The power of symbolic automata and transducers

Loris D'Antoni[1] and Margus Veanes[2]

[1] University of Wisconsin, Madison
loris@cs.wisc.edu

[2] Microsoft Research
margus@microsoft.com

Abstract. Symbolic automata and transducers extend finite automata and transducers by allowing transitions to carry predicates and functions over rich alphabet theories, such as linear arithmetic. Therefore, these models extend their classic counterparts to operate over infinite alphabets, such as the set of rational numbers. Due to their expressiveness, symbolic automata and transducers have been used to verify functional programs operating over lists and trees, to prove the correctness of complex implementations of BASE64 and UTF encoders, and to expose data parallelism in computations that may otherwise seem inherently sequential. In this paper, we give an overview of what is currently known about symbolic automata and transducers as well as their variants. We discuss what makes these models different from their finite-alphabet counterparts, what kind of applications symbolic models can enable, and what challenges arise when reasoning about these formalisms. Finally, we present a list of open problems and research directions that relate to both the theory and practice of symbolic automata and transducers.

1 Introduction

This paper summarizes the recent results in the theory and applications of symbolic automata and transducers, which are models for reasoning about lists and trees over complex domains. Finite automata and transducers are used in many applications in software engineering, including software veri cation [13], text processing [7], and computational linguistics [38]. Despite their many applications, these models su er from a major drawback: in the most common forms they can only handle nite and small alphabets.

To overcome this limitation, symbolic automata and transducers allow transitions to carry predicates and functions over a speci ed alphabet theory, such as linear arithmetic, and therefore extend nite automata to operate over in nite alphabets, such as the set of rational numbers. Despite this generality, symbolic models retain many of the good properties of their nite-alphabet counterparts and have enabled new applications such as veri cation of string sanitizers [30], analysis of tree-manipulating programs [23], and program synthesis [33].

Despite this success, traditional algorithms that work over finite alphabets have been proven hard to generalize to the symbolic setting, making the design of algorithms for symbolic models challenging and theoretically interesting. In certain cases, properties that hold for finite alphabets stop holding in the symbolic setting—e.g., while it is decidable to check whether a finite state transducer is injective, the same problem is undecidable for symbolic finite transducers.

Intention and organization. The intention of this paper is to give an overview of what is currently known about symbolic automata and transducers. At the same time, we take this opportunity to present new properties that were not formally investigated in earlier papers and explain to the reader what differentiates symbolic models from their finite-alphabet counterparts. We also show what applications have been made possible thanks to the models we present.

In summary, the paper describes:

- The existing results on symbolic finite automata, their extensions (§ 2), and their applications (§ 3);
- The existing results on symbolic finite transducers, their extensions (§ 4), and their applications (§ 5); and
- A brief list of the current challenges and open problems related to symbolic automata and transducers (§ 6).

Related work. It should be noted that the concept of automata with predicates instead of concrete symbols was first mentioned in [59] and was discussed in [49] in the context of natural language processing. This paper focuses on work done following the definition of symbolic finite automata presented in [55], where predicates have to be drawn from a decidable Boolean algebra. The term symbolic automata is sometimes used to refer to automata over finite alphabets where the state space is represented using BDDs [43]. This meaning is different from the one described in this paper.

Finally, it is hard to describe all the work related to symbolic automata in one paper and the authors curate an updated list of papers on symbolic automata and transducers [3]. Many of the algorithms we discuss in this paper are implemented in the open source libraries AutomataDotNet (in C#) [1] and symbolicautomata (in Java) [4], and many of the benchmarks used in the applications cited in this paper are available in the open source collection of benchmarks AutomatArk [2].

2 Symbolic Automata

In symbolic automata, transitions carry predicates over a Boolean algebra. Formally, an *effective Boolean algebra* $\mathcal{A}$ is a tuple $(\mathfrak{D}, \Psi, [\![_]\!], \bot, \top, \vee, \wedge, \neg)$ where $\mathfrak{D}$ is a set of *domain elements*; Ψ is a set of *predicates* closed under the Boolean connectives, with $\bot, \top \in \Psi$; the component $[\![_]\!] : \Psi \to 2^{\mathfrak{D}}$ is a *denotation function* such that (i) $[\![\bot]\!] = \emptyset$, (ii) $[\![\top]\!] = \mathfrak{D}$, and (iii) for all $\varphi, \psi \in \Psi$, $[\![\varphi \vee \psi]\!] = [\![\varphi]\!] \cup [\![\psi]\!]$, $[\![\varphi \wedge \psi]\!] = [\![\varphi]\!] \cap [\![\psi]\!]$, and $[\![\neg\varphi]\!] = \mathfrak{D} \setminus [\![\varphi]\!]$. We also require that checking *satisfiability* of φ—i.e., whether $[\![\varphi]\!] \neq \emptyset$—is *decidable*.

In practice, an (effective) Boolean algebra is implemented as an API with corresponding methods implementing the Boolean operations.

Example 1 (Equality Algebra). The *equality algebra* over an arbitrary set $\mathfrak{D}$ has an atomic predicate φ_a for every $a \in \mathfrak{D}$ such that $[\![\varphi_a]\!] = \{a\}$ as well as predicates $\bot$ and $\top$. The set of predicates Ψ is the Boolean closure generated from the atomic predicates — e.g., $\varphi_a \vee \varphi_b$ and $\neg\varphi_a$ where $a, b \in \mathfrak{D}$ are predicates in Ψ.

Example 2 (SMT Algebra). Consider a fixed type τ and let Ψ be the set of all quantifier free formulas with one fixed free variable x of type τ. Intuitively, SMT_τ with is a Boolean algebra representing a restricted use of an SMT solver such as Z3 [24]. Formally, $SMT_\tau = (\mathfrak{D}, \Psi, [\![_]\!], \bot, \top, \vee, \wedge, \neg)$, where $\mathfrak{D}$ is the set of all elements of type τ, Ψ is the set of all quantifier free formulas containing a single uninterpreted constant $x : \tau$, the true predicate $\top$ is $x = x$, the false predicate $\bot$ is $x \neq x$, and the Boolean operations are the corresponding connectives in SMT formulas. The interpretation function $[\![\varphi]\!]$ is defined using the operations of satisfiability checking and model generation provided by an SMT solver. For example, we can imagine that $SMT_{\mathbb{Z}}$ is the algebra in which elements have type $\tau = \mathbb{Z}$ and predicates are in integer linear arithmetic. Examples of such predicates are $\varphi_{>0}(x) \stackrel{\text{def}}{=} x > 0$ and $\varphi_{odd}(x) \stackrel{\text{def}}{=} x\%2 = 1$.

We can now define symbolic finite automata, which are finite automata over a symbolic alphabet, where edge labels are replaced by predicates.

Definition 1. A *symbolic finite automaton* (s-FA) is a tuple $M = (\mathcal{A}, Q, q^0, F, \Delta)$ where $\mathcal{A}$ is an effective Boolean algebra, Q is a finite set of *states*, $q^0 \in Q$ is the *initial state*, $F \subseteq Q$ is the set of *final states*, and $\Delta \subseteq Q \times \Psi_{\mathcal{A}} \times Q$ is a finite set of *transitions*.

Elements of $\mathfrak{D}$ are called *characters* and finite sequences of characters are called *strings* — i.e., elements of $\mathfrak{D}^*$. A transition $\rho = (q_1, \varphi, q_2) \in \Delta$, also denoted $q_1 \xrightarrow{\varphi} q_2$, is a transition from the *source* state q_1 to the *target* state q_2, where φ is the *guard* or *predicate* of the transition. For a character $a \in \mathfrak{D}$, an *a-transition* of M, denoted $q_1 \xrightarrow{a} q_2$ is a transition $q_1 \xrightarrow{\varphi} q_2$ such that $a \in [\![\varphi]\!]$.

