0, \omega_0)$, where $Q$ is a finite set of control locations, $\Gamma$ is a finite stack alphabet, $\mathcal{C}$ is a finite set of regular languages over $\Gamma$, $\Delta_c \subseteq Q \times \Gamma \times \mathcal{C} \times Q \times \Gamma^*$ is the set of transition rules, and $q_0 \in Q$ and $\omega_0 \in \Gamma^*$ are the initial control location and stack contents, respectively. We write $\langle p, \gamma\rangle \overset{L}{\hookrightarrow} \langle q, \omega\rangle$ if $(p, \gamma, L, q, \omega) \in \Delta_c$. A computation relation $\Rightarrow_c$ on configurations is defined such that $\langle p, \gamma\omega'\rangle \Rightarrow_c \langle q, \omega\omega'\rangle$ for all $\omega' \in \Gamma^*$ if $\langle p, \gamma\rangle \overset{L}{\hookrightarrow} \langle q, \omega\rangle$ and $\omega' \in L$. Given a set of configurations $C$, we define $cpre^*(C) = \{c' \mid \exists c \in C : c' \Rightarrow_c^* c\}$ and $cpost^*(C) = \{c' \mid \exists c \in C : c \Rightarrow_c^* c'\}$ to be the (possibly infinite) set of pre- and post- images of $C$, respectively.

A CPDS extends a PDS by further associating each transition with regular languages over the stack that specify conditions under which the transition can be applied

**Definition 2.7** A **conditional weighted pushdown system** $W_c$ is a triplet $(\mathcal{P}_c, S, f_c)$, where $P_c = (Q, \Gamma, \mathcal{C}, \Delta_c, q_0, \omega_0)$ is a CPDS, $S = (D, \oplus, \otimes, \bar{0}, \bar{1})$ is a bounded idempotent semiring, and $f_c : \Delta \to D$ is a weight assignment function.

Li et al. lifted the MOVP problem in Def. 2.4 to CWPDSs by replacing the underlying PDSs with CPDSs, denoted by $\mathsf{MOVP}_c(S, T)$, and showed that model checking problems can be reduced to that on WPDSs.

Given a regular language $L$, a *condition automaton* $A = (S, \Gamma, \delta, \dot{s}, F)$ with respect to $L$ is a *total deterministic finite automaton* that recognizes $L$, where $S$ is the set of states, $\Gamma$ is the input alphabet, $\delta : S \times \Gamma \to S$ is the transition function, $\dot{s}$ is the initial state, and $F \subseteq S$ is the set of final states. Let $\mathcal{C} = \{L_1, \ldots, L_n\}$ in $\mathcal{P}_c$, and let $L^R$ denote the set of reversed words of words from $L$.

$\prod_{1 \leqslant i \leqslant n} A_i$ is defined as a Cartesian product of all condition automata $A_i$ with respect to conditions $L_i^R$, and it is not hard to see being total and deterministic, i.e., a condition automaton. We denote by $\mathbf{s}_i$ the $i^{th}$ component of a state $\mathbf{s}$ in $\hat{S}$.

Let $A_i = (S_i, \Gamma, \delta_i, \dot{s}_i, F_i)$ for $1 \leqslant i \leqslant n$, and let $\prod_{1 \leqslant i \leqslant n} A_i = (\hat{S}, \Gamma, \hat{\delta}, \dot{\mathbf{s}}, \hat{F})$, where $\hat{\delta}(\mathbf{s}, \gamma) = (\delta_1(\mathbf{s}_1, \gamma), \ldots, \delta_n(\mathbf{s}_n, \gamma))$ for any $\mathbf{s} \in \hat{S}$, $\dot{\mathbf{s}} = (\dot{s}_1, \ldots, \dot{s}_n)$, and $\hat{F} = \{(s_1, \ldots, s_n) \mid \exists 1 \leqslant i \leqslant n : s_i \in F_i\}$. The authors proposed an *offline algorithm* TRANS that translates a CWPDS $\mathcal{W}_c$ to a WPDS $\mathcal{W}_t$, by synchronizing the underlying pushdown system and $\prod_{1 \leqslant i \leqslant n} A_i$. TRANS translates transitions $\delta$ in the left to $\delta'$ in the right

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

where $\mathbf{r} \in \hat{S}$, $\mathbf{r}_i \in F_i$ and $\hat{\delta}(\mathbf{r}, \gamma'') = \mathbf{t}$, and $f(\delta') = f_c(\delta)$.

