DOI:10.1145/3419404

Symbolic automata better balances how automata are implemented in practice.

BY LORIS D'ANTONI AND MARGUS VEANES

Automata Modulo Theories

FINITE AUTOMATA ARE one of the most fundamental models of computation and are taught in almost all undergraduate computer-science curricula. Although automata are typically presented as a theoretical model of computation, they have found their place in a variety of practical applications, such as natural language processing, networking, program verification, and regular-expression matching. In this article, we argue that the classical definition of finite automata is not well suited for practical implementation and present symbolic automata, a model that addresses this limitation.

In most of the literature, finite automata are intuitively presented as follows. A finite automaton is a graph that describes a set of strings over a finite alphabet Σ. In the graph, nodes are called states, some states are initial, some states are final, and each edge "reads" a symbol from Σ. Every path from an initial to a final state describes a string accepted by the automaton. For example, the automaton in Figure 1(a), given $\Sigma = \{0, 1\}$, describes the set of all binary strings starting with 0. Researchers have designed a number of decision procedures and extensions for this simple but very powerful model.

As discussed in textbooks, example Hopcroft and Ullman,[22] the aforementioned way of defining automata is simple and easy to implement.

> *Any automaton can be represented by a set of edges, where an edge (p, a, q) goes from state p to q reading symbol a.*

For certain applications, such as natural language processing, this type of implementation is appropriate. However, in many applications (particularly in software verification), the alphabet over which the automaton has to operate is so big (sometimes infinite) that an explicit representation of all its symbols is infeasible.

The following two applications of finite automata are examples of this problem.

Regular expressions. Automata are used in most implementations of regular-expression matching and analysis algorithms. In this domain, the alphabet Σ_R is typically very large, ranging between

» key insights

- ***Gap*: Real implementation of finite automata use the structure of the alphabet, for example, character classes, to enable efficient data structures and algorithms. These aspects are missing in the way automata are taught and defined in the literature.**
- ***Innovation*: Symbolic automata match how automata are implemented in practice by representing the alphabet and automaton structures separately. This separation yields a more general model that can handle complex and infinite alphabets while retaining all the desirable properties of finite automata and enabling new practical applications.**
- ***Opportunity*: Separating the alphabet and automaton representations opens opportunities for redesigning the ways in which we teach automata and design algorithms for them.**

IMAGE BY NATASH/SHUTTERSTOCK

hundreds and millions of characters, for example, extended ASCII has 256 characters and Unicode has over one million characters. A quick Web search of finite-automata implementations for regular expressions shows that in all the performing implementations, transitions (that is, the edges of the graph) *do not* "read" a single character in Σ_R as done in the classical definition.[a] Instead, practical implementations allow transitions to carry (that is, "read") *sets of characters*, that is, an edge in the graph denotes a transition of the form (p, S, q) where $S \subseteq \Sigma_R$, see Figure 1(b). Moreover, in most implementations, S is represented using a dedicated data structure, for example, an interval ['0', '9'] representing the set of all ASCII decimal digits {'0', '1', …, '9'}, see Figure 1(c). Use of intervals assumes that all characters have a numeric code that implies a total order over Σ_R—for example, the ith decimal ASCII digit has code $48+i$.

Model checking. Automata are the backbone of many model checking algorithms,[35] where these models are used to describe properties a system must obey. In model checking, the alphabet is typically defined with respect to a finite set $AP_k = \{b_i\}_{i=0}^{k-1}$ of Boolean properties of interest, often called *atomic propositions*. The alphabet is $\Sigma_{MC} = 2^{AP_k}$, where a character $\alpha \in \Sigma_{MC}$ represents the assignment $\{b = true\}_{b \in \alpha} \cup \{b = false\}_{b \in AP \setminus \alpha}$. In practical implementations of automata for model checking, transitions do not read individual characters and instead read *sets of characters*.[18] The tools assume a particular representation of the sets, for example as binary decision diagrams, see Figure 1(d), or as Boolean formulas φ over the variables AP where each satisfying assignment defines a character α, denoted by $\alpha \models \varphi$. For example, given $k = 7$, the predicate $\varphi = \overline{b_6} \wedge b_5 \wedge b_4 \wedge (\overline{b_3} \vee (\overline{b_2} \wedge \overline{b_1}))$ in Figure 1(f) has exactly 10 satisfying assignments, such as $\{b_0, b_4, b_5\} \models \varphi$.

The above applications clarify the following point.

Practical implementations of automata allow transitions to carry sets of characters instead of individual characters and take advantage of the structure of the input alphabet.

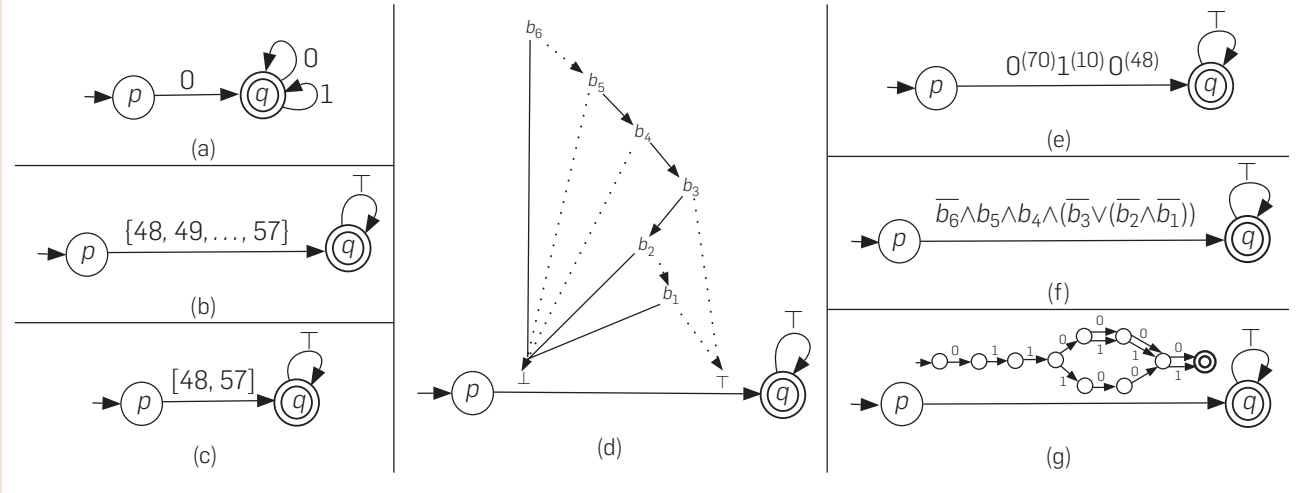
These examples show that the classical theory of automata does not do a good job in capturing the ways in which automata are implemented in practice. Therefore, all the theoretical results—for example, decision procedures and closure properties—do not directly apply to such implementations and cannot "take advantage" of the structure present in how automata are implemented. *Symbolic automata* and their variants (the models presented in this article) address this problem.

Symbolic automata explicitly model how the alphabet is represented/implemented in practical applications and allow us to design automata decision procedures that take advantage of the alphabet representation.

Symbolic automata extend finite automata by allowing transitions to carry predicates over rich alphabet theories (for example, intervals or binary decision diagrams). Unlike what is done in existing implementations of automata, where the alphabet representation is chosen a priori and hard-coded in the model, symbolic automata explicitly separate the representation of the alphabet structure from one of the finite state graph structures. Concretely, symbolic automata provide a unifying approach for different alphabet

a regex-automata (https://github.com/BurntSushi/regexautomata/blob/master/src/nfa.rs), Brics library (https://www.brics.dk/automaton/), Dregex - Deterministic Regular Expression Engine (https://github.com/marianobarrios/dregex/blob/master/src/main/scala/dregex/impl/Dfa.scala).

Figure 1. (a) classical automaton over the alphabet {0, 1} accepting all strings that start with 0; (b–g) symbolic automata accepting all ASCII strings starting with a decimal digit with predicates represented by: (b) hashsets; (c) intervals; (d) binary decision diagrams where dashed arrow is case $b_i = 0$; (e) bitvectors; (f) Boolean formulas; and (g) finite automata over the alphabet {0, 1} restricted to {0, 1}$^{(7)}$.



representations by representing the alphabet using an effective Boolean algebra $\mathcal{A}$, also called the alphabet theory. The automata algorithms are now designed *modulo* $\mathcal{A}$. This separation of concerns allows one to seamlessly change the alphabet representation (for example, choosing any of the representations in Figure 1), without changing any of the underlying automata algorithms (that is, each decision procedure is implemented in a way that is agnostic to the choice of the alphabet theory).

Not only symbolic automata better reflect how automata are implemented in practice, but they also allow one to represent strings over infinite alphabets, for example, the set of rational numbers. Despite this increase in expressiveness, symbolic automata enjoy many of the desirable closure and decidability properties of finite automata and have been in fact used in a variety of applications: verification of functional programs operating over lists and trees,[13] analysis of complex implementations of BASE64 and UTF encoders,[10] automatic synthesis of inverses of complex string-manipulating programs,[23] analysis of programs involving regular expressions over large alphabets,[8,9,36] automatic parallelization of list-processing code,[31] solving constraints over sequences,[33] vulnerability detection in Web-applications,[4] analysis of program binaries,[5] and fast execution of string transformations.[1,28]

The goal of this article is to give an overview of what is currently known about symbolic automata and their variants, and what applications these models have enabled. The concept of automata with predicates instead of concrete symbols was first mentioned in Watson.[39] This article focuses on work done following the definition of symbolic finite automata presented in Veanes et al.,[36] 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 binary decision diagrams. This meaning is different from the one described in this article.

It is hard to describe all the works related to symbolic automata in one article, and the authors curate an updated list of papers on symbolic automata and transducers (http://pages.cs.wisc.edu/~loris/symbolic-automata.html). Moreover, the algorithms discussed in this article are implemented in the open source libraries AutomataDotNet (C#, https://github.com/AutomataDotNet/) and symbolic automata (Java, https://github.com/lorisdanto/symbolicautomata/).

Symbolic automata extend finite automata by allowing transitions to carry predicates over rich alphabet theories.

Symbolic Automata

Structured alphabets and Boolean algebras. As we illustrated in Figure 1, in symbolic automata, transitions carry predicates, and to formally describe predicates, we need to define Boolean algebras. 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 $\bot, \top \in \Psi$; and 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—that is, whether $[\![\varphi]\!] \neq \emptyset$—is *decidable*. $\mathcal{A}$ is extensional when $[\![\varphi]\!] = [\![\psi]\!]$ implies that $\varphi = \psi$—that is, semantic equivalence coincides with syntactic equivalence.

The following example shows how some of the character representations described in Figure 1 are Boolean algebras.

Example 2.1 (ASCII Algebras). Regular expressions in modern programming languages use *character classes* as basic building blocks. Character classes can be unioned and complemented. For example, the character classes [A-Z] and [a-z] denote the set of all ASCII upper and lower case letters, respectively, [A-Za-z] denotes their union, and [^A-Za-z] denotes the complement of [A-Za-z]. A natural choice for domain $\mathfrak{D}$ is the set of all ASCII codes $\{n \mid 0 \leq n \leq 127\}$. Figure 1(b–g) shows several different options for representing Ψ. Consider the following representations of $\varphi \in \Psi$:

(a) φ is a binary decision diagram (BDD) over AP_7 (Figure 1(d)); $\wedge$ is AND-product of BDDs; and $\bot$ is the false-leaf BDD;
(b) φ is a 128-bit bitvector whose nth bit is 1 iff $n \in [\![\varphi]\!]$ (Figure 1(e)); $\wedge$

is bitwise-AND; and $\bot$ is $0^{(128)}$—that is, a sequence of 128 zeroes;

(c) φ is a Boolean formula over AP_7 (Figure 1(f)); $\wedge$ is syntactic conjunction; and $\top$ is the true predicate.

(d) φ is an automaton over $\Sigma = \{0, 1\}$ restricted to $\Sigma^{(7)}$ (Figure 1(g)); $\wedge$ is product of automata; and $\top$ is the fixed automaton accepting all strings in $\Sigma^{(7)}$.

We identify the underlying semantic domain $\mathfrak{D}$ in this example with character codes. This reflects the fact that implementations that use ASCII often use expressions such as A and \x41 as equivalent notations of the same underlying character (code). In other words, $[\![\varphi]\!] \subseteq \mathfrak{D}$ is a set of natural numbers, namely the set of all character codes denoted by φ. In particular, in (a) and (c), $[\![\varphi]\!] = \{\sum_{b_i \in \alpha} 2^i \mid \alpha \vDash \varphi\}$, and in (d), $[\![\varphi]\!] = \{\sum_{1 \le i \le 7, v_i = 1} 2^{7-i} \mid v \in \mathcal{L}(\varphi)\}$. Observe that the algebras in (a) and (b) are extensional where satisfiability is trivial (nonequality to $\bot$). For example, in (b), the disjunction $0^{(64)}1^{(64)} \vee 1^{(64)}0^{(64)}$ (where $\vee$ is bitwise-OR) is the predicate $1^{(128)}$, that is, $\top$. The algebra in (c) is nonextensional and satisfiability can be decided using a satisfiability (SAT) solver. The algebra in (d) is also nonextensional but its satisfiability is trivial, assuming unreachable states and deadends are always removed from (the automaton) φ. ⊠

The following Boolean algebra mimics how finite alphabets are represented in traditional automata implementations.[b]

b Technically, s-FAs over the equality algebra are still slightly more general than traditional automata as s-FAs can still allow individual transitions to carry multiple characters.

Example 2.2 (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 $\top$. There are no formal requirements on $\mathfrak{D}$, but one case is that $\mathfrak{D} = \Sigma_{MC}$ is a powerset of a finite set of Boolean properties as discussed above. The set of predicates Ψ is the Boolean closure generated from the atomic predicates—for example, $\varphi_a \vee \varphi_b$ and $\neg\varphi_a$, where $a, b \in \mathfrak{D}$ are predicates in Ψ. Intuitively, one can think of each predicate in this algebra to be of the form $\lambda x.\ x = a_1 \vee \ldots \vee x = a_n$, where each a_i is an element of $\mathfrak{D}$. Hence, the name equality algebra. ⊠

The following example shows how the predicate representations employed by modern SMT solvers[14] can be used to design Boolean algebras for arbitrarily complex domains.

Example 2.3 (SMT Algebra). Let Ψ be the set of all quantifier-free formulas with one fixed free variable x of a fixed type τ. Formally, $SMT_\tau = (\mathfrak{D}, \Psi, [\![\_]\!], \bot, \top, \vee, \wedge, \neg)$, where $\mathfrak{D}$ is the domain of τ, and the Boolean operations are the corresponding connectives in SMT formulas. Intuitively, SMT_τ represents a restricted use of an SMT solver. The interpretation function $[\![\varphi]\!]$ is defined using the operations of satisfiability checking and witness generation of an SMT solver. For example, in $SMT_{\mathbb{Z}}$, elements are integers and predicates are linear arithmetic formulas, such as $\varphi_{>0} \stackrel{\text{def}}{=} x > 0$ and $\varphi_{odd} \stackrel{\text{def}}{=} x\%2 = 1$. ⊠

Symbolic finite automata. We can now define symbolic finite automata, which are finite automata where edge labels are replaced by predicates in a Boolean algebra.

Definition 2.4. 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*.

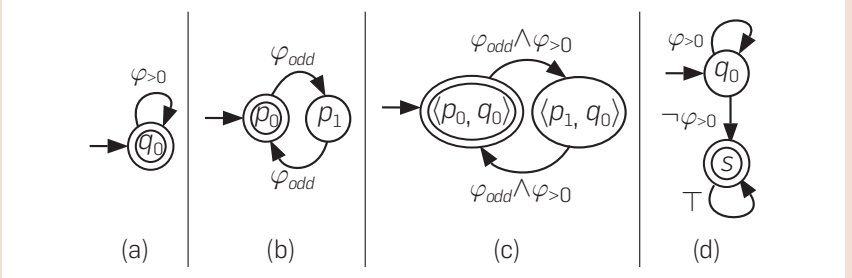
Elements of the domain $\mathfrak{D}$ of $\mathcal{A}$ are called characters, and elements of $\mathfrak{D}^*$ are called strings. A transition $\rho = (q_1, \varphi, q_2) \in \Delta$, also denoted $q_1 \xrightarrow{\varphi} q_2$, has *source* state q_1, *target* state q_2, and *guard* φ. For $a \in \mathfrak{D}$, the concrete a-transition $q_1 \xrightarrow{a} q_2$ denotes that $q_1 \xrightarrow{\varphi} q_2$ and $a \in [\![\varphi]\!]$ for some φ; M is *deterministic* when there is at most one concrete a-transition from any source q_1 and character a.

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

It is convenient to work with s-FAs that are *normalized* by treating Δ as a function from $Q \times Q$ to Ψ with $\Delta(p,q) = \bot$ when there is no transition from p to q. To this end, let $\Delta(p,q) = \vee\{\varphi \mid (p,\varphi,q) \in \Delta\}$, where $\vee\emptyset = \bot$. We also define $dom(p) \stackrel{\text{def}}{=} \vee\{\varphi \mid \exists q : (p,\varphi,q) \in \Delta\}$, as the domain-predicate of p, and say that p is *complete* if $[\![dom(p)]\!] = \mathfrak{D}_{\mathcal{A}}$; p is partial otherwise. Observe that p is partial iff $\neg dom(p)$ is satisfiable. M is *complete* if all states of M are complete; M is *partial* otherwise.

Example 2.5. Examples of s-FAs are $\mathbf{M_{pos}}$ and $\mathbf{M_{ev/odd}}$ in Figure 2(a) and (b), respectively. These two s-FAs have 1 and 2 states, respectively, and they both operate over the Boolean algebra $SMT_{\mathbb{Z}}$ from Example 2.3. The s-FA $\mathbf{M_{pos}}$ accepts all strings consisting only of positive numbers, whereas 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 [1, 3, 5, 3] and rejects strings [1, 3, 5] 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 $\mathcal{L}(\mathbf{M_{pos}}) \cap \mathcal{L}(\mathbf{M_{ev/odd}})$ (Figure 2(c)). Both s-FAs are partial—for example, neither of them has transitions for character 0. Finally, $\mathbf{M^c_{pos}}$ accepts the complement of the language of $\mathbf{M_{pos}}$ (Figure 2(d)). ⊠

Figure 2. Symbolic automata: (a) $\mathbf{M_{pos}}$; (b) $\mathbf{M_{ev/odd}}$; (c) $\mathbf{M_{ev/odd}} \times \mathbf{M_{pos}}$; and (d) $\mathbf{M^c_{pos}}$.



Properties of symbolic automata. In this section, we illustrate some basic properties of s-FAs and show how, although these models are more expressive (they can model infinite alphabets) and succinct (they allow multiple characters on individual transitions) than finite automata,

they still enjoy many desirable decidability and closure properties. A key characteristic of all s-FAs algorithms is that there is no explicit use of characters because $\mathfrak{D}$ may be infinite and the interface to the Boolean algebra does not directly support use of individual characters. This aspect is in sharp contrast to traditional finite automata algorithms.

Closure properties. First, such as for finite automata, nondeterminism does not add expressiveness for s-FAs.

THEOREM 2.6 (DETERMINIZABILITY[36]). *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 finite alphabets, but also requires combining predicates appearing in different transitions. If M contains k inequivalent predicates and n states, then the number of distinct predicates (other than $\top$) 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 classical state space explosion risk, there is also a predicate space explosion risk. Figure 3 illustrates such explosion for $k = 2$ and $n = 2$.

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

THEOREM 2.7 (BOOLEAN OