Symbolic Visibly Pushdown Automata*

Loris D'Antoni and Rajeev Alur

University of Pennsylvania
{lorisdan,alur}@seas.upenn.edu

Abstract. Nested words model data with both linear and hierarchical structure such as XML documents and program traces. A nested word is a sequence of positions together with a matching relation that connects open tags (calls) with the corresponding close tags (returns). Visibly Pushdown Automata are a restricted class of pushdown automata that process nested words, and have many appealing theoretical properties such as closure under Boolean operations and decidable equivalence. However, like any classical automata models, they are limited to finite alphabets. This limitation is restrictive for practical applications to both XML processing and program trace analysis, where values for individual symbols are usually drawn from an unbounded domain. With this motivation, we introduce Symbolic Visibly Pushdown Automata (SVPA) as an executable model for nested words over infinite alphabets. In this model, transitions are labeled with predicates over the input alphabet, analogous to symbolic automata processing strings over infinite alphabets. A key novelty of SVPAs is the use of binary predicates to model relations between open and close tags in a nested word. We show how SVPAs still enjoy the decidability and closure properties of Visibly Pushdown Automata. We use SVPAs to model XML validation policies and program properties that are not naturally expressible with previous formalisms and provide experimental results for our implementation.

Keywords: visibly pushdown automata, symbolic automata, XML.

1 Introduction

Nested words model data with both linear and hierarchical structure such as XML documents and program traces. A nested word is a sequence of positions together with a matching relation that connects open tags (calls) with the corresponding close tags (returns). Visibly Pushdown Languages operate over nested words, and are defined as the languages accepted by Visibly Pushdown Automata (VPA) [1,2]. It can be shown that this class is closed under Boolean operations and enjoys decidable equivalence. The model of VPA has been proven to be useful in many computational tasks, from streaming XML processing [11,13,18] to verification of recursive programs [5,12]. As many classical models, VPAs build on two basic assumptions: there is a finite state space; and there is a finite alphabet.

* This research was supported by NSF Expeditions in Computing award CCF 1138996.

A. Biere and R. Bloem (Eds.): CAV 2014, LNCS 8559, pp. 209–225, 2014.

While finiteness of the state-space is a key aspect that enables many decidable properties, the finite alphabet assumption is in general not necessary. Moreover, practical applications such as XML processing and program trace analysis, use values for individual symbols that are typically drawn from an infinite domain. This paper focuses on this limitation and proposes a way to extend VPAs to infinite domains based on the recently proposed idea of symbolic automata.

Symbolic Finite Automata (SFAs) [3,9,19] are finite state automata in which the alphabet is given by a Boolean algebra that may have an infinite domain, and transitions are labeled with predicates over such algebra. In order for SFAs to be closed under Boolean operations and preserve decidability of equivalence, it should be decidable to check whether predicates in the algebra are satisfiable. SFAs accept languages of strings over a potentially infinite domain. Although strictly more expressive than finite-state automata, Symbolic Finite Automata are closed under Boolean operations and admit decidable equivalence.

We introduce Symbolic Visibly Pushdown Automata (SVPA) as an executable model for nested words over infinite alphabets. In SVPAs transitions are labeled with predicates over the input alphabet, analogous to symbolic automata for strings over infinite alphabets. A key novelty of SVPAs is the use of binary predicates to model relations between open and close tags in a nested word. Even though SVPAs completely subsume VPAs, we show how SVPAs still enjoy the decidability and closure properties of VPAs. This result is quite surprising since previous extensions of Symbolic Automata with binary predicates have undecidable equivalence and are not closed under Boolean operations [7].

We finally investigate potential applications of SVPAs in the context of analysis of XML documents and monitoring of recursive programs over infinite domains. We show how SVPAs can model XML validation policies and program properties that are not naturally expressible with previous formalisms and provide experimental results on the performance of our implementation. For example SVPAs can naturally express the following properties: an XML document is well-matched (every close tag is the same as the corresponding open tag), every person has age greater than 5, and every person's name starts with a capital letter. Using the closure properties of SVPAs, all these properties can then be expressed as a single deterministic SVPAs that can be efficiently executed.

Contributions: In summary, our contributions are:

- the new model of Symbolic Visibly Pushdown Automata (Section 3);
- new algorithms for intersecting, complementing, and determinizing SVPAs, and for checking emptiness of SVPAs, that extend classical algorithms to the symbolic setting (Section 4); and
- a prototype implementation of SVPAs and its evaluation using XML processing and program monitoring as case-studies (Section 5).

2 Motivating Example: Dynamic Analysis of Programs

In dynamic analysis program properties are monitored at runtime. Automata theory has come handy in specifying monitors. Let x be a global variable of

a program P. We can use Finite State Automata (FSA) to describe "correct" values of x during the execution of P. For example if x has type *bool*, an FSA can specify that x starts with value $true$ and has value $false$ when P terminates.

Infinite Domains. In the previous example, x has type *bool*. In practice, one would want to express properties about variables of any type. If x is of type *int* and has infinitely many possible values, FSAs do not suffice any more. For example no FSA can express the property φ_{ev} stating that x remains even throughout the whole execution of P. One solution to this problem is that of using predicate abstraction and create an alphabet of two symbols $even(x)$ and $\neg even(x)$. However, this solution causes the input alphabet to be different from the original one ($\{even(x), \neg even(x)\}$ instead of the set of integers), and requires to choose a priori which abstraction to use.

Symbolic Finite Automata (SFA) [9,19] solve this problem by allowing transitions to be labeled with predicates over a decidable theory. Despite this, SFAs enjoy all the closure and decidability properties of finite state automata. The SFA A_{ev} for the property φ_{ev} has one state looping on an edge labeled with the predicate $even(x)$ expressible in Presburger arithmetic. Unlike predicate abstraction, SFAs do not change the underlying alphabet and allow predicates to be combined. For example, let A_{pos} be the SFA accepting all the sequences of positive integers. When intersecting A_{pos} and A_{ev} the transitions predicates will be combined, and we will obtain an SFA accepting all the sequences containing only integers that are both even and positive. An important restriction is that the underlying theory of the predicates needs to be decidable. For example, the property φ_{pr}, which states that x is a prime number at some point in P, cannot be expressed by an SFA.

SFAs allow only unary predicates and cannot relate values at different positions. Extended Symbolic Finite Automata (ESFA) [8] allow binary predicates for comparing adjacent positions, but this extension causes the model to lose closure and decidability properties [7]. Other models for comparing values over infinite alphabets at different positions are Data Automata (DA) [4] and Register Automata (RA) [6] where one can for example check that all the symbols in an input sequence are equal. This property is not expressible by an SFA or an ESFA, however Data Automata can only use equality and cannot specify properties such as $even(x)$.

Procedure Calls. Let x be of type *bool* and let's assume that the program P contains a procedure q. The following property $\varphi_{=}$ can be specified by neither an FSA nor a SFA: every time q is called, the value of x at the call is the same as the value of x when q returns. The problem is that none of the previous model is able to "remember"' which call corresponds to which return. Visibly Pushdown Automata (VPA) [2] solve this problem by storing the value of x on a stack at a call and then retrieve it at the corresponding return. Unlike classical pushdown automata, this model still enjoys closure under Boolean operations and decidable equivalence. This is achieved by making calls and returns visible in the input and allowing the stack to push only at calls and to pop only at returns.

Table 1. Properties of different automata models

Model	Bool. Closure and Decidable Equiv.	Determinizability	Hierarchical Inputs	Infinite Alphabets	Binary Predicates
FSA	✓	✓	✗	✗	—
SFA	✓	✓	✗	✓	—
ESFA	✗	✗	✗	✓	Adjacent Positions
DA, RA	some variants	✗	trees	✓	Only Equality
VPA	✓	✓	✓	✗	—
SVPA (this paper)	✓	✓	✓	✓	Calls/Returns

Procedure Calls and Infinite Domains. Let x be of type *int* and let's assume that the program P contains a procedure q. No VPA can express the property $\psi_<$ requiring that, whenever q is called the value of x at the call is smaller than the value of x at the corresponding return. Expressing this kind of property in a decidable automaton model is the topic of this paper.

We introduce Symbolic Visibly Pushdown Automata (SVPA) that combine the features of SFAs and VPAs by allowing transitions to be labeled with predicates over any decidable theory and values to be stored on a stack at calls and retrieved at the corresponding returns. The property $\psi_<$ can then be expressed by an SVPA $A_<$ as follows. At a procedure call of q, $A_<$ will store the value c of x on the stack. When reading the value r of x at a procedure return of q, the value c of x at the corresponding call will be on top of the stack. Using the predicate $c < r$, the transition assures that the property $\psi_<$ is met. SVPAs still enjoy closure under Boolean operations, determinizability, and decidable equivalence, and the key to decidability is that binary predicates can only be used to compare values at matching calls and returns (unlike ESFAs). Data Automata and Register Automata have been extended to trees and grammars [4,6,14] but their expressiveness, for the same reason we discussed for strings, is orthogonal to that of SVPAs. Table 1 summarizes the properties of all the models we discussed.

3 Symbolic Visibly Pushdown Automata

In this section we formally define Symbolic Visibly Pushdown Automata (SVPA). We first provide some preliminary definitions for symbolic alphabets. Next we recall the basic definition of tagged alphabet and nested words, and we extend such definition to infinite alphabets. Last, we define SVPAs and their semantics.

3.1 Preliminaries

We use standard first-order logic and follow the notational conventions that are consistent with the original definition of symbolic transducers [21]. We write Σ for the input alphabet. A *label theory* is given by a recursively enumerable set Ψ of formulas that is closed under Boolean operations. We use $\mathbb{P}_x(\Psi)$ and $\mathbb{P}_{x,y}(\Psi)$ to denote the set of unary and binary predicates in Ψ respectively. We assume that every unary predicate in $\mathbb{P}_x(\Psi)$ contains x as the only free variable (similarly

$\mathbb{P}_{x,y}(\Psi)$ with x and y). It is easy to observe that given two unary predicates $\varphi_1, \varphi_2 \in \mathbb{P}_x(\Psi)$, the predicates $\varphi_1 \wedge \varphi_2$ and $\neg\varphi_1$ are also unary predicates in $\mathbb{P}_x(\Psi)$, and given a predicate $\varphi_1 \in \mathbb{P}_x(\Psi) \cup \mathbb{P}_{x,y}(\Psi)$ and a binary predicate $\varphi_2 \in \mathbb{P}_{x,y}(\Psi)$ the predicates $\varphi_1 \wedge \varphi_2$ and $\neg\varphi_2$ are also binary predicates in $\mathbb{P}_{x,y}(\Psi)$. A predicate $\varphi \in \mathbb{P}_x$ (resp. $\varphi \in \mathbb{P}_{x,y}$) is satisfiable, $IsSat(\varphi)$, if there exists a *witness* $a \in \Sigma$ (resp. $(a,b) \in \Sigma \times \Sigma$) that when substituted to x makes φ true, $[\![\varphi[a/x]]\!] = true$ (resp. $[\![\varphi[a/x, b/y]]\!] = true$). A label theory Ψ is *decidable* when, for any $\varphi \in \Psi$, checking whether $IsSat(\varphi)$ is true is decidable.

Nested words. Data with both linear and hierarchical structure can be encoded using nested words [2]. Given a set Σ of symbols, the *tagged alphabet* $\hat{\Sigma}$ consists of the symbols a, $\langle a$, and $a\rangle$, for each $a \in \Sigma$. A *nested word* over Σ is a finite sequence over $\hat{\Sigma}$. For a nested word $a_1 \cdots a_k$, a position j, for $1 \leq j \leq k$, is said to be a *call* position if the symbol a_j is of the form $\langle a$, a *return* position if the symbol a_j is of the form $a\rangle$, and an *internal* position otherwise. The tags induce a matching relation between call and return positions. Nested words can naturally encode strings and ordered trees.

3.2 Model

We can now formally define the model of symbolic visibly pushdown automata.

Definition 1 (SVPA). *A (nondeterministic) symbolic visibly pushdown automaton over an alphabet Σ is a tuple $A = (Q, Q_0, P, \delta_i, \delta_c, \delta_r, Q_F)$, where*

- *Q is a finite set of states,*
- *$Q_0 \subseteq Q$ is a set of initial states,*
- *P is a finite set of stack symbols,*
- *$\delta_i \subseteq Q \times \mathbb{P}_x \times Q$ is a finite set of internal transitions*
- *$\delta_c \subseteq Q \times \mathbb{P}_x \times Q \times P$, is a finite set of call transitions,*
- *$\delta_r \subseteq Q \times \mathbb{P}_{x,y} \times P \times Q$, is a finite set of return transitions,*
- *$\delta_b \subseteq Q \times \mathbb{P}_x \times Q$, is a finite set of empty-stack return transitions, and*
- *$Q_F \subseteq Q$ is a set of accepting states.*

A transition $(q, \varphi, q') \in \delta_i$, where $\varphi \in \mathbb{P}_x$, when reading a symbol a such that $a \in [\![\varphi]\!]$, starting in state q, updates the state to q'. A transition $(q, \varphi, q', p) \in \delta_c$, where $\varphi \in \mathbb{P}_x$, and $p \in P$, when reading a symbol $\langle a$ such that $a \in [\![\varphi]\!]$, starting in state q, pushes the symbol p on the stack along with the symbol a, and updates the state to q'. A transition $(q, \varphi, p, q') \in \delta_r$, where $\varphi \in \mathbb{P}_{x,y}$, is triggered when reading an input $b\rangle$, starting in state q, and with $(p, a) \in P \times \Sigma$ on top of the stack such that $(a, b) \in [\![\varphi]\!]$; the transition pops the element on the top of the stack and updates the state to q'. A transition $(q, \varphi, q') \in \delta_b$, where $\varphi \in \mathbb{P}_x$, is triggered when reading a tagged input $a\rangle$ such that $a \in [\![\varphi]\!]$, starting in state q, and with the current stack being empty; the transition updates the state to q'.

A stack is a finite sequence over $P \times \Sigma$. We denote by Γ the set of all stacks. Given a nested word $w = a_1 \ldots a_k$ in Σ^*, a run of M on w starting in state q is a sequence $\rho_q(w) = (q_1, \theta_1), \ldots, (q_{k+1}, \theta_{k+1})$, where $q = q_1$, each $q_i \in Q$, each

$\theta_i \in \Gamma$, the initial stack θ_1 is the empty sequence ε, and for every $1 \leq i \leq k$ the following holds:

Internal if a_i is internal, there exists $(q, \varphi, q') \in \delta_i$, such that $q = q_i$, $q' = q_{i+1}$, $a_i \in [\![\varphi]\!]$, and $\theta_{i+1} = \theta_i$;
Call if $a_i = \langle a$, for some a, there exists $(q, \varphi, q', p) \in \delta_c$, such that $q = q_i$, $q' = q_{i+1}$, $a \in [\![\varphi]\!]$, and $\theta_{i+1} = \theta_i(p, a)$; and
Return if $a_i = a\rangle$, for some a, there exists $(q, \varphi, p, q') \in \delta_r$, $b \in \Sigma$, and $\theta' \in \Gamma$, such that $q = q_i$, $q' = q_{i+1}$, $\theta_i = \theta'(p, b)$, $\theta_{i+1} = \theta'$, and $(b, a) \in [\![\varphi]\!]$.
Bottom if $a_i = a\rangle$, for some a, there exists $(q, \varphi, q') \in \delta_b$, such that $q = q_i$, $q' = q_{i+1}$, $\theta_i = \theta_{i+1} = \varepsilon$, and $a \in [\![\varphi]\!]$.

A run is *accepting* if q_1 is an initial state in Q_0 and q_{k+1} is a final state in F. A nested word w is accepted by A if there exists an accepting run of A on w. The language $L(A)$ accepted by A is the set of nested words accepted by A.

Definition 2 (Deterministic SVPA). *A symbolic visibly pushdown automaton A is deterministic iff $|Q_0| = 1$ and*

- *for each two transitions $t_1 = (q_1, \varphi_1, q_1'), t_2 = (q_2, \varphi_2, q_2') \in \delta_i$, if $q_1 = q_2$ and $IsSat(\varphi_1 \wedge \varphi_2)$, then $q_1' = q_2'$;*
- *for each two transitions $t_1 = (q_1, \varphi_1, q_1', p_1), t_2 = (q_2, \varphi_2, q_2', p_2) \in \delta_c$, if $q_1 = q_2$ and $IsSat(\varphi_1 \wedge \varphi_2)$, then $q_1' = q_2'$ and $p_1 = p_2$;*
- *for each two transitions $t_1 = (q_1, \varphi_1, p_1, q_1'), t_2 = (q_2, \varphi_2, p_2, q_2') \in \delta_r$, if $q_1 = q_2$, $p_1 = p_2$, and $IsSat(\varphi_1 \wedge \varphi_2)$, then $q_1' = q_2'$; and*
- *for each two transitions $t_1 = (q_1, \varphi_1, q_1'), t_2 = (q_2, \varphi_2, q_2') \in \delta_b$, if $q_1 = q_2$, and $IsSat(\varphi_1 \wedge \varphi_2)$, then $q_1' = q_2'$.*

For a deterministic SVPA A we use q_0 to denote the only initial state of A.

Definition 3 (Complete SVPA). *A deterministic symbolic visibly pushdown automaton A is complete iff for each $q \in Q$, $a, b \in \Sigma$, and $p \in P$, there exist 1) a transition $(q, \varphi, q') \in \delta_i$, such that $a \in [\![\varphi]\!]$; 2) a transition $(q, \varphi, q', p') \in \delta_c$, such that $a \in [\![\varphi]\!]$; 3) a transition $(q, \varphi, p, q') \in \delta_r$, such that $(a, b) \in [\![\varphi]\!]$; and 4) a transition $(q, \varphi, q') \in \delta_b$, such that $a \in [\![\varphi]\!]$.*

4 Closure Properties and Decision Procedures

In this section we describe the closure and decidability properties of SVPAs. We first introduce few preliminary concepts and then show how SVPAs are equivalent in expressiveness to deterministic SVPAs, and complete SVPAs. We then prove that SVPAs are closed under Boolean operations. Last, we provide an algorithm for checking emptiness of SVPAs over decidable label theories and use it to prove the decidability of SVPA language equivalence. For each construction we provide a complexity parameterized by the underlying theory, and we assume that transitions are only added to a construction when satisfiable.