A configuration $\langle p, (\gamma_n, \mathbf{r}_n)(\gamma_{n-1}, \mathbf{r}_{n-1}) \ldots (\gamma_0, \mathbf{r}_0)\rangle$ of $\mathcal{W}_t$ is *consistent* if $r_0 = \dot{\mathbf{r}}$ and $\hat{\delta}(\mathbf{r}_i, \gamma_i) = \mathbf{r}_{i+1}$ for each $1 \leqslant i \leqslant n-1$. Let $\rho(\langle p, (\gamma_n, \mathbf{r}_n)(\gamma_{n-1}, \mathbf{r}_{n-1}) \ldots (\gamma_0, \mathbf{r}_0)\rangle) = \langle p, \gamma_n \ldots \gamma_0\rangle$, and let $\rho^{-1}$ be the inverse. The computation of $\mathcal{W}_t$ over consistent configurations and the computation of $\mathcal{W}_c$ are bisimilar, and as such the model checking problems on $\mathcal{W}_c$ is reduced to $\mathcal{W}_t$ over consistent configurations. Note that the time complexity increases from the unconditional case by a factor that is polynomial in the size of $\hat{S}$, and $|\hat{S}|$ can be exponentially large.

## 3. An On-The-Fly Algorithm for CWPDSs

The offline algorithm synchronizes the underlying pushdown system and condition automata, and generates the entire space of transitions that may explode due to the potential blow-up of $|\hat{S}|$. However, a large part of the transitions in $\mathcal{W}_t$ may never be used in answering the model checking enquiry. We present an on-the-fly algorithm to compute $\mathsf{MOVP}_c(C, T)$ that synchronizes the two computing machineries on-demand while computing post-images. An on-the-fly algorithm in terms of computing pre-images can be similarly constructed. Given a $\mathcal{P}$-automaton $\mathcal{A}_C$ that recognizes a regular configuration $C$.

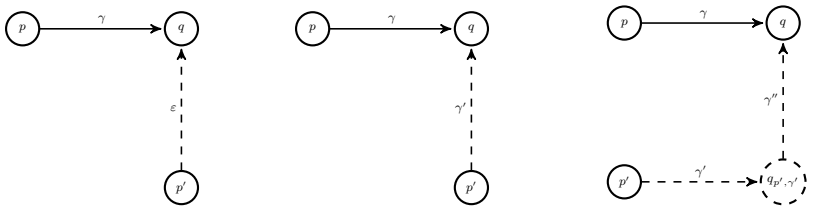
The on-the-fly model checking algorithm consists in consecutive four steps,

(1) building a $\mathcal{P}$-automaton $\mathcal{A}_{C'}$ that recognizes consistent configurations $C'$ of $\mathcal{W}_t$ such that $C' = \{\rho^{-1}(c) \mid c \in C\}$, as given in Algorithm 1, and

(2) applying the saturation procedure to computing a $\mathcal{P}$-automaton $\mathcal{A}_{cpost*(C')}$ that accepts $cpost^*(C')$, by synchronizing the pushdown system and condition automata on-demand, as given in Algorithm 2, and

(3) building a $\mathcal{P}$-automaton $\mathcal{A}_{T'}$ that recognizes consistent configurations $T'$ of $\mathcal{W}_t$ such that $T' = \{\rho^{-1}(c) \mid c \in T\}$, as done in Step (1), and computing a $\mathcal{P}$-automaton $\mathcal{B}$ by $\mathcal{B} = \mathcal{A}_{T'} \cap \mathcal{A}_{cpost*(C')}$, and

(4) computing $\mathsf{MOVP}_c(C, T)$ by running Algorithm 3 on $\mathcal{B}$, which is based on Algorithm 4 in Fig. 19 of [4].