An s-FA M is *deterministic* if, for all transitions $(q, \varphi_1, q_1), (q, \varphi_2, q_2) \in \Delta$, if $q_1 \neq q_2$ then $[\![\varphi_1 \wedge \varphi_2]\!] = \emptyset$ — i.e., for each state q and character a there is at most one a-transition from q.

A string $w = a_1 a_2 \ldots a_k$ is *accepted at state q* iff, for $1 \leq i \leq k$, there exist transitions $q_{i-1} \xrightarrow{a_i} q_i$ such that $q_0 = q$ and $q_k \in F$. We refer to the set of strings accepted at q as the *language of M accepted at q*, denoted as $\mathcal{L}_q(M)$; the *language accepted by M* is $\mathcal{L}(M) = \mathcal{L}_{q^0}(M)$.

It is convenient to work with s-FAs that are *normalized* and have at most one transition from any state to another. For any two states p and q in Q we define $\Delta(p, q) \stackrel{\text{def}}{=} \bigvee\{\varphi \mid (p, \varphi, q) \in \Delta\}$ where $\bigvee\emptyset \stackrel{\text{def}}{=} \bot$. We can then define the normalized representation of an s-FA where for every two states p and q, we assume a single transition $p \xrightarrow{\Delta(p,q)} q$. Equivalently, in this normalized representation Δ is a function from $Q \times Q$ to Ψ with $\Delta(p, q) = \bot$ when there is no transition from p

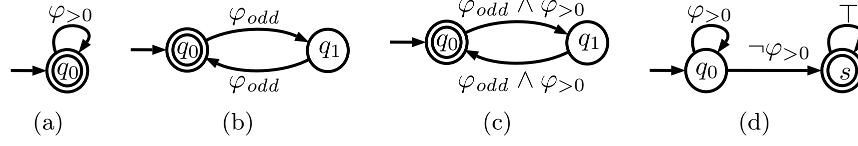

Fig. 1. Symbolic automata, a) $\mathbf{M_{pos}}$; b) $\mathbf{M_{ev/odd}}$; c) $\mathbf{M_{ev/odd}} \times \mathbf{M_{pos}}$; d) $\mathbf{M^c_{pos}}$.

to q. We also de ne $dom(p) \stackrel{\text{def}}{=} \bigvee\{\varphi \mid \exists q : (p, \varphi, q) \in \Delta\}$, to denote the set of all characters for which there exists a transition from a state p. A state p of M is *complete* if $[\![dom(p)]\!] = \mathfrak{D}_\mathcal{A}$; p is *partial* otherwise. Observe that p is partial i $\neg dom(p)$ is satis able. The s-FA M is *complete* if all states of M are complete; M is *partial* otherwise.

Example 3. Examples of s-FAs are $\mathbf{M_{pos}}$ and $\mathbf{M_{ev/odd}}$ in Figure 1. These two s-FAs have 1 and 2 states respectively, and they both operate over the Boolean algebra $SMT_\mathbb{Z}$ from Example 2. The s-FA $\mathbf{M_{pos}}$ accepts all strings consisting only of positive numbers, while the s-FA $\mathbf{M_{ev/odd}}$ accepts all strings of even length consisting only of odd numbers. For example, $\mathbf{M_{ev/odd}}$ accepts the string [2, 4, 6, 2] and rejects strings [2, 4, 6] and [51, 26]. The product automaton of $\mathbf{M_{pos}}$ and $\mathbf{M_{ev/odd}}$, $\mathbf{M_{ev/odd}} \times \mathbf{M_{pos}}$, accepts the language $\mathcal{L}(\mathbf{M_{pos}}) \cap \mathcal{L}(\mathbf{M_{ev/odd}})$. Both s-FAs are partial| e.g., neither of them has transitions for character -1.

2.1 Interesting Properties

In this section, we illustrate some basic properties of s-FAs and show how these models di er from nite automata. A key characteristic of all s-FAs algorithms is that there is no explicit use of characters because $\mathfrak{D}$ may be in nite and the interface to the Boolean algebra does not directly support use of individual characters.

Similarly to what happens for nite automata, nondeterminism does not add expressiveness for s-FAs.

Theorem 1 (Determinizability [55]). *Given an s-FA M one can effectively construct a* deterministic *s-FA M_{det} such that $\mathcal{L}(M) = \mathcal{L}(M_{\text{det}})$.*

The determinization algorithm is similar to the subset construction for automata over nite alphabets, but also requires combining predicates appearing in di erent transitions. If M contains k inequivalent predicates and n states, then the number of distinct predicates in M_{det} is at most 2^k and the number of states is at most 2^n. In other words, in addition to the classic state space explosion risk there is also a predicate space explosion risk.

Since s-FAs can be determinized, we can show that s-FAs are closed under Boolean operations using variations of classic automata constructions.

Theorem 2 (Boolean Operations [55]). *Given s-FAs M_1 and M_2 one can effectively construct s-FAs M_1^c and $M_1 \times M_2$ such that $\mathcal{L}(M_1^c) = \mathfrak{D}_\mathcal{A}^* \setminus \mathcal{L}(M_1)$ and $\mathcal{L}(M_1 \times M_2) = \mathcal{L}(M_1) \cap \mathcal{L}(M_2)$.*

The intersection of two s-FAs is computed using a variation of the classic product construction in which transitions are \synchronized" using conjunction. For example, the intersection of $\mathbf{M_{pos}}$ and $\mathbf{M_{ev/odd}}$ from Example 3 is shown in Figure 1(c).

To complement a deterministic partial s-FA M, M is rst *completed* by adding a new non- nal state s with loop $s \xrightarrow{\top} s$ and for each partial state p a transition $p \xrightarrow{\neg dom(p)} s$. Then the nal states and the non- nal states are swapped in M^c. Following this procedure, the complement of $\mathbf{M_{pos}}$ from Example 3 is shown in Figure 1(d).

Next, s-FAs enjoy the same decidability properties of nite automata.

Theorem 3 (Decidability [55]). *Given s-FAs M_1 and M_2 it is decidable to check if M_1 is empty—i.e., whether $\mathcal{L}(M_1) = \emptyset$—and if M_1 and M_2 are language-equivalent—i.e. whether $\mathcal{L}(M_1) = \mathcal{L}(M_2)$.*

Checking emptiness requires checking what transitions are satis able and, once unsatis able transitions are removed, any path reaching a nal state from an initial state represents at least one accepting string. Equivalence can be reduce to emptiness using closure under Boolean operations.

Algorithms have also been proposed for minimizing *deterministic* s-FAs [18], for checking language inclusion [34], for computing forward bisimulations of s-FAs [21], and for learning s-FAs from membership and equivalence queries [25].

Alphabet equivalence classes. Classic automata can only describe sequences over nite alphabets. Despite this limitation, there is a way to convert every s-FA M into a nite automaton that, in some sense, preserves the set of all strings accepted by the s-FA. Although the set S of all predicates appearing in a given s-FA (or nite collection of s-FAs over the same alphabet algebra) operate over an in - nite domain, the set of maximal satis able Boolean combinations $\mathit{Minterms}(S)$ | also called minterms | of such predicates induces a nite set of equivalence classes. In order to perform operations over one or more s-FAs M by using classical automata algorithms, one can consider $\Sigma = \mathit{Minterms}(\mathit{Predicates}(M))$ as the induced nite alphabet and replace each original transition $p \xrightarrow{\varphi} q$ by the transitions $\{p \xrightarrow{c} q \mid c \in \Sigma, \mathbf{SAT}(c \wedge \varphi)\}$ and consequently treat the automata as classic nite automata over the alphabet Σ.