4.1 Closure Properties

Before describing the determinization algorithm we introduce the concept of a minterm. The notion of a minterm is fundamental for determinizing symbolic automata, and it captures the set of equivalence classes of the input alphabet for a given symbolic automaton. Intuitively, for every state q of the symbolic automaton, a minterm is a set of input symbols that q will always treat in the same manner. Given a set of predicates Φ a *minterm* is a minimal satisfiable Boolean combination of all predicates that occur in Φ. We use the notation $Mt(\Phi)$ to denote the set of minterms of Φ. For example the set of predicates $\Phi = \{x > 2, x < 5\}$ over the theory of linear integer arithmetic has minterms $Mt(\Phi) = \{x > 2 \wedge x < 5,\ \neg x > 2 \wedge x < 5,\ x > 2 \wedge \neg x < 5\}$. While in the case of symbolic finite automata this definition is simpler (see [9]), in our setting we need to pay extra attention to the presence of binary predicates. We need therefore to define two types of minterms, one for unary predicates and one for binary predicates. Given an SVPA A we define

- the set Φ_1^A of unary predicates of A as the set $\{\varphi \mid \exists q, q', p.(q, \varphi, q') \in \delta_i \vee (q, \varphi, q', p) \in \delta_c \vee (q, \varphi, q') \in \delta_b\}$;
- the set Φ_2^A of binary predicates of A as the set $\{\varphi \mid \exists q, q', p.(q, \varphi, p, q') \in \delta_r\}$;
- the set Mt_1^A as the set $Mt(\Phi_1^A)$ of unary predicate minterms of A; and
- the set Mt_2^A as the set $Mt(\Phi_2^A)$ of binary predicate minterms of A.

The goal of minterms is that of capturing the equivalence classes of the label theory in the current SVPA. Let Φ be the set of minterms of an SVPA A. Consider two nested words $s = a_1 \dots a_n$ and $t = b_1 \dots b_n$ of equal length and such that for every i, a_i has the same tag as b_i (both internals, etc.). Now assume the following is true: for every $1 \leq i \leq n$, if a_i is internal there exists a minterm $\varphi \in Mt_1^A$ such that both a_i and b_i are models of φ, and, if a_i is a call with corresponding return a_j, then there exists a minterm $\psi \in Mt_2^A$ such that both (a_i, a_j) and (b_i, b_j) are models of ψ. If the previous condition holds, the two nested words will be indistinguishable in the SVPA A, meaning that they will have exactly the same set of runs. Following, this intuition we have that even though the alphabet might be infinite, only a finite number of predicates is *interesting*. We can now discuss the determinization construction.

Theorem 1 (Determinization). *For every SVPA A there exists a deterministic SVPA B accepting the same language.*

Proof. The main difference between the determinization algorithm in [2] and the symbolic version is in the use of minterms. Similarly to the approach presented in [9], we use the minterm computation to generate a finite set of relevant predicates. After we have done so, we can generalize the determinization construction shown in [2].

We now describe the intuition behind the construction. Given a nested word n, A can have multiple runs over n. Thus, at any position, the state of B needs to keep track of all possible states of A, as in case of classical subset construction for determinization of nondeterministic word automata. However, keeping only

a set of states of A is not enough: at a return position, B needs to use the information on the top of the stack and in the state to figure out which pairs of states (starting in state q you can reach state q') belong to the same run. The main idea behind the construction is to do a subset construction over summaries (pairs of states) but postpone handling the call-transitions by storing the set of summaries before the call, along with the minterm containing the call symbol, in the stack, and simulate the effect of the corresponding call-transition at the time of the matching return for every possible minterm.

The components of the deterministic automaton B equivalent to $A = (Q, Q_0, P, \delta_c, \delta_i, \delta_r, Q_F)$ are the following. The states of B are $Q' = 2^{Q \times Q}$. The initial state is the set $Q_0 \times Q_0$ of pairs of initial states. A state $S \in Q'$ is accepting iff it contains a pair of the form (q, q') with $q' \in Q_f$. The stack symbols of B are $P' = Q' \times Mt_1^A$. The internal transition function δ_i' is given by: for $S \in Q'$, and $\varphi \in Mt_1^A$, $\delta_i'(S, \varphi)$ consists of pairs (q, q'') such that there exists $(q, q') \in S$ and an internal transition $(q', \varphi', q'') \in \delta_i$ such that $IsSat(\varphi \wedge \varphi')$. The call transition function δ_c' is given by: for $S \in Q'$ and $\varphi \in Mt_1^A$, $\delta_c'(S, \varphi) = (S', (S, \varphi))$, where S' consists of pairs (q'', q'') such that there exists $(q, q') \in S$, a stack symbol $p \in P$, and a call transition $(q', \varphi', q'', p) \in \delta_c$ such that $IsSat(\varphi \wedge \varphi')$. The return transition function δ_r' is given by: for $S, S' \in Q'$ and $\varphi_1 \in Mt_1^A, \varphi_2 \in Mt_2^A$, the state $\delta_r'(S, (S', \varphi_1), \varphi_2)$ consists of pairs (q, q'') such that there exists $(q, q') \in S'$, $(q_1, q_2) \in S$, a stack symbol $p \in P$, a call transition $(q', \varphi_1', q_1, p) \in \delta_c$, and a return transition $(q_2, p, \varphi_2', q'') \in \delta_r$ such that $IsSat(\varphi_1 \wedge \varphi_1')$ and $IsSat(\varphi_2 \wedge \varphi_2')$. The empty-stack return transition function δ_b' is given by: for $S \in Q'$ and $\varphi \in Mt_1^A$, the state $\delta_b'(S, \varphi)$ consists of pairs (q, q'') such that there exists $(q, q') \in S$ and a return transition $(q', \varphi', q'') \in \delta_b$ such that $IsSat(\varphi \wedge \varphi')$. Our construction differs from the one in [2] in two aspects:

- in [2] each stack symbol contains an element from Σ. This technique cannot be used in our setting and in our construction each stack symbol contains a predicate from the set of unary minterms.
- the construction in [2] builds on the notion of reachability and looks for matching pairs of calls and returns. In our construction, this operation has to be performed symbolically, by checking whether the unary predicate stored by the call on the stack and the binary predicate at the return are not disjoint.

We finally discuss the complexity of the determinization procedure. Assume A has n states, m stack symbols, and p different predicates of size at most ℓ. We first observer that the number of minterms is at most 2^p and each minterm has size $O(p\ell)$.[1] If $f(a)$ is the cost of checking the satisfiability of a predicate of size a, then the minterm computation has complexity $O(2^p f(\ell p))$. The resulting automaton B has $O(2^{n^2})$ states, and $O(2^p 2^{n^2})$ stack symbols. The determinization procedure has worst complexity $O(2^p 2^{n^2} m + 2^p f(p\ell))$.

Theorem 2 (Completeness). *For every SVPA A there exists a complete SVPA B accepting the same language.*

[1] If the alphabet is finite the number of minterms is bounded by $min(2^p, |\Sigma|)$.