(a) $\langle p,\gamma\rangle \hookrightarrow \langle p',\varepsilon\rangle$ (b) $\langle p,\gamma\rangle \hookrightarrow \langle p',\gamma'\rangle$ (c) $\langle p,\gamma\rangle \hookrightarrow \langle p',\gamma'\gamma''\rangle$

**Fig. 1** Saturation Rules for Computing $post^*(C)$

Note that, $\mathcal{A}_{C'}$ and $\mathcal{A}_{T'}$ are unweigthed automata, and $\mathcal{A}_{cpost*(C')}$ is a weighted automaton. They all accept consistent configurations of $\mathcal{W}_t$, although $\mathcal{W}_t$ is not explicitly constructed at all in the on-the-fly algorithm. The intersection of $\mathcal{A}_{T'}$ and $\mathcal{A}_{cpost*(C')}$ is done the same way as standard, except that the weights of $A_{cpost*(C')}$ are carried over to $\mathcal{B}$.

In the sequel, we fix a CWPDS $\mathcal{W}_c = (\mathcal{P}_c, S, f)$, where $\mathcal{P}_c = (Q,\Gamma,\mathcal{C},\Delta_c,q_0,\gamma_0)$, $\mathcal{C} = \{L_1,\ldots,L_n\}$ and $S = (D,\oplus,\otimes,\bar{0},\bar{1})$. Without loss of generality, we assume $\gamma_0 \in \Gamma$. Let $A = \{A_1,\ldots,A_n\}$ where $A_i = (S_i,\Gamma,\delta_i,\dot{s}_i,F_i)$ is the condition automaton with respect to $L_i^R$ for $1 \leqslant i \leqslant n$.

**Algorithm 1** *BuildInitial*($\mathcal{A}_C$)

**Require:** a $\mathcal{P}$-automaton $\mathcal{A}_C = (Q',\Gamma,\rightarrow,Q,F)$ that recognizes a regular set of configuration $C \subseteq Q \times \Gamma^*$, and $\mathcal{A}_C$ has no transitions into $Q$ states and has no $\varepsilon$-transitions.

**Ensure:** a $\mathcal{P}$-automaton $\mathcal{A}_{C'} = (H,\Gamma',\rightarrow',Q,F')$ that recognizes $C'$ where $C' = \{\rho^{-1}(c) \mid c \in C\}$.

$F' := \varnothing$; $\rightarrow' := \varnothing$; $visited := \varnothing$
$\dot{\mathbf{s}} := (\dot{s}_1,\dot{s}_2,\ldots,\dot{s}_n)$
**for each** $q \in F$ **do**
  $F' := F' \cup \{(q,\dot{\mathbf{s}})\}$
$H := F'$
$workset := F'$
**while** $workset \neq \varnothing$ **do**
  select and remove a state $u = (p,\mathbf{s})$ from $workset$
  $visited := visited \cup \{(p,\mathbf{s})\}$
  **for each** $(q,\gamma,p) \in \rightarrow$ **do**
    $\mathbf{t} := \hat{\delta}(\mathbf{s},\gamma)$
    $H := H \cup \{(q,\mathbf{t})\}$
    $\rightarrow' := \rightarrow' \cup \{((q,\mathbf{t}),(\gamma,\mathbf{s}),(p,\mathbf{s}))\}$
    **if** $(p,\mathbf{t}) \notin visited$ **then**
      $workset := workset \cup \{(q,\mathbf{t})\}$
**for each** $((q,\mathbf{t}),(\gamma,\mathbf{s}),(p,\mathbf{s})) \in \rightarrow'$ **do**
  **if** $q \in Q$ **then**
    $\rightarrow' := \rightarrow' \cup \{(q,(\gamma,\mathbf{s}),(p,\mathbf{s}))\}$
**return** $(H,\Gamma',\rightarrow',Q,F')$