Example 4. Consider the two s-FAs $\mathbf{M_{pos}}$ and $\mathbf{M_{ev/odd}}$ in Figure 1. Then

$$S = \mathit{Predicates}(\mathbf{M_{pos}}, \mathbf{M_{ev/odd}}) = \{\varphi_{>0}, \varphi_{odd}\}$$

and

$$\Sigma = \mathit{Minterms}(S) = \{\underbrace{\varphi_{odd} \wedge \varphi_{>0}}_{a}, \underbrace{\neg\varphi_{odd} \wedge \varphi_{>0}}_{b}, \underbrace{\varphi_{odd} \wedge \neg\varphi_{>0}}_{c}, \underbrace{\neg\varphi_{odd} \wedge \neg\varphi_{>0}}_{d}\}$$

Then, as a DFA over the nite alphabet Σ, $\mathbf{M_{pos}}$ has the transitions $\{(q_0, a, q_0), (q_0, b, q_0)\}$ and $\mathbf{M_{ev/odd}}$ has the transitions $\{(q_0, a, q_1), (q_0, c, q_1),$

$(q_1, a, q_0), (q_1, c, q_0)\}$. In the product $\mathbf{M_{pos}} \times \mathbf{M_{ev/odd}}$ only the a-transitions remain.

Intuitively, using only the predicates in Σ there is no way to, for example, distinguish the number 1 from the number 3| i.e., given any string s, if one replaces any element 1 in s with the element 3, the new sequence s' is accepted by the s-FA i s is also accepted by the s-FA. ⊠

Using this argument, every s-FA M can be compiled into a symbolically equivalent nite automaton over any alphabet $\mathit{Minterms}(S)$ where $\mathit{Predicates}(M)$ forms a subset of S and S is a nite subset of Ψ. This idea, also referred to as predicate abstraction, is often used in program veri cation [26].

In general, computing the set $\mathit{Minterms}(M) \stackrel{\text{def}}{=} \mathit{Minterms}(\mathit{Predicates}(M))$ is an expensive procedure that generate exponentially many predicates. The following theorem exactly characterizes the size of the set $\mathit{Minterms}(M)$.

Theorem 4 (Number of minterms). *Let M be a complete and normalized s-FA with n states. Then $|\mathit{Minterms}(M)| \leq 2^{(n^2)}$. If M is deterministic then $|\mathit{Minterms}(M)| \leq 2^{n \log_2 n}$.*

Proof. Let $S = \mathit{Predicates}(M)$. Since M is normalized we have $|\Delta| \leq n^2$ and so $|S| \leq n^2$, and since $|\mathit{Minterms}(S)| \leq 2^{|S|}$ the rst claim follows. Assume now that M is deterministic. Then every source state p_i of M, for $i < n$, de nes a partition P_i of $\mathfrak{D}$ such that $|P_i| \leq n$ because M is normalized, where each part of P_i is de ned by the guard of a transition from p_i. Given two partitions P_i and P_j of $\mathfrak{D}$ let $P_i \sqcap P_j$ denote the coarsest partition of $\mathfrak{D}$ that re nes both P_i and P_j. Then $\{[\![\mu]\!] \mid \mu \in \mathit{Minterms}(S)\} = \prod_{i<n} P_i$. Since, for every i, $|P_i| \leq m$ implies $|\prod_{i<n} P_i| \leq m^n$, the following holds: $|\mathit{Minterms}(S)| \leq n^n = 2^{n \log_2 n}$. ⊠

2.2 Parametric complexities

In the previous paragraphs we did not discuss the complexities of the presented algorithms. Since s-FAs are parametric in an underlying alphabet theory, the complexities of the algorithms must in some way depend on the complexities of performing certain operations in the alphabet theory.

For example, checking emptiness of an s-FA requires checking satis ability of all predicates in the s-FA and the complexity depends on \how costly" it is to check satis ability of such predicates. Another issue arises from algorithms that generate new predicates that did not belong to the original s-FAs. In particular, repeated predicate conjunctions, unions, and complementations will cause predicates to grow in size and might therefore result in satis ability queries with higher costs. This peculiar aspect of s-FAs opens a new set of complexity questions that have not been studied in classic automata theory.

Let's consider again the problem of checking emptiness of an s-FA. In classic automata, this problem has complexity $\mathcal{O}(kn)$ where k is the size of the alphabet and n is the number of states in the automaton. For an s-FA M, if we assume that the largest predicate in M has size ℓ and $f(x)$ is the cost of checking satis ability

of predicates of size x in the underlying alphabet theory, then checking emptiness has complexity $\mathcal{O}(m \cdot f(\ell))$, where m is the number of transitions in the s-FA M. Observe also that for s-FAs it is reasonable to work with normalized representations which implies that m is at most n^2 and m is independent of the alphabet size and the total size of M is $\mathcal{O}(m\ell)$.

For certain problems, the complexities can get more complicated and different algorithms will have different incomparable complexities. For example, consider the problem of minimizing a deterministic s-FA. For classic automata, there are two algorithms for solving this problem: (i) Moore's algorithm, which has complexity $\mathcal{O}(kn^2)$; (ii) Hopcroft's algorithm, which has complexity $\mathcal{O}(kn \log n)$. It is therefore clear that Hopcroft's algorithm has better asymptotic complexity than Moore's algorithm. In the case of s-FAs, the situation is more complicated. For an s-FAs M with n states and m transitions, if we assume that the largest predicate in M has size ℓ and $f(x)$ is the cost of checking satisfiability of predicates of size x in the underlying alphabet theory, the symbolic adaptation of Moore's algorithm has complexity $\mathcal{O}(mn \cdot f(\ell))$, while the symbolic adaptation of Hopcroft's algorithm has complexity $\mathcal{O}(m \log n \cdot f(n\ell))$. For s-FAs, the two algorithms have somewhat orthogonal theoretical complexities: Hopcroft's algorithm saves a logarithmic factor in terms of state complexity, but this saving comes at the cost of running more expensive satisfiability queries on predicates of size $n\ell$. Given the recent advances in satisfiability procedures, the second algorithm behaves better in practice.

2.3 Variants

Symbolic automata have been extended in various ways. Symbolic alternating automata (s-AFA) together with a practical equivalence algorithm are presented in [17]. s-AFAs are equivalent in expressiveness to s-FAs, but achieve succinctness by extending s-FAs with alternation [14] and, despite the high theoretical complexity, this model can at times be more practical than s-FAs. A very common extension of s-FAs is to allow multiple initial states, in particular when dealing with nondeterministic s-FAs [21].

Symbolic tree automata (s-TA) operate over trees instead of strings. s-FAs are a special case of s-TAs in which all nodes in the tree have one child or are leaves. s-TAs have the same closure and decidability properties as s-FAs [52]. Moreover, the minimization algorithms for s-FAs has been extended to s-TAs [20].

Symbolic visibly pushdown automata (s-VPA) operate over nested words, which are used to model data with both linear and hierarchical structure such—e.g., XML documents and recursive program traces. s-VPAs can be determinized and have the same closure and decidability properties of s-FAs [16].

All the previous extensions show cases in which adapting classic models to the symbolic setting does not affect closure and decidability properties. This is not the case for Symbolic Extended Finite Automata (s-EFA) [19]. s-EFAs are symbolic automata in which each transition can read more than a single character. In this model, predicates apply to finite tuples of elements up to a fixed length, but the semantics flattens the tuples.

Formally, the domain $\mathfrak{D}$ of $\mathcal{A}$ is assumed to contain tuples to enable the use of multiple variables in this setting. There are predicates $IsTup_k$ for checking if an element is a k-tuple for $k \geq 1$ and there are projection terms x_i or *variables* such that for a k-tuple $a = (a_1, \ldots, a_k)$, and $1 \leq i \leq k$, $[\![x_i]\!](a) = a_i$. For example, using equality or disequality, one can relate elements of tuples. A predicate over k-tuples is called k*-ary*.