Proof. Since the procedure is trivial we only discuss its complexity. Assume A has n states, m stack symbols, and p different predicates of size at most ℓ. Let $f(a)$ be the cost of checking the satisfiability of a predicate of size a. The procedure has complexity $O(nmf(\ell p))$. ⊠

Theorem 3 (Boolean Closure). *SVPAs are closed under Boolean operations.*

Proof. We prove that SVPAs are closed under complement and intersection. We first prove that SVPAs are closed under complement. Given an SVPA A we construct a complete SVPA C such that C accepts a nested word n iff n is not accepted by A. First, we use Theorem 2 to construct an equivalent deterministic SVPA $B = (Q, q_0, P, \delta_i, \delta_c, \delta_r, Q_F)$. We can now construct the SVPA $C = (Q, q_0, P, \delta_i, \delta_c, \delta_r, Q \setminus Q_F)$ in which the set of accepting states is complemented.

We next prove that SVPAs are closed under intersection. Given two deterministic SVPAs $A_1 = (Q^1, q_0^1, P^1, \delta_i^1, \delta_c^1, \delta_r^1, Q_F^1)$ and $A_2 = (Q^2, q_0^2, P^2, \delta_i^2, \delta_c^2, \delta_r^2, Q_F^2)$ (using Theorem 1) we construct an SVPA B such that B accepts a nested word n iff n is accepted by both A_1 and A_2. The construction of B is a classical product construction. The SVPA B will have state set $Q' = Q^1 \times Q^2$, initial state $q_0' = (q_0^1, q_0^2)$, stack symbol set $P' = P^1 \times P^2$, and final state set $Q'_F = Q_F^1 \times Q_F^2$, The transition function will simulate both A_1 and A_2 at the same time.

- for each $(q_1, \varphi_1, q_1') \in \delta_i^1$, and $(q_2, \varphi_2, q_2') \in \delta_i^2$, δ_i' will contain the transition $((q_1, q_2), \varphi_1 \wedge \varphi_2, (q_1', q_2'))$;
- for each $(q_1, \varphi_1, q_1', p_1) \in \delta_c^1$, and $(q_2, \varphi_2, q_2', p_2) \in \delta_c^2$, and δ_c' will contain the transition $((q_1, q_2), \varphi_1 \wedge \varphi_2, (q_1', q_2'), (p_1, p_2))$;
- for each $(q_1, \varphi_1, p_1, q_1') \in \delta_r^1$, and $(q_2, \varphi_2, p_2, q_2') \in \delta_r^2$, δ_r' will contain the transition $((q_1, q_2), \varphi_1 \wedge \varphi_2, (p_1, p_2), (q_1', q_2'))$; and
- for each $(q_1, \varphi_1, q_1') \in \delta_b^1$, and $(q_2, \varphi_2, q_2') \in \delta_b^2$, δ_b' will contain the transition $((q_1, q_2), \varphi_1 \wedge \varphi_2, (q_1', q_2'))$;

Assume each SVPA A_i has n_i states, m_i stack symbols, and p_i different predicates of size at most ℓ_i. Let $f(a)$ be the cost of checking the satisfiability of a predicate of size a. The intersection procedure has complexity $O(n_1 n_2 m_1 m_2 + p_1 p_2 f(\ell_1 + \ell_2))$.

4.2 Decision Procedures

We conclude this section with an algorithm for checking emptiness of SVPAs over decidable label theories, which we finally use to prove the decidability of SVPA equivalence.

Theorem 4 (Emptiness). *Given an SVPA A over a decidable label theory it is decidable whether $L(A) = \emptyset$.*

Proof. The algorithm for checking emptiness is a symbolic variant of the algorithm for checking emptiness of a pushdown automaton. We are given an SVPA $A = (Q, q_0, P, \delta_i, \delta_c, \delta_r, Q_F)$ over a decidable theory Ψ. First, for every

two states $q, q' \in Q$ we compute the reachability relation $R_{wm} \subseteq Q \times Q$ such that $(q, q') \in R_{wm}$ iff there exists a run $\rho_q(w)$ that, staring in state q, after reading a well matched nested word w, ends in state q'. We define R_{wm} as follows:

- for all $q \in Q$, $(q, q) \in R_{wm}$;
- if $(q_1, q_2) \in R_{wm}$, and there exists $q, q' \in Q$, $p \in P$, $\varphi_1 \in \mathbb{P}_x(\Psi), \varphi_2 \in \mathbb{P}_{x,y}(\Psi)$, such that $(q, \varphi_1, q_1, p) \in \delta_c$, $(q_2, \varphi_2, p, q') \in \delta_r$, and $IsSat(\varphi_1 \wedge \varphi_2)$, then $(q, q') \in R_{wm}$. Observe that unary and binary predicates unify on the first variable x;
- if $(q_1, q_2) \in R_{wm}$, and there exists $q \in Q$, $\varphi \in \mathbb{P}_x(\Psi)$, such that $(q, \varphi, q_1) \in \delta_i$, and $IsSat(\varphi)$, then $(q, q_2) \in R_{wm}$; and
- if $(q_1, q_2) \in R_{wm}$ and $(q_2, q_3) \in R_{wm}$, then $(q_1, q_3) \in R_{wm}$.

The above reachability relation captures all the runs over well-matched nested words. Unmatched calls and returns can be handled using a similar set of rules. We can then compute therefore the reachability relation $R \subseteq Q \times Q$ such that $(q, q') \in R$ iff there exists a run $\rho_q(w)$ that ends in state q' after reading a nested word w. The SVPA A is empty iff $(Q_0 \times Q_F) \cap R = \emptyset$.

Assume A has n states, m stack symbols, t transitions, and p predicates of size at most ℓ. Let $f(a)$ be the cost of checking the satisfiability of a predicate of size a. The emptiness procedure has complexity $O(n^3mt + p^2 f(\ell))$.

We can now combine the closure under Boolean operations and the decidability of emptiness to show that equivalence of SVPAs is decidable.