Algorithm 1 gives a procedure that, given $\mathcal{A}_C$ that accepts a regular set $C$ of configurations, builds the $\mathcal{P}$-automaton $\mathcal{A}_{C'}$ such that $C' = \{\rho^{-1}(c) \mid c \in C\}$. Since $\mathcal{A}_C$ has no $\varepsilon$-transitions, it is not hard to conclude by the algorithm construction that $\mathcal{A}_{C'}$ has no transitions into $Q$ states and has no $\varepsilon$-transitions. By properly choosing data structures, all the needed membership test, removal operations, etc., take

**Algorithm 2** Computing $\mathcal{A}_{cpost*(C')}$ On The Fly

**Require:** a $\mathcal{P}$-automaton $\mathcal{A}_C = (Q',\Gamma,\rightarrow,Q,F)$ that recognizes a regular set of configuration $C \subseteq Q \times \Gamma^*$, and $\mathcal{A}_C$ has no transitions into $Q$ states and has no $\varepsilon$-transitions.

**Ensure:** a $\mathcal{P}$-automaton $\mathcal{A}_{cpost*(C')} = (H',\Gamma',\rightarrow',Q,F')$ that recognizes $cpost^*(C)$ of $\mathcal{W}_t$, where $C' = \{\rho^{-1}(c) \mid c \in C\}$; and a mapping $l : (\rightarrow') \rightarrow D$.

**procedure** UPDATE($t,v$)
**begin**
  $\rightarrow' := \rightarrow' \cup \{t\}$
  $newValue := l(t) \oplus v$
  $changed := (newValue \neq l(t))$
  **if** $changed$ **then**
    $workset := workset \cup \{t\}$
    $l(t) := newValue$
**end**

$\mathcal{A}_{C'} := BuildInitial(\mathcal{A}_C)$
Let $\mathcal{A}_{C'} = (H,\Gamma',\rightarrow_0,Q,F')$
$\rightarrow' := \rightarrow_0$; $workset := \rightarrow_0$; $l := \lambda t.\bar{0}$
**for all** $t \in \rightarrow_0$ **do** $l(t) := \bar{1}$
$H' := H$
**while** $workset \neq \varnothing$ **do**
  take and remove a transition $t = (p,(\gamma,\mathbf{u}),q)$ from $workset$
  **if** $\gamma \neq \varepsilon$ **then**
    **for all** $r = \langle p,\gamma\rangle \overset{A_i}{\hookrightarrow} \langle p',\varepsilon\rangle \in \Delta_c$ with $A_i = (S_i,\Gamma,\delta_i,\dot{s}_i,F_i)$ **do**
      **if** $\mathbf{u}_i \in F_i$ **then**
        UPDATE($(p',\varepsilon,q), l(t) \otimes f(r)$)
    **for all** $r = \langle p,\gamma\rangle \overset{A_i}{\hookrightarrow} \langle p',\gamma'\rangle \in \Delta_c$ with $A_i = (S_i,\Gamma,\delta_i,\dot{s}_i,F_i)$ **do**
      **if** $\mathbf{u}_i \in F_i$ **then**
        UPDATE($(p',(\gamma',\mathbf{u}),q), l(t) \otimes f(r)$)
    **for all** $r = \langle p,\gamma\rangle \overset{A_i}{\hookrightarrow} \langle p',\gamma'\gamma''\rangle \in \Delta_c$ with $A_i = (S_i,\Gamma,\delta_i,\dot{s}_i,F_i)$ **do**
      **if** $\mathbf{u}_i \in F_i$ **then**
        let $\mathbf{z} = \hat{\delta}(\mathbf{u},\gamma'')$
        $H' := H' \cup \{h_{(p',\gamma')},\mathbf{z}\}$
        UPDATE($(p',(\gamma',\mathbf{z}),(h_{(p',\gamma')},\mathbf{z})), \bar{1}$)
        UPDATE($(h_{(p',\gamma')},\mathbf{z}),(\gamma'',\mathbf{u}),q), l(t) \otimes f(r)$)
        **if** $changed$ **then**
          **for all** $t' = (p'',\varepsilon,(h_{(p',\gamma')},\mathbf{z}))$ **do**
            UPDATE($(p'',(\gamma'',\mathbf{u}),q), l(t) \otimes f(r) \otimes l(t')$)
  **else**
    **for all** $t' = (q,(\gamma',\mathbf{w}),q') \in \rightarrow$ **do**
      UPDATE($(p,(\gamma',\mathbf{w}),q'), l(t') \otimes l(t)$)
**return** $((H',\Gamma',\rightarrow',Q,F'),l)$