Example 5. A predicate $IsTup_2 \wedge x_1 \neq x_2 \wedge \varphi$ is satis able i there exists $a \in \mathfrak{D}$ such that a is a pair (a_1, a_2) and $a_1 \neq a_2$, and $[\![\varphi]\!](a_1, a_2)$ holds. ⊠

Thus, if $[(a, b, c), (d), (e, f)] \in \mathcal{L}(M)$ where M is a considered as an s-FA then $[a, b, c, d, e, f] \in \mathcal{L}^{\mathrm{e}}(M)$ when M is considered as an s-EFA. Each individual transition guard must uniquely de ne the length k of the tuple that determines its arity. For example, the following transition reads two adjacent symbols x_1 and x_2 and checks whether the two symbols are equal:

$$p \xrightarrow[2]{x_1 = x_2} q.$$

While for automata over nite alphabet adding the the ability to consume multiple characters in a single transition does not increase expressiveness, s-EFAs are strictly more expressive than s-FAs. Moreover, s-EFAs lack many of the desirable properties s-FAs enjoy: s-EFAs are not closed under Boolean operations, nondeterministic s-EFAs are strictly more expressive than their deterministic counterpart, it is undecidable to check whether two s-EFAs are equivalent, or even to check whether their intersection is empty. An important subclass of s-EFAs, called *Cartesian* s-EFAs [19], has the same expressive power as s-FAs and allows transitions with lookahead but the guards must be predicates whose atoms only mention one variable at a time. Thus the atom $x_1 = x_2$ would not be allowed. A related problem, called *monadic decomposition* [54] arises if we want to decide if a predicate can be e ectively transformed into an equivalent Cartesian form.

3 Symbolic Automata in Practice

The development of the theory of symbolic automata is motivated by concrete practical problems. Here we discuss some of them.

3.1 Analysis of Regular Expressions

The connection between automata and regular expressions has been studied for more than 50 years. However, real-world regular expressions are much more complex than the simple model described in a typical theory of computation course. In particular, in practical regular expressions the size of the alphabet is 2^{16} due to the widely adopted UTF16 standard of Unicode characters. The inability of classic automata to e ciently handle large alphabets is what started the study of symbolic automata.

Using s-FAs, the alphabet of Unicode characters can be modeled as a theory of bit-vectors where predicates are represented as Binary Decision Diagrams (BDDs) over such bit-vectors [31] or using bit-vector arithmetic in Z3 [55]. These representations turned out to be a viable way to model practical regular expressions and led to advanced analysis in the context of parametrized unit testing in the tool PEX [48], automatic SQL query exploration in QEX [56], and random password generation [18].

In applications that perform many Boolean operations on the regular expressions| e.g., in text processing and analysis of string-manipulating programs [7, 57]| s-FAs may generate very large number of states despite their succinct alphabet representations. The extension of s-FAs with alternation, s-AFAs, can succinctly represent Boolean combinations of s-FAs and it was shown to be an e ective model for checking equivalence of complex combinations of regular expressions.

3.2 Other applications

Thanks to the symbolic treatment of the alphabet, symbolic automata are an executable model and can be used to generate e cient code. This idea has been used to achieve speed-ups in regular expression processing [45] and XML processing [16].

Recently, s-VPAs have been used in the context of static analysis of program failures to succinctly model properties of control- ow graphs [40]. This model is particularly helpful in modelling properties of inter-procedural programs with many di erent functions. In this setting, a classic automaton will need to have number of states and transitions proportional to the number of functions| i.e., when a function f is invoked, push a state remembering the name f on a stack and pop it at the function return. On the other hand, symbolic visibly pushdown automata can model this call/return interaction symbolically with a single transition that simply requires the function that is currently returning to have the same name as the last called function.

4 Symbolic Transducers

In this section, we present symbolic nite transducers, which are symbolic automata that can produce outputs. The presentation here follows the original de nition from [57] but omits type annotations. In addition to predicates we use expressions for representing anonymous functions that we call *function terms*. Let $\mathcal{A}$ be a Boolean algebra as de ned in Section 2. The set of function terms is denoted by Λ and a term $f \in \Lambda$ denotes a function $[\![f]\!]$ over $\mathfrak{D}$, such that if $f, g \in \Lambda$ then $g(f) \in \Lambda$ and it is such that for every $a \in \mathfrak{D}$:

$$[\![g(f)]\!](a) = [\![g]\!]([\![f]\!](a)).$$

Similarly, if $\varphi \in \Psi$ and $f \in \Lambda$ then $\varphi(f) \in \Psi$ such that, for $a \in \mathfrak{D}$:

$$a \in [\![\varphi(f)]\!] \Leftrightarrow [\![f]\!](a) \in [\![\varphi]\!].$$

Moreover, $f = g$ is an *equality* predicate in $\Psi_\mathcal{A}$ such that, for $a \in \mathfrak{D}$:

$$a \in [\![f = g]\!] \Leftrightarrow [\![f]\!](a) = [\![g]\!](a).$$

Observe that $f = g$ *does not* mean $[\![f]\!] = [\![g]\!]$. We write $f \neq g$ for $\neg f = g$. Thus, $f \neq g$ is satisfiable iff $[\![f]\!] \neq [\![g]\!]$.

Furthermore, there is an *identity* (function) term $x \in \Lambda$ such that, for all $a \in \mathfrak{D}$, $[\![x]\!](a) = a$, and for all $c \in \mathfrak{D}$ there is a *constant* term $c \in \Lambda$ such that for all $a \in \mathfrak{D}$, $[\![c]\!](a) = c$.

Example 6. Predicate $\varphi \wedge f \neq g$ is satisfiable iff there exists $a \in [\![\varphi]\!]$ such that $[\![f]\!](a) \neq [\![g]\!](a)$ — i.e., when f and g are not equivalent wrt φ. Predicate $f \neq c$ for a given $c \in \mathfrak{D}$ is satisfiable iff f does not denote the constant function c. ⊠

Terms are typically typed but we omit type annotations here. We call such an extended (effective) Boolean algebra with the additional components an (*effective*) *label algebra*.

Definition 2. A *Symbolic Finite Transducer (s-FT)* T is a tuple $(\mathcal{A}, Q, q^0, \Delta, F)$ where: $\mathcal{A}$ is an effective label algebra; Q is a finite set of *states*; $q^0 \in Q$ is the *initial state*; Δ is a finite subset of $Q \times \Psi \times \Lambda^* \times Q$ called *transitions*; $F \subseteq Q$ is the set of final states.

In a transition (p, φ, f, q), also denoted $p \xrightarrow{\varphi/\bar{f}} q$, f is called the *output*. Observe that an s-FT in which all the transitions output the empty list corresponds to an s-FA. We also call the s-FA that is obtained from an s-FT T by removing the output component its *domain automaton*, $DOM(T)$.

Example 7. Let $\mathcal{A}$ correspond to integer linear arithmetic. So Λ contains terms such as $x\%2$ (x modulo 2), and Ψ contains atomic predicates such as $x > 0$. Here x has type $\mathbb{Z}$. The following are two examples of s-FTs:

$$T_1 = (\mathcal{A}, \{p\}, p, \{p \xrightarrow{x>0/[x,x]} p\}, \{p\}),$$

$$T_2 = (\mathcal{A}, \{q\}, q, \{q \xrightarrow{x\%2\neq 0/[x]} q, q \xrightarrow{x\%2=0/[]} q\}, \{q\}).$$

Here, T_1 accepts only positive numbers and duplicates them and T_2 deletes all the even numbers. For example, on input [1, 2, 3], the s-FT T_1 outputs [1, 1, 2, 2, 3, 3], while the s-FT T_2 outputs [1, 3]. ⊠

We now define the semantics of s-FTs. In the remainder of the section, let $T = (\mathcal{A}, Q, q^0, \Delta, F)$ be a fixed s-FT. For each transition r in Δ we define the set $[\![r]\!]$ of corresponding concrete transitions as follows.

$$[\![p \xrightarrow{\varphi/[f_1,\ldots,f_k]} q]\!] \stackrel{\text{def}}{=} \{(p, a) \mapsto ([[\![f_1]\!](a), \ldots, [\![f_k]\!](a)], q) \mid a \in [\![\varphi]\!]\}$$

Intuitively, a transition $p \xrightarrow{\varphi/\bar{f}} q$ reads one input symbol a in state p that satisfies the guard φ and produces a sequence of output symbols by applying the output

functions in f to a and enters state q. In the following, let $[\![\varDelta]\!] \stackrel{\text{def}}{=} \bigcup_{r\in\varDelta}[\![r]\!]$ and let $s_1 \cdot s_2$ denote the concatenation of two sequences s_1 and s_2. We let $\mathfrak{D}^*$ denote a *disjoint* universe from $\mathfrak{D}$ of sequences of elements over $\mathfrak{D}$, to avoid the possible ambiguity as far as concatenation is concerned.