Corollary 1 (Equivalence). *Given two SVPAs A and B over a decidable label theory it is decidable whether $L(A) \subseteq L(B)$ and whether $L(A) = L(B)$.*

Complexity: In [2] it is shown that the VPA universality, inclusion, and equivalence problems are ExpTime-hard. If the function $IsSat()$ can be computed in polynomial time the same complexity bounds hold for SVPAs. ⊠

5 Applications and Evaluation

In this section we present potential applications of SVPAs together with experimental results. First, we illustrate how the presence of symbolic alphabets and closure properties enables complex XML validation, HTML sanitization, and runtime monitoring of recursive programs. Finally, we present some experimental results on SVPA's execution and algorithms.[2]

5.1 XML Validation

XML (and HTML) documents are ubiquitous. Validating an XML document is the task of checking whether such a document meets a given specification. XML Schema is the most common language for writing XML specifications and their properties have been studied in depth [17,22]. The XML schema S shown in

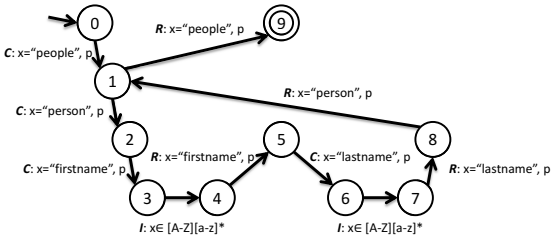
[2] All the experiments were run on a 4 Cores Intel i7-2600 CPU 3.40GHz, with 8GB of RAM. The library is configured for 32 bits architecture.

```
<xs:schema>
<xs:element name="people"  type="PeopleType"/>
<xs:complexType name="PeopleType"><xs:sequence>
 <xs:element name="person" minOccurs="0" maxOccurs="unbounded">
  <xs:complexType><xs:sequence>
   <xs:element name="firstname">
    <xs:simpleType><xs:restriction base="xs:string">
     <xs:pattern value="[A-Z]([a-z])*"/>
    </xs:restriction></xs:simpleType>
   </xs:element>
   <xs:element name="lastname">
    <xs:simpleType><xs:restriction base="xs:string">
     <xs:pattern value="[A-Z]([a-z])*"/>
    </xs:restriction></xs:simpleType>
   </xs:element>
  </xs:sequence></xs:complexType>
 </xs:element>
</xs:sequence></xs:complexType>
</xs:schema>
```

(1) XML Schema S

```
<people>
 <person>
  <firstname>
    Mark
  </firstname>
  <lastname>
    Red
  </lastname>
 </person>
 <person>
  <firstname>
    Mario
  </firstname>
  <lastname>
    Rossi
  </lastname>
 </person>
</people>
```

(2) Document Example



(3) SVPA A_s

Fig. 1. (1) XML Schema S describing documents containing a person with a first and last name, (2) an XML document accepted by S, and 3) an SVPA A_S accepting the same XML documents as S.

Figure 1 describes the format of XML documents containing first and last names. In words the document should start with the tag `people` and then contain a sequence of `person` each of which has a first and last name. First and last name should be both strings belonging to the regular expression `[A-Z]([a-z])*`.

Dealing with infinite alphabets. Although the XML Schema in Figure 1 only uses a finite set of possible nodes (`people`, `firstname`, etc.), it allows the content of the leaves to be any string accepted by the regular expression `[A-Z]([a-z])*`. This kind of constraints can be easily captured using an SVPA over the theory of strings. Such a SVPA A_S is depicted in Figure 1.3.[3] The letters *I, C*, and *R* on each transition respectively stand for internal, call, and return transitions.

Although in this particular setting the alphabet could be made finite by linearizing each string, such encoding would not be natural and would cause the corresponding VPA to be very complex. Moreover previous models that use such an encoding, such as [16,17], require the parser to further split each node value into separate characters. In the case of SVPAs, as it can be observed in Figure 1,

[3] We encode each XML document as a nested word over the theory of strings. For each open tag, close tag, attribute, attribute value, and text node the nested word contains one input symbol with the corresponding value.

there is a clear separation between the constraints on the tree structure (captured by the states), and the constraints on the leaves (captured by the predicates). This natural representation makes the model more succinct and executable since it reflects the typical representation of XML via events (SAX parser, etc.).

5.2 HTML Filters

A central concern for secure web application is untrusted user inputs. These lead to cross-site scripting (XSS) attacks, which may echo an untrusted input verbatim back to the browser. HTML filters aim at blocking potentially malicious user HTML code from being executed on the server. For example, a security sensitive application might want to discard all documents containing `script` nodes which might contain malicious JavaScript code (this is commonly done in HTML sanitization). Since HTML5 allows to define custom tags, the set of possible node names is infinite and cannot be known a priori. In this particular setting, an HTML schema would not be able to characterize such an HTML filter. This simple property can be checked using an SVPA over the theory of strings. Such an SVPA A is depicted on the left of Figure 2. The SVPA A only accepts nested words that do not contain `script` nodes. Notice that the call transition is triggered by any string different from `script` and the alphabet is therefore infinite.

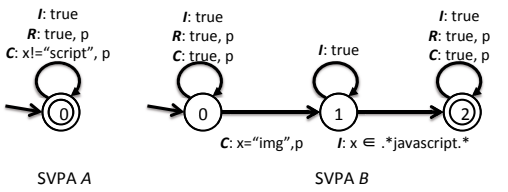


Fig. 2. The SVPA A rejects HTML documents that contain scripts, while the SVPA B accepts the documents containing malicious images

Since SVPAs can be intersected, complemented, and determinized, we can take advantage of these properties to make the design of HTML filters modular. We now consider an example for which it is much simpler to specify what it means for a document to be malicious rather than to be safe. On the right of Figure 2 it is shown a non-deterministic SVPA B for checking whether a `img` tag may call JavaScript code in one of its attributes. To compute our filter (the set of safe inputs) we can now compute the complement B' of B that only accepts HTML documents that do not contain malicious `img` tags.

We can now combine A and B' into a single filter. This can be easily done by computing the intersection $F = A \cap B'$. If necessary, the SVPA F can then be determinized, obtaining an executable filter that can efficiently process HTML documents with a single left-to-right pass.

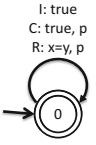


Fig. 3. SVPA W accepting HTML documents with matching open and close tags

The power of binary predicates. The previous HTML filter is meant to process only well-formed HTML documents. A well-formed HTML document is one in which all the open tags are correctly matched (every open tag

is closed by a close tag containing the same symbol). In practice the input documents goes first through a well-formedness checker and then through a filter. This causes the input HTML to be processed multiple times and in performance critical applications this is not feasible. This check can however be performed by the SVPA W in Figure 3.

5.3 Runtime Program Monitors

We already discussed in Section 2 how SVPAs are useful for defining monitors for dynamic analysis of programs. In this section we present an example of how SVPAs can be used to express complex properties about programs over infinite domains such as integers. Consider the recursive implementation of Fibonacci on the right. Let's assume we are interested into monitoring the values of x at every call of Fib, and the values returned by Fib. For example for the input 5, our monitored nested word will be $\langle 2\ \langle 1\ 1\rangle\ \langle 0\ 0\rangle\ 1\rangle$. The following properties can all be expressed using SVPAs:

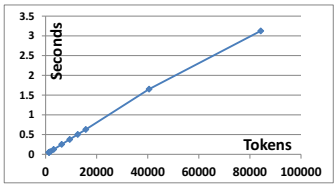
```
function Fib(int x)
  if x < 2 then return x
  return Fib(x − 1) + Fib(x − 2)
```

1. if the input of Fib is greater or equal than 0, then the same hold for all the subsequent inputs of Fib;
2. if the output of Fib is negative, than Fib was called exactly once in the whole execution and with a negative input;
3. the output of Fib is greater or equal than the corresponding input.

We can then intersect the SVPAs corresponding to each property and generate a single pass linear time monitor for Fib. As we discussed in Section 2, SVPAs cannot express properties that relate all the values in the computation such as: the value of a variable x increases monotonically throughout the computation. However, thanks to the presence of binary predicates at returns, SVPAs provide a model for describing pre and post conditions of programs over decidable theories (see property 3). In particular, SVPAs can describe post conditions that relate the values of the inputs and the outputs of a function.

5.4 Experimental Results

Execution performance. We implemented the filter $F = A \cap \bar{B} \cap W$ and analyzed the performance of filtering HTML documents with size between 8 and 1293 KB, depth between 3 and 11, number of tokens between 1305 and 84242, and average token length between 11 and 14 characters.[4] Constructing the SVPA F took 435 milliseconds. The running times



[4] To solve the underlying theory of equality plus regular constraints, we implemented a solver on top of the Microsoft Automata library [20]. We decided not to use a full blown solver such as Hampi [15], since it was not necessary in our setting.

per number of tokens (in seconds) are shown in the figure on the right. We observed that the depth of the input does not affect the running time, while the length affects it linearly. Surprisingly, the running time is also not affected by the average length of the tokens. This is due to the fact that most tokens can be rejected by partially reading them.

Algorithms performance: data. We evaluated the determinization and equivalence algorithms on a representative set of SVPAs over three different alphabet theories: strings, integers, and bitvectors (characters).[5] For each theory t we generated an initial set of SVPAs S_1^t containing 5 nondeterministic SVPAs for properties of the following form: 1) the input contains a call and matching return with different symbols; 2) the input contains an internal symbol satisfying a predicate φ_0; 3) the input contains a subword $\langle a\ b\ c\rangle$ such that $a \in [\![\varphi_1]\!]$, $b \in [\![\varphi_2]\!]$, and $a = c$; 4) the input contains a subword $\langle a\ \langle b$ such that $a \in [\![\varphi_3]\!]$, $b \in [\![\varphi_4]\!]$; and 5) for every internal symbol a in the input, $a \in [\![\varphi_4]\!]$, or $a \in [\![\varphi_5]\!]$. The predicates $\Phi = \{\varphi_0, \ldots, \varphi_5\}$ vary for each theory and are all different. For each theory we then computed the set $S_2^t = \{A \cap B \mid A, B \in S_1^t\} \cup \{A \cap B \cap C \mid A, B, C \in S_1^t\}$. We then used the sets S_2^t to evaluate the determinization algorithm, and computed the corresponding set of deterministic SVPAs D_2^t. Finally we checked equivalence of any two SVPAs A and B in D_2^t.

The results of our experiments are shown in Figure 4: the left column shows the size of each test set and the number of instances for which the algorithms timed out (5 minutes). The right column shows the running time for the instances in which the algorithms did not time out. For both algorithms we plot against number of states and number of transitions. For the determinization, the sizes refer to the automaton before determinization, while in the case of equivalence, the sizes refer to the sum of the corresponding metrics of the two input SVPAs. The distribution of the sizes of the SVPAs differed slightly when varying the theories, but since the differences are very small we show the average sizes. For each theory we determinized a total of 65 SVPAs and checked for equivalence 241 pairs of SVPAs. For both operation, on average, 96% of the time is spent in the theory solver. For the theory of characters we also compared our tool to the VPALib library, a Java implementation of VPAs.[6] The application timed out for all the inputs considered in our experiments.

Algorithms performance: data analysis. Except for few instances involving the theory of integers, our implementation was able to determinize all the considered SVPAs in less than 5 minuets. The situation was different in the case of equivalence, where most of the input pairs with more than 250 transitions or 13 states timed out. Most of such pairs were required to check satisfiability of more than 15000 predicates that were generated when building the intersected SVPAs necessary to check equivalence. We could observe that the theory of characters is

[5] The characters and strings solver are implemented on top of the Automata library [20] which is based on BDDs, while the integer solver is Z3 [10].