**Algorithm 3** $GenValue(\mathcal{B})$

**Require:** a $\mathcal{P}$-automaton $\mathcal{B} = (H, \Gamma, \rightarrow, Q, F)$ that has no transitions into $Q$ states, and a mapping $l$ from $\rightarrow$ to $D$.
**Ensure:** a value $d \in D$

```
1: for each u ∈ H\(Q ∪ F) do
2:     V_u := 0̄
3: for each u ∈ F do
4:     V_u := 1̄
5: workset := F
6: while workset ≠ ∅ do
7:     select and remove a state u from workset
8:     for each t = (s, γ, u) ∈→ do
9:         newValue := V_s ⊕ (V_u ⊗ l(t))
10:        if newValue ≠ V_s then
11:            V_s := newValue
12:            workset := workset ∪ {s}
13: val := 0̄
14: for each u ∈ Q do
15:     val := val ⊕ V_u
    return val
```

constant time. Let L be the longest length of descending chains and $t_{op}$ be the time of performing binary operators of the semiring $\mathcal{S}$. By a simple analysis over line 6 to 12, Algorithm 1 takes $O(|\rightarrow| \cdot |Q| \cdot |\hat{S}| \cdot |t_{op}| \cdot L)$ time and space.

Algorithm 2 gives the on-the-fly procedure for computing $cpost^*(C')$. The synchronization points for condition checking with the original procedure for computing $post^*$ are highlighted by underlines. Note that the procedure for computing and propagating weights is kept unchanged. It is not hard to conclude by the algorithm construction that $\mathcal{A}_{cpost*}(C')$ has no transitions into $Q$ states, given the above invariant on Algorithm 1. The algorithm extends the one for efficiently computing $A_{post*}$ (Algorithm 2 of Section 3) in [5], and the complexity increased by a factor of $n_1 = |\hat{S}| \cdot |t_{op}| \cdot L$. Algorithm 2 takes $O((|Q| \cdot |\Delta| \cdot (|H \backslash Q| + n_0) + |Q| \cdot |\ \rightarrow_0\ |) \cdot n_1)$ time and space, where $n_0$ is the number of different pairs $(p, \gamma)$ such that there exists a rule $\langle p, \gamma \rangle \hookrightarrow \langle p', \gamma'\gamma'' \rangle$ in $\Delta_c$.

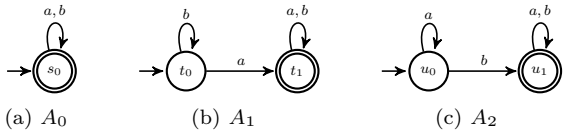
**Remark 3.1** By the construction of Algorithm 1, we have that $\mathbf{s} = \mathbf{s}'$ for each transition $(\_, (\gamma, \mathbf{s}'), (p, \mathbf{s}))$ of $\mathcal{A}_{C'}$. Based on this invariant on $A_{C'}$ and the construction of Algorithm 2, the same fact applies to transitions of $\mathcal{A}_{cpost*(C')}$. It indicates that we can further enhance the performance of the algorithm by first removing the second components of $\Gamma'$ in Algorithm 1 and 2, and then directly compute $\mathcal{B} = \mathcal{A}_T \cap \mathcal{A}_{cpost*(C')}$, instead of computing $\mathcal{A}_{T'}$ and $\mathcal{B} = \mathcal{A}_{T'} \cap \mathcal{A}_{cpost*(C')}$. Note that this optimisation would be crucial to practical efficiency. Without this optimisation, $\mathcal{A}_{T'}$ has to be computed, which, however, could easily explode.