Definition 3. For $u = [a_1, a_2, \ldots, a_n], v \in \mathfrak{D}^*, q \in Q, q' \in Q$, de ne $q \xrightarrow{u/v}\!\!\!\!\rightarrow_T q'$ i either $u = v = []$ and $q = q'$, or there is $n \geq 1$ and $\{(p_{i-1}, a_i) \mapsto (v_i, p_i)\}_{i=1}^n \subseteq [\![\varDelta]\!]$ such that $v = v_1 \cdot v_2 \cdots v_n$, $q = p_0$, and $q' = p_n$. The *transduction of* T is the relation $\mathscr{T}_T \subseteq \mathfrak{D}^* \times \mathfrak{D}^*$ such that $\mathscr{T}_T(u, v) \Leftrightarrow \exists q \in F : q^0 \xrightarrow{u/v}\!\!\!\!\rightarrow q$. Let $\mathscr{T}_T(u) \stackrel{\text{def}}{=} \{v \mid \mathscr{T}_T(u, v)\}$. Finally, he domain of T is de ned as $\mathbf{dom}(T) \stackrel{\text{def}}{=} \{u \in \mathfrak{D}^* \mid \exists v : \mathscr{T}_T(u, v)\}$, and the range of T is de ned as $\mathbf{ran}(T) \stackrel{\text{def}}{=} \{v \in \mathfrak{D}^* \mid \exists u : \mathscr{T}_T(u, v)\}$.

The s-FT T is *deterministic* when $[\![\varDelta]\!]$ is a partial function from $Q \times \mathfrak{D}$ to $\mathfrak{D}^* \times Q$. The s-FT T is *single-valued* or *functional* if, for all u, $|\mathscr{T}_T(u)| \leq 1$| i.e., $\mathscr{T}_T$ represents a partial function over $\mathfrak{D}^*$. Observe that if T is deterministic then T is also functional. Both the s-FTs in Example 7 are deterministic.

4.1 Interesting Properties

In this section, we illustrate some of the basic properties of s-FTs and show what aspects di erentiate these models from nite transducers [38], their nite-alphabet counterpart. First, while both the domain and the range of a nite state transduction are de nable using a nite automaton, this is not the case for s-FTs. By a *regular language* here we mean a language accepted by an s-FA.

An s-FT T *admits quantifier elimination* if for every transition $(p, \varphi, [f_i]_{i=1}^k, q)$ in T where $k \geq 1$ one can e ectively compute a predicate $\psi \in \Psi$ such that the following is true: for all $b \in \mathfrak{D}$, we have $b \in [\![\psi]\!]$ i b is a k-tuple $(b_i)_{i=1}^k$ such that there exists $a \in [\![\varphi]\!]$ such that $b_i = [\![f_i]\!](a)$ for $1 \leq i \leq k$. In other words, computation of ψ corresponds to eliminating the quanti er $\exists y$ from $\exists y : \varphi(y) \wedge \bigwedge_{i=1}^k x_i = f_i(y)$. Note that the predicate ψ is a k-ary predicate.

Theorem 5 (Domain and Range Languages). *Given an s-FT T, one can compute an s-FA $DOM(T)$ such that $\mathcal{L}(DOM(T)) = \mathbf{dom}(T)$ and, provided that T admits quantifier elimination, there is an s-EFA $RAN(T)$ such that $\mathcal{L}^{\mathrm{e}}(RAN(T)) = \mathbf{ran}(T)$.*

In general, the range of an s-FT is not regular.

Example 8. Take an s-FT T with a single transition $q \xrightarrow{\varphi_{odd}(x)/[x,x]} q$ that duplicates its input if the input is odd. Then $\mathbf{ran}(T)$ is not regular, but it can be accepted by the s-EFA with one transition $q \xrightarrow[2]{x_1=x_2} q$. ⊠

s-FTs are closed under sequential composition. This is a property that enables several interesting program analyses [30] and optimizations.

Theorem 6 (Closure under Composition [57]). *Given two s-FTs T_1 and T_2, one can compute an s-FT $T_2(T_1)$ such that for $u, v \in \mathfrak{D}^*$,*

$$\mathscr{T}_{T_2(T_1)}(u, v) \Leftrightarrow \exists w : \mathscr{T}_{T_1}(u, w) \wedge \mathscr{T}_{T_2}(w, v).$$

We illustrate the role of the *substitution* operator $\cdot(\cdot)$ in a label algebra in the context of computing $T_2(T_1)$. Consider the transition $p \xrightarrow{\varphi/[f_1,f_2]} p'$ in T_1 and the transitions $q \xrightarrow{\psi/[g]} q' \xrightarrow{\gamma/[h]} q''$ in T_2. The set of states $Q_{T_2(T_1)}$ of the composed transducer is a reachable subset of $Q_1 \times Q_2$. The initial state of $T_2(T_1)$ is $(q^0_{T_1}, q^0_{T_2})$. When a state (p, q) is explored then the transition

$$(p, q) \xrightarrow{\varphi \wedge \psi(f_1) \wedge \gamma(f_2)/[g(f_1), h(f_2)]} (p', q'')$$

is constructed from the above transitions where the substitution operator is applied to construct the combined guard and output functions. The composed transition is omitted if $\varphi \wedge \psi(f_1) \wedge \gamma(f_2)$ is unsatis able.

Example 9. Recall T_1 and T_2 from Example 7. Consider $T = T_2(T_1)$. Then $Q_T = \{(p, q)\}$. There are four composed candidates for the transitions in Δ_T but only the following two have satis able guards:

$$(p, q) \xrightarrow{x>0 \wedge x\%2 \neq 0 \wedge x\%2 \neq 0/[x,x]} (p, q), \quad (p, q) \xrightarrow{x>0 \wedge x\%2=0 \wedge x\%2=0/[]} (p, q)$$

Therefore T, given a list of positive numbers, duplicates all odd numbers and deletes the even ones. For example, on input [1, 2, 3], T outputs [1, 1, 3, 3]. ⊠

The following result follows from the closure properties of s-FAs and the closure under composition of s-FTs.

Corollary 1 (Type-checking). *Given an s-FTs T and s-FAs M_I and M_O, the following problem is decidable: check if for all $v \in \mathcal{L}(M_I)$: $\mathscr{T}_T(v) \subseteq \mathcal{L}(M_O)$.*

For example, using the type-checking algorithm one can prove that, for every input list, the transducer T from Example 9 always outputs a list of odd numbers of even length.

Checking whether two s-FTs are equivalent is in general undecidable (already over nite alphabets [29]). However, the problem becomes decidable when the two s-FTs are functional (single-valued), which is itself a decidable property to check.

Theorem 7 (Decidable functionality [57]). *Given an s-FTs T it is decidable to check whether T is functional.*

Theorem 8 (Decidable functional equivalence [57]). *Given two functional s-FTs T_1 and T_2 it is decidable to check whether $\mathscr{T}_{T_1} = \mathscr{T}_{T_2}$.*

Both theorems use a more general decision problem that decides for two s-FTs T_1 and T_2, if for all $u, v, w \in \mathfrak{D}^*$ it is true that if $\mathscr{T}_{T_1}(u, v)$ and $\mathscr{T}_{T_2}(u, w)$ then $v = w$. The algorithm of this decision problem [57, Figure 3] uses the disequality operator $\neq$ and, in particular, the predicates shown in Example 6.