[6] Available `http://www.emn.fr/z-info/hnguyen/vpa/`

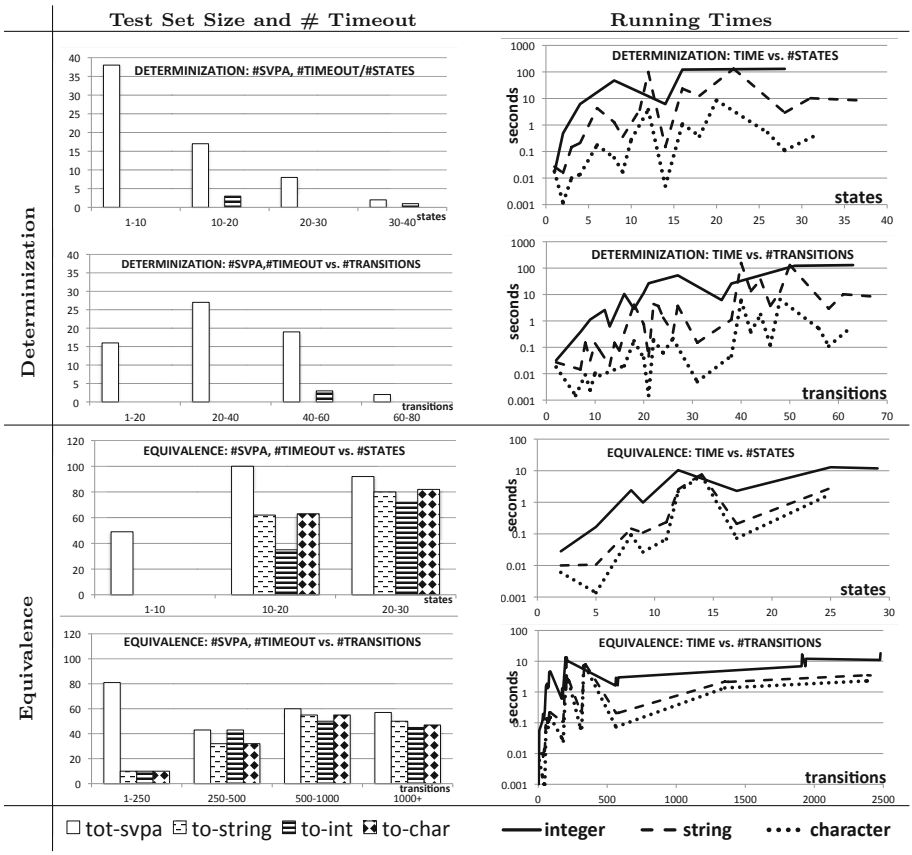


Fig. 4. Running times for equivalence and determinization

on average 6 times faster than the theory of strings and 10 times faster than the theory of integers. However, we did observe that the theory of integers timed out less often in the case of equivalence. We believe that this is due to the different choices of predicates in Φ and to the fact that Z3 uses a caching mechanism that avoids checking for satisfiability of the same predicate twice. While during the determinization such a technique is not very beneficial due to the limited number of different minterms, in the case of equivalence, especially for bigger inputs, many of the predicates are repeated making caching useful in practice.

When comparing against an existing implementation of VPAs, we observed that the benefit of using SVPA is immense: due to the large size of the alphabet (2^{16} characters), VPALib timed out for each input we considered.

6 Conclusion

We introduce Symbolic Visibly Pushdown Automata that extend VPAs with predicates over a decidable input theory, while preserving the closure prop-

erties of VPAs. We show how XML/HTML processing and program monitoring can benefit from the expressiveness and closure properties of SVPAs. We implemented SVPAs on top of different and potentially infinite input theories and observed that our implementation can handle reasonably big and complex SVPAs. Moreover, we observed that thanks to their succinctness SVPAs are able to handle large finite input alphabets, such as UTF16, that previous implementations of VPAs cannot handle.

References

1. Alur, R., Madhusudan, P.: Visibly pushdown languages. In: Proceedings of the 36th ACM Symposium on Theory of Computing, pp. 202–211 (2004)
2. Alur, R., Madhusudan, P.: Adding nesting structure to words. Journal of the ACM 56(3) (2009)
3. Bès, A.: An application of the Feferman-Vaught theorem to automata and logics for words over an infinite alphabet. CoRR, abs/0801.2498 (2008)
4. Bojanczyk, M., David, C., Muscholl, A., Schwentick, T., Segoufin, L.: Two-variable logic on data words. ACM Trans. Comput. Log. 12(4), 27 (2011)
5. Chaudhuri, S., Alur, R.: Instrumenting C programs with nested word monitors. In: Bošnački, D., Edelkamp, S. (eds.) SPIN 2007. LNCS, vol. 4595, pp. 279–283. Springer, Heidelberg (2007)
6. Cheng, E.Y., Kaminski, M.: Context-free languages over infinite alphabets. Acta Informatica 35(3), 245–267 (1998)
7. D'Antoni, L., Veanes, M.: Equivalence of extended symbolic finite transducers. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 624–639. Springer, Heidelberg (2013)
8. D'Antoni, L., Veanes, M.: Static analysis of string encoders and decoders. In: Giacobazzi, R., Berdine, J., Mastroeni, I. (eds.) VMCAI 2013. LNCS, vol. 7737, pp. 209–228. Springer, Heidelberg (2013)
9. D'Antoni, L., Veanes, M.: Minimization of Symbolic Automata. In: Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (2014)
10. de Moura, L., Bjørner, N.S.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
11. Debarbieux, D., Gauwin, O., Niehren, J., Sebastian, T., Zergaoui, M.: Early nested word automata for xPath query answering on XML streams. In: Konstantinidis, S. (ed.) CIAA 2013. LNCS, vol. 7982, pp. 292–305. Springer, Heidelberg (2013)
12. Driscoll, E., Thakur, A., Reps, T.: Opennwa: A nested-word automaton library. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 665–671. Springer, Heidelberg (2012)
13. Gauwin, O., Niehren, J.: Streamable fragments of forward xPath. In: Bouchou-Markhoff, B., Caron, P., Champarnaud, J.-M., Maurel, D. (eds.) CIAA 2011. LNCS, vol. 6807, pp. 3–15. Springer, Heidelberg (2011)
14. Kaminski, M., Tan, T.: Tree automata over infinite alphabets. In: Avron, A., Dershowitz, N., Rabinovich, A. (eds.) Pillars of Computer Science. LNCS, vol. 4800, pp. 386–423. Springer, Heidelberg (2008)
15. Kieżun, A., Ganesh, V., Guo, P.J., Hooimeijer, P., Ernst, M.D.: HAMPI: A solver for string constraints. In: Proceedings of the 2009 International Symposium on Software Testing and Analysis, ISSTA 2009, Chicago, IL, USA, July 21-23 (2009)

16. Kumar, V., Madhusudan, P., Viswanathan, M.: Visibly pushdown automata for streaming XML. In: WWW 2007: Proceedings of the 16th International Conference on World Wide Web, pp. 1053–1062. ACM, New York (2007)
17. Martens, W., Neven, F., Schwentick, T.: Complexity of decision problems for XML schemas and chain regular expressions. SIAM J. Comput. 39(4), 1486–1530 (2009)
18. Mozafari, B., Zeng, K., D'Antoni, L., Zaniolo, C.: High-Performance Complex Event Processing over Hierarchical Data. In: ACM TODS's Special Issue on Best of SIGMOD (2013)
19. Veanes, M.: Applications of symbolic finite automata. In: Konstantinidis, S. (ed.) CIAA 2013. LNCS, vol. 7982, pp. 16–23. Springer, Heidelberg (2013)
20. Veanes, M., Bjørner, N.: Symbolic automata: The toolkit. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 472–477. Springer, Heidelberg (2012)
21. Veanes, M., Hooimeijer, P., Livshits, B., Molnar, D., Bjorner, N.: Symbolic finite state transducers: Algorithms and applications. In: Proceedings of the Symposium on Principles of Programming Languages (POPL) (January 2012)
22. Zilio, S.D., Lugiez, D.: XML schema, tree logic and sheaves automata. In: Nieuwenhuis, R. (ed.) RTA 2003. LNCS, vol. 2706, pp. 246–263. Springer, Heidelberg (2003)