**Example 3.2** Consider a CPDS $\mathcal{P}_c = (Q, \Gamma, \{A_0, A_1, A_2, \}, \Delta_c, p_0\ , a)$, where $Q = \{p_0, p_1, p_2\}$, $\Gamma = \{a, b\}$, and the reverse of regular languages that represent regular conditions are directly described by condition automata $A_0, A_1, A_2$, as shown in Fig. 2, where $L(A_0) = \Gamma^*$, $L(A_1) = b^*a\Gamma^*$ and $L(A_2) = a^*b\Gamma^*$, and their Cartesian product $A$ is depicted in Fig. 3.

Transitions are as follows

$$\Delta_c = \left\{ \begin{array}{lcl} \langle p_0, a \rangle & \overset{A_0}{\hookrightarrow} & \langle p_0, ba \rangle \\ \langle p_0, b \rangle & \overset{A_0}{\hookrightarrow} & \langle p_0, bb \rangle \\ \langle p_0, b \rangle & \overset{A_0}{\hookrightarrow} & \langle p_0, \varepsilon \rangle \\ \langle p_0, b \rangle & \overset{A_1}{\hookrightarrow} & \langle p_1, b \rangle \\ \langle p_0, a \rangle & \overset{A_2}{\hookrightarrow} & \langle p_2, a \rangle \end{array} \right\}$$

Given the initial $\mathcal{P}$-automaton $\mathcal{A}_S$ in Fig. 4, we are interested in knowing whether states $p_1$ and $p_2$ are reachable or not, i.e., there exists $\omega \in \Gamma^*$ such that $p_i \xrightarrow{\omega}{}^* q$ for some final state $q$ in the output $\mathcal{P}$-automaton, where $i \in \{0, 1\}$. We consider a CWPDS with $\mathcal{P}_c$ as the underlying CPDS and the semiring $(\{\bar{0}, \bar{1}\}, \oplus, \otimes, \bar{0}, \bar{1})$. The result after applying Algorithm 1 and Algorithm 2 are shown in Fig. 5 and Fig. 6, respectively. Given the initial $\mathcal{P}$-automaton, $\mathcal{P}_c$ generates configurations $\langle p_0, b^*aa \rangle$. We know $p_1$ is reachable and $p_2$ is not. The result is correctly shown by the output automaton.




(a) $A_0$ (b) $A_1$ (c) $A_2$

**Fig. 2** Condition Automata $A_0$, $A_1$ and $A_2$

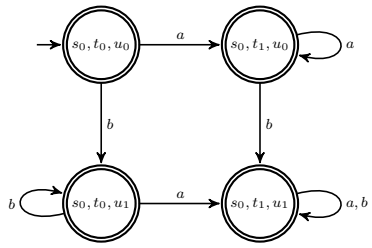



**Fig. 3** $\prod_{1 \leqslant i \leqslant n} A_i$

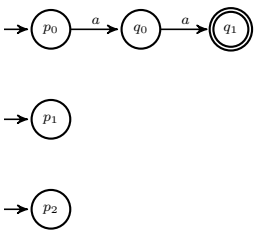



**Fig. 4** The Initial $\mathcal{P}$-automaton $\mathcal{A}_S$

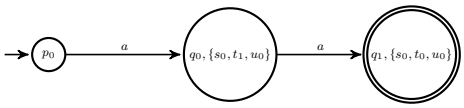



**Fig. 5** $\mathcal{A}_{S'}$ after Applying Algorithm 1

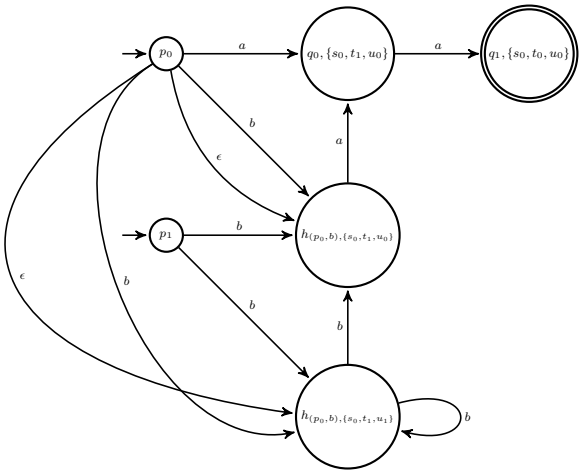



**Fig. 6** $\mathcal{A}_{cpost*(S')}$ after Applying Algorithm 2

## 4. Experiments

### 4.1 Experimental Configuration

We implement all the algorithms in Java (Java Development Kit 1.7). Our package extends the weighted pushdown system library of jMoped [6]. We use the Java automaton library developed by Anders Moller [3] to represent regular conditions associated with transitions by finite automata. The library provides facilities of a fast manipulation and easy translation from regular expressions to deterministic automata. Experiments are conducted on a system with Intel®Xeon®CPU E5-2690 0 @2.9GHz, Windows Server 2008 R2 Standard with 32GB RAM.

Comparison between the offline algorithm and the on-the-fly algorithm is conducted with a big realistic example obtained from a study of reachability analysis of HTML5 parser specifications [2], in which specifications are translated to CPDSs. Some details of the input CPDS are:

- Initial configuration $S = \{\langle 323, X\rangle\}$.
- Size of stack alphabet $|\Gamma| = 25$.
- Number of control locations $|Q| = 487$.
- Number of pushdown transitions $|\Delta| = 19679$.
- Each condition $L_i$ in the system is described by a regular expression, e.g., `((Div|(Optgroup|(Option|(P|(Rp|Ruby))))))*Li(@)`, where @ means any word. The corresponding condition automaton $A_i$ may contain up to 6 states.

### 4.2 Experimental results

**Table 1** A Comparison of Offline and On-The-Fly Algorithms

| $\|\mathcal{C}\|$ | $\|\mathcal{A}_{cpost*}\|$ | Offline algorithm | | On-the-fly algorithm | |
|---|---|---|---|---|---|
| | | *Time* | *Memory* | *Time* | *Memory* |
| 2 | 31 414 | 1s | 61 268 | 0.798s | 56 949 |
| 5 | 31 412 | 8s | 1 147 307 | 0.82s | 57 266 |
| 10 | 109 960 | $\infty$ | $\infty$ | 1s | 96 678 |
| 15 | 211 368 | $\infty$ | $\infty$ | 2s | 43 877 |
| 20 | 418 363 | $\infty$ | $\infty$ | 2s | 138 844 |
| 60 | 3 445 895 | $\infty$ | $\infty$ | 12s | 760 871 |
| 70 | 3 244 153 | $\infty$ | $\infty$ | 12s | 877 967 |
| 80 | 3 331 394 | $\infty$ | $\infty$ | 15s | 715 939 |
| 118 | 32 075 528 | $\infty$ | $\infty$ | 3m 46s | 7 410 874 |

Our experimental results are given in Table 1, which compares the space and time efficiency of the offline and on-the-fly algorithms. The first column $|\mathcal{C}|$ shows the number of conditions in CPDSs, for which we first remove conditions of all transitions and then increase conditions randomly. Recall that the number of conditions, i.e., the dimension of the product automaton, is the factor causing the state blow-up. The second column gives the size of the output $\mathcal{P}$-automaton, in terms of the number of transitions, after applying Algorithm 2. The runtime and memory consumption are estimated in seconds (s) and Kilobyte (KB), given in the next columns, and $\infty$ means either taking longer than 24 hours or getting out-of-memory.

When the number of conditions $|\mathcal{C}|$ is up to 5, the offline algorithm is still able to work, but the resource and overhead cost taken is at least two times more than that of the on-the-fly algorithm. When the number of conditions $|\mathcal{C}|$ is over 5, theoretically the product of condition DFA could explode exponentially as much as $6^{|\mathcal{C}|}$ in size, where 6 indicates the maximum number of states in a single condition DFA. It is not surprising that the offline algorithm run out of memory. Table 1 proves efficiency of the on-the-fly algorithm towards performance. The last row of Table 1 corresponds to the runtime profile of the entire example. It shows that analysing the big example up to step (2) takes about 4 minutes and 7GB memory. Note that, the memory usage takes into account garbage collection. We allocated 24GB to the Java virtual machine for running the analysis.

### 4.3 Discussion

Taking the semiring containing only unit elements $\bar{0}$ and $\bar{1}$, we computed $\mathsf{MOVP}_c(S, T)$ for target configurations $T$ generated from the study of reachability analysis of HTML5 parser specifications [2]. Since the model checking problem is EXPTIME-complete, the runtime performance of the analysis at large depends on practical instances. For instance, it always took our package less than 2 secs to analyse targets in the form of $\{\langle p, \gamma\omega\rangle \mid \omega \in \Gamma^*, \gamma \in Sym\}$, denoted by $(p, Sym\Gamma^*)$, where $p \in Q$ is some control location, and

$Sym \subseteq \Gamma$ is a set of stack symbols. For those form of targets, we can safely halt the analysis once any head of the target configurations is reachable during the saturation. However, our analysis run out of memory for other form of targets during the step (3), i.e, making intersection of $\mathcal{A}_{post*} \cap A_T$ (which may result in a quadratic increase in the automaton size), for which the garbage collector keeps up hard work to gain more space.

The algorithm in [2] computes backward saturation of target configurations and our analysis computes forward saturation from the source configurations. The advantage of our analysis is that $\mathcal{A}_{post*}(C)$ is computed once and then reused for analysing all target configurations. In contrast, the analysis in [2] need to compute $\mathcal{A}_{pre*}(T)$ for each target $T$. However, backward saturation has an inherit merit of possibly exploring smaller state space as opposed to forward saturation. We expect that our analysis would likely excel at analysing problem instances generated by the analysis and verification of object-oriented programs as originally discussed in [1], which do not contain as many as distinguished regular conditions as those in [2], and targets of interest in program analysis and verification are in those simple forms as discussed above.

## 5. Conclusion

We present an on-the-fly algorithm for model checking CWPDSs, to tackle the space explosion problem caused by its original offline algorithm. The idea is to interleave condition checking with the saturation procedures for computing post-images of regular configurations. We implemented the on-the-fly algorithm in terms of computing post-images and conducted experiments on realistic big examples from the study of reachability analysis of HTML5 parser specification and compatibility checking [2]. Our experimental results show that the on-the-fly algorithm drastically outperforms the offline algorithm regarding both space and time efficiency in practice. It would be interesting to apply our model checker to analysis and verification problems of object-oriented programs like Java.

**Acknowledgments** We would like to thank Yasuhiko Minamide for sharing us with their model and test data generated from HTML5 parser specifications. We thank Mizuhito Ogawa for comments and support. This work is partially supported by Kakenhi 25730039.

### References

[1] Li, X. and Ogawa, M.: Conditional weighted pushdown systems and applications, *Proceedings of the ACM SIGPLAN 2010 workshop on Partial evaluation and program manipulation - PEPM '10*, p. 141 (online), DOI: 10.1145/1706356.1706382 (2010).

[2] Minamide, Y. and Mori, S.: Reachability Analysis of the HTML5 Parser Specification and its Application to Compatibility Testing, *18th International symposium on formal methods* (2012).

[3] Moller, A.: dk.brics.automaton – Finite-State Automata and Regular Expressions for Java (2010).

[4] Reps, T., Schwoon, S., Jha, S. and Melski, D.: Weighted pushdown systems and their application to interprocedural dataflow analysis, *Science of Computer Programming*, Vol. 58, No. 1-2, pp. 206–263 (online), DOI: 10.1016/j.scico.2005.02.009 (2005).

[5] Schwoon, S.: Model-Checking Pushdown Systems, Ph.D. Thesis, Technische Universität München (2002).

[6] Suwimonteerabuth, D.: Reachability in Pushdown Systems: Algorithms and Applications, PhD Thesis, Technische Universität München (2009).

[7] Xin Li, H. V. L. T.: Generating Stack-based Access Control Policies, *arXiv.org*, pp. 1–18 (online), available from ⟨http://arxiv.org/abs/1307.2964v2⟩ (2013).