We conclude this section with an interesting property that is decidable for classic finite state transducers [27] but undecidable for s-FTs. We say that an s-FT T is *injective* if for all $u, v \in \mathfrak{D}^*$ we have $\mathscr{T}_T(u) \cap \mathscr{T}_T(v) = \emptyset$.

Theorem 9 (Undecidable injectivity [33]). *Given a deterministic s-FT T, it is undecidable to check whether T is injective.*

The proof of undecidability presented in [33, Theorem 4.8] is given for s-EFTs and is based on showing that it is undecidable to check whether there exist two different accepting paths for the same string in the s-EFA $RAN(T)$. It is easy to show that the theorem also holds for s-FTs since every s-EFA in this theory can be produced as the range language of some s-FT.

4.2 Variants

Symbolic finite transducers have been extended in various ways. The basic extension of s-FTs is to consider *finalizers* | i.e., specific transitions that are used to output final sequences upon end of input. Finite state transducers with finalizers are called *subsequential* [46, 6]. Finalizers enable certain scenarios not possible without sacrificing determinism. Consider for example a decoder that decodes a string by replacing all patterns `"&"` by the character `"&"`. If the input string ends with for example `"&"` the decoder will need to output `"&"` instead of `"&"` upon reaching the end of the input and finding out that `";"` is missing. Similarly, for capturing *minimality*, s-FTs may also be extended with *initial* outputs [44]. For many purposes it is enough to imagine that $\mathfrak{D}$ is extended with two new symbols that are used exclusively to detect start and end of an input sequence. In a typed universe this approach is cumbersome and complicates the notion of composition by requiring bookkeeping and special treatment of the extra symbols which have to be taken outside the type domain.

Similarly to how s-EFAs extend s-FAs, Symbolic Extended Finite Transducers (s-EFT) are symbolic transducers in which each transition can read more than a single character. Essentially, the definition of $\mathscr{T}_T$ changes to $\mathscr{T}_T^{\mathrm{e}}$, similar to the change from $\mathcal{L}(M)$ to $\mathcal{L}^{\mathrm{e}}(M)$, where the input is flattened. s-EFA already lack many desirable properties and s-EFTs further add to this list. s-EFTs are not closed under composition and equivalence is undecidable even for deterministic s-EFTs. However, equivalence becomes decidable when for every transition that reads n characters using a predicate $\varphi(x_1, \ldots, x_n)$, one can replace the predicate with an equivalent disjunction of predicates of the form $\varphi_1(x_1) \wedge \ldots \wedge \varphi_n(x_n)$ [19].

A further extension of s-FTs, called s-RTs, incorporates the notion of bounded *look-back* and *roll-back* in form of roll-back-transitions, not present in any other transducer formalisms, to accommodate default or exceptional behavior [50]. The key application is to simplify handling of default transitions such as the

followings: if none of those patterns matches then read and output the next input character \as is". Having to hand-code state machines for such cases gets complicated and error prone very quickly| e.g., see [57, Figure 7].

s-FTs have also been extended with *registers* [57] and are called *symbolic transducers*. The key motivation is to support loop-carried data state, such as the maximal number seen so far. This model is closed under composition, but most decision problems for it are undecidable, even emptiness.

A further extension of symbolic transducers uses *branching transitions*, which are transitions with multiple target states in form of if-then-else structures [45]. The purpose is to better facilitate code generation by maintaining code structure, sharing, and predicate evaluation order for deterministic transducers. For example, instead of two separate transitions $p \xrightarrow{\varphi/\bar{f}} q$ and $p \xrightarrow{\neg\varphi/\bar{g}} r$, there is a single branching transition $p \mapsto \mathit{if}\ \varphi\ \mathit{then}\ (f, q)\ \mathit{else}\ (g, r)$. If there is one branching transition per state then determinism is built-in. One can of course apply the same idea to s-FAs.

Symbolic *tree* transducers (s-TT) operate over trees instead of strings. s-FTs are a special case of s-TTs in which all nodes in the tree have one child or are leaves. s-TTs are only closed under composition when certain assumptions hold and their properties are studied in [28]. Equivalence of a restricted class of s-TTs is shown decidable in [51]. s-TTs with regular look-ahead are studied in [23].

5 Symbolic Transducers in Practice

Here we provide a high-level overview of the main applications involving symbolic nite transducers and their variants.

5.1 Analysis of string encoders and sanitizers

The original motivation for s-FTs came from analysis of string *sanitizers* [30]. String sanitizers are particular string to string functions over Unicode designed to encode special characters in text that may otherwise trigger malicious code execution in certain sensitive contexts, primarily in HTML pages. Thus, sanitizers provide a rst line of defence against cross site scripting (XSS) attacks. When sanitizers can be represented as s-FTs, one can, for example, decide if two sanitizers A and B commute| i.e., if $\mathscr{T}_{A(B)} = \mathscr{T}_{B(A)}$| if a sanitizer A is idempotent| i.e., if $\mathscr{T}_{A(A)} = \mathscr{T}_{A}$| or if A cannot be compromised with an input attack vector| i.e., if $\mathbf{ran}(A) \subseteq \mathit{SafeSet}$. Checking such properties can help to ensure the correct usage of sanitizers.

One drawback of s-FTs is that they consider one input element at a time. While this is often su cient for individual character-based transformations appearing in common sanitizers, in more complex transformations, such as BASE64 encoders and decoders, it is often necessary to be able to look at a group of characters at once in order to decode them. For example, a BASE64 encoder reads three characters at a time and outputs complex combinations and bit-level transformations of the bits appearing in the characters. This is the original motivation

behind s-EFTs, which are studied in [19]. Using s-EFTs one can prove that efficient implementations of BASE64 or UTF encoders and decoders correctly invert each other. Recently, s-EFTs have been used to automatically compute inverses of encoders that are correct by construction [33].

Variants of algorithms for learning symbolic automata and transducers have been used to automatically extract models of PHP input filters [12] and string sanitizers [8]. In these applications, symbolic automata and transducers have enabled modelling of programs that were beyond the reach of existing automata-learning algorithms.

Symbolic transducers have also been used to perform static analysis of functional programs that operate over lists and trees [23]. In particular, symbolic tree transducers were used to verify HTML sanitizers, to check interference of augmented reality applications submitted to an app store, and to perform deforestation, a technique to speed-up function composition, in functional language compilation.

5.2 Code generation and parallelization

Symbolic transducers can be used to expose data parallelism in computations that may otherwise seem inherently sequential. This idea builds on the property that the state transition function of a DFA can be viewed as a particular kind of matrix multiplication operation which is associative and therefore lends itself to parallelization [39]. This property can be lifted to the symbolic setting and applied to many common string transformations expressed as symbolic transducers [58].

Using closure under composition, complex combinations of symbolic transducers can be composed in a manner that supports efficient code generation. The main context where this has been evaluated is in log/data processing pipelines that require loop-carried state for data processing [45]. In this context the symbolic transducers have registers and use *branching rules* that are rules with multiple target states in form of if-then-else structures. The main purpose of the branching rules is to support serial code generation.

Symbolic automata and transducers also provide the backbone of DReX, a declarative language for efficiently executing regular string transformations in a single left-to-right pass over the input [7]. DReX has also been extended to stream numerical data computations using a "numerical" extension of symbolic transducers [36].

6 Open Problems and Future Directions

We conclude this paper with a list of open theoretical questions that are unique to symbolic automata and transducers, as well as a summary of what unexplored applications could benefit from these models.

6.1 Adapting efficient algorithms for finite alphabets

Several algorithms for classic nite automata are based on e cient data structures that directly leverage the fact that the alphabet is nite. For example, Hopcroft's algorithm for automata minimization, at each step, iterates over the alphabet to nd potential ways to split state partitions [32]. It turns out that this iteration can be avoided in symbolic automata using satis ability checks on certain carefully crafted predicates [18].

Paige-Tarjan's algorithm for computing forward bisimulations of nondeterministic nite automata is similar to Hopcroft's algorithm for DFA minimization [5, 41]. The e cient implementation of Paige-Tarjan's algorithm presented in [5] keeps, for every symbol a in the alphabet, for every state q in the automaton, and for every state partition P, a count of how many transitions from q on symbol a reach the partition P. Using this data-structure, the algorithm can compute the partition of forward-bisimilar states in time $\mathcal{O}(km \log n)$. Unlike Hopcroft's algorithm, this algorithm is hard to adapt to the symbolic setting. In fact, the current adaptation has complexity $\mathcal{O}(2^m \log n + 2^m f(n\ell))$ [21]. In contrast, the simpler $\mathcal{O}(km^2)$ algorithm for forward bisimulations can be easily turned into a symbolic $\mathcal{O}(m^2 f(\ell))$ algorithm [21]. This example shows how it can be hard to convert the most e cient algorithms for automata over nite alphabets to the symbolic setting. In fact, it remains open whether an e cient symbolic adaptation of Paige-Tarjan's algorithm exists.

Another example of this complexity of adaptation is the algorithm for checking equivalence of two nondeterministic unambiguous nite automata [47]. This algorithm checks equivalence of two automata in polynomial time by \counting" how many strings of all lengths smaller or equal than some small length the two automata accepts. These numbers can only be computed if the alphabet is nite and it is unclear whether one can e ciently adapt this algorithm to the symbolic setting.

Some symbolic models are still not well understood because they do not have a nite automata counterpart. In particular, s-EFAs do not enjoy many good properties, but it is possible that they have practical subclasses| e.g., deterministic, unambiguous, etc.| with good properties.

Finally, the problem of learning symbolic automata has only received limited attention [25], and there is an opportunity to develop interesting new theories in this domain. Classic learning algorithm require querying an oracle for all characters in the alphabet and this is impossible for symbolic automata. On the other hand, the learner simply needs to learn the predicates on each transition of the s-FA, which might require a nite number of queries to the oracle [25]. This is a common problem in computational learning theory and there is an opportunity to apply concepts from this domain to the problem of learning symbolic automata.

6.2 Theoretical treatments

Complexity and expressiveness. In classic automata theory, the complexities of the algorithms are given with respect to the number of states and transitions

in the automaton. We discussed in Section 2 how the complexities of symbolic automata and transducers operations depend on the complexities of performing certain operations in the alphabet theory. Existing structural complexity results for automata algorithms only dwell on state size, but we showed how certain algorithms pose trade-off between state complexity and alphabet complexity in the case of symbolic automata. Exactly understanding these trade-offs is an interesting research question.

There has been a lot of interest in providing algebraic and co-algebraic treatments of classic automata theory [11]. These abstract treatments are helping us understand the essence of classic algorithms and are simplifying complex proofs that were otherwise tedious. It is unclear how to extend these notions to symbolic models, making the problem intriguing from a theoretical standpoint.

Combination with nominal automata. In data words, each character is a pair (a, d) where a is an element of the finite alphabet and d is a data element over an infinite potentially ordered domain. Various models of automata have been introduced for data words [9]. In these models, data elements at different positions can be compared using a predefined operator—e.g., equality—but individual data elements cannot be checked against predicates in a Boolean algebra. Nominal automata [37] provide an elegant algebraic model for describing computations on data words and combining nominal automata with symbolic automata is an interesting research direction: on one hand we know that s-EFA do not enjoy good theoretical properties because they allow comparisons between different characters, and on the other hand nominal automata enjoy decidable properties by restricting what operations one can use to compare data elements.

6.3 New potential applications

SMT solving with sequences. SMT solvers such as Z3 [24] have drastically changed the world of programming languages and turned previously unsolvable problems into feasible ones. The recent interest in verifying programs operating over sequences has created a need for extending existing SMT solving techniques to handle sequences over complex theories [53, 22]. Solvers that are able to handle strings, typically do so by building automata and then performing complex operations over such automata [35]. Existing solvers only handle strings over finite small alphabets [35] and s-FAs have the potential to impact the way in which such solvers for SMT are built. Recently, Z3 [24] has started incorporating s-FAs to reason about sequences. The SMT community has also been discussing how to integrate sequences and regular expressions into the SMT-lib standard [10].

Security. Dalla Preda et al. recently investigated how to use s-FAs to model program binaries [15]. s-FAs can use their state space to capture the control flow of a program and their predicates to abstract the I/O semantics of basic blocks appearing in the programs. This approach unifies existing syntactic and semantic techniques for similarity of binaries and has the promise to lead us to better understand techniques for malware detection in low-level code. The same

authors recently started investigating whether, using s-FTs, the same techniques could be extended to perform analysis of re ective code| i.e., code that can self-modify itself at runtime [42].

7 Conclusion

Symbolic automata and transducers have proven to be a versatile and powerful model to reason about practical applications that were beyond the reach of models that operate over nite alphabets. In this paper, we summarized what theoretical results are known for symbolic models, described the numerous extensions of symbolic automata and transducers, and clari ed why these models are di erent from their nite-alphabet counterparts. We also presented the following list of open problems we hope that the research community will help us solve: Can we provide theoretical treatments of the complexities of the algorithms for symbolic models? Can we extend algorithms for automata over nite alphabets to the symbolic setting? Can we combine symbolic automata with other automata models such as nominal automata? Can we use symbolic automata algorithms to design decision procedures for the SMT theory of sequences?

References

1. *AutomataDotNet*. `https://github.com/AutomataDotNet/`.
2. *AutomatArk*. `https://github.com/lorisdanto/automatark`.
3. *Symbolic Automata*. `http://pages.cs.wisc.edu/ loris/symbolicautomata.html`.
4. *symbolicautomata*. `https://github.com/lorisdanto/symbolicautomata/`.
5. P. Abdulla, J. Deneux, L. Kaati, and M. Nilsson. Minimization of non-deterministic automata with large alphabets. In *CIAA 2005*, LNCS, pages 31–42. Springer, 2006.
6. C. Allauzen and M. Mohri. Finitely subsequential transducers. *International Journal of Foundations of Computer Science*, 14(6):983–994, 2003.
7. R. Alur, L. D'Antoni, and M. Raghothaman. Drex: A declarative language for efficiently evaluating regular string transformations. *ACM SIGPLAN Notices – POPL'15*, 50(1):125–137, 2015.
8. G. Argyros, I. Stais, A. Kiayias, and A. D. Keromytis. Back in black: Towards formal, black box analysis of sanitizers and filters. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 91–109, 2016.
9. M. Benedikt, C. Ley, and G. Puppis. *Automata vs. Logics on Data Words*, pages 110–124. Springer, Berlin, Heidelberg, 2010.
10. N. Bjørner, V. Ganesh, R. Michel, and M. Veanes. An SMT-LIB format for sequences and regular expressions. In *SMT workshop*, 2012.
11. F. Bonchi, M. M. Bonsangue, H. H. Hansen, P. Panangaden, J. J. M. M. Rutten, and A. Silva. Algebra-coalgebra duality in Brzozowski's minimization algorithm. *ACM Trans. Comput. Logic*, 15(1):3:1–3:29, Mar. 2014.
12. M. Botinčan and D. Babić. Sigma*: symbolic learning of input-output specifications. *ACM SIGPLAN Notices – POPL'13*, 48(1):443–456, 2013.
13. A. Bouajjani, P. Habermehl, and T. Vojnar. *Abstract Regular Model Checking*, pages 372–386. Springer International Publishing, Berlin, Heidelberg, 2004.

14. A. K. Chandra, D. C. Kozen, and L. J. Stockmeyer. Alternation. *Journal of the ACM*, 28(1):114–133, Jan. 1981.
15. M. Dalla Preda, R. Giacobazzi, A. Lakhotia, and I. Mastroeni. Abstract symbolic automata: Mixed syntactic/semantic similarity analysis of executables. *ACM SIGPLAN Notices – POPL'15*, 50(1):329–341, 2015.
16. L. D'Antoni and R. Alur. Symbolic visibly pushdown automata. In *CAV'14*, pages 209–225, Cham, 2014. Springer International Publishing.
17. L. D'Antoni, Z. Kincaid, and F. Wang. A symbolic decision procedure for symbolic alternating finite automata. In *MFPS'17*, 2017.
18. L. D'Antoni and M. Veanes. Minimization of symbolic automata. *ACM SIGPLAN Notices – POPL'14*, 49(1):541–553, 2014.
19. L. D'antoni and M. Veanes. Extended symbolic finite automata and transducers. *Formal Methods System Design*, 47(1):93–119, Aug. 2015.
20. L. D'Antoni and M. Veanes. Minimization of symbolic tree automata. In *LICS '16*, pages 873–882, New York, NY, USA, 2016. ACM.
21. L. D'Antoni and M. Veanes. Forward bisimulations for nondeterministic symbolic finite automata. In *TACAS'17*, LNCS, pages 518–534. Springer, 2017.
22. L. D'Antoni and M. Veanes. Monadic second-order logic on finite sequences. *ACM SIGPLAN Notices – POPL'17*, 52(1):232–245, 2017.
23. L. D'Antoni, M. Veanes, B. Livshits, and D. Molnar. Fast: A transducer-based language for tree manipulation. *ACM TOPLAS*, 38(1):1–32, 2015.
24. L. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *TACAS'08*, LNCS, pages 337–340. Springer, 2008.
25. S. Drews and L. D'Antoni. Learning symbolic automata. In *TACAS'17*, pages 173–189, 2017.
26. C. Flanagan and S. Qadeer. Predicate abstraction for software verification. *ACM SIGPLAN Notices – POPL'02*, 37(1):191–202, 2002.
27. Z. Fülöp and P. Gyenizse. On injectivity of deterministic top–down tree transducers. *Information processing letters*, 48(4):183–188, 1993.
28. Z. Fülöp and H. Vogler. Forward and backward application of symbolic tree transducers. *Acta Informatica*, 51(5):297–325, 2014.
29. T. Griffiths. The unsolvability of the equivalence problem for Λ-free nondeterministic generalized machines. *Journal of the ACM*, 15:409–413, 1968.
30. P. Hooimeijer, B. Livshits, D. Molnar, P. Saxena, and M. Veanes. Fast and precise sanitizer analysis with bek. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11, pages 1–1, Berkeley, CA, USA, 2011. USENIX Association.
31. P. Hooimeijer and M. Veanes. An evaluation of automata algorithms for string analysis. In *VMCAI'11*, volume 6538 of *LNCS*, pages 248–262. Springer, 2011.
32. J. Hopcroft. An $n\log n$ algorithm for minimizing states in a finite automaton. In Z. Kohavi, editor, *Theory of machines and computations, Proc. Internat. Sympos., Technion, Haifa, 1971*, pages 189–196, 1971.
33. Q. Hu and L. D'Antoni. Automatic program inversion using symbolic transducers. In *ACM SIGPLAN Notices – PLDI '17 (To appear)*, 2017.
34. M. Keil and P. Thiemann. Symbolic solving of extended regular expression inequalities. In *FSTTCS'14*, LIPIcs, pages 175–186, 2014.
35. T. Liang, A. Reynolds, C. Tinelli, C. Barrett, and M. Deters. A DPLL(T) theory solver for a theory of strings and regular expressions. In *CAV'14*, pages 646–662, 2014.
36. K. Mamouras, M. Raghotaman, R. Alur, Z. G. Ives, and S. Khanna. StreamQRE: Modular specification and efficient evaluation of quantitative queries over streaming data. In *ACM SIGPLAN Notices – PLDI '17 (To appear)*, 2017.

37. J. Moerman, M. Sammartino, A. Silva, B. Klin, and M. Szynwelski. Learning nominal automata. *ACM SIGPLAN Notices – POPL'17*, 52(1):613–625, 2017.
38. M. Mohri. Finite-state transducers in language and speech processing. *Computational Linguistic*, 23(2):269–311, June 1997.
39. T. Mytkowicz, M. Musuvathi, and W. Schulte. Data-parallel finite-state machines. *ACM SIGPLAN Notices – ASPLOS'14*, 49(4):529–542, 2014.
40. P. Ohmann, A. Brooks, L. D'Antoni, and B. Liblit. Control-flow recovery from partial failure reports. In *ACM SIGPLAN Notices – PLDI '17 (To appear)*, 2017.
41. R. Paige and R. E. Tarjan. Three partition refinement algorithms. *SIAM Journal on Computing*, 16(6):973–989, 1987.
42. M. D. Preda, R. Giacobazzi, and I. Mastroeni. Completeness in approximate transduction. In *SAS'16*, pages 126–146, 2016.
43. K. Y. Rozier and M. Y. Vardi. A multi-encoding approach for LTL symbolic satisfiability checking. In *FM'11*, pages 417–431, 2011.
44. O. Saarikivi and M. Veanes. Minimization of symbolic transducers. In *CAV'17*, LNCS. Springer, 2017.
45. O. Saarikivi, M. Veanes, T. Mytkowicz, and M. Musuvathi. Fusing effectful comprehensions. In *ACM SIGPLAN Notices – PLDI'17 (To appear)*. ACM, 2017.
46. M. P. Schützenberger. Sur une variante des fonctions séquentielles. *Theoretical Computer Science*, 4:47–57, 1977.
47. R. E. Stearns and H. B. Hunt. On the equivalence and containment problems for unambiguous regular expressions, grammars, and automata. In *sfcs'81*, pages 74–81, Oct 1981.
48. N. Tillmann and J. de Halleux. Pex - white box test generation for .NET. In *TAP'08*, LNCS, pages 134–153, Prato, Italy, April 2008.
49. G. van Noord and D. Gerdemann. Finite state transducers with predicates and identities. *Grammars*, 4(3):263–286, 2001.
50. M. Veanes. Symbolic string transformations with regular lookahead and rollback. In *PSI'14*, LNCS, pages 335–350, 2015.
51. M. Veanes and N. Bjørner. Symbolic tree transducers. In *PSI'11*, pages 377–393, 2011.
52. M. Veanes and N. Bjørner. Symbolic tree automata. *Information Processing Letters*, 115(3):418–424, 2015.
53. M. Veanes, N. Bjørner, and L. de Moura. Symbolic automata constraint solving. In *LPAR-17*, LNCS, pages 640–654, 2010.
54. M. Veanes, N. Bjørner, L. Nachmanson, and S. Bereg. Monadic decomposition. *Journal of the ACM*, 64(2):14:1–14:28, 2017.
55. M. Veanes, P. de Halleux, and N. Tillmann. Rex: Symbolic regular expression explorer. In *ICST'10*, pages 498–507. IEEE, 2010.
56. M. Veanes, P. Grigorenko, P. de Halleux, and N. Tillmann. Symbolic query exploration. In *ICFEM'09*, LNCS, pages 49–68. Springer, 2009.
57. M. Veanes, P. Hooimeijer, B. Livshits, D. Molnar, and N. Bjørner. Symbolic finite state transducers: Algorithms and applications. *ACM SIGPLAN Notices – POPL'12*, 47(1):137–150, 2012.
58. M. Veanes, T. Mytkowicz, D. Molnar, and B. Livshits. Data-parallel string-manipulating programs. *ACM SIGPLAN Notices – POPL'15*, 50(1):139–152, 2015.
59. B. W. Watson. Implementing and using finite automata toolkits. In *Extended finite state models of language*, pages 19–36. Cambridge University Press, 1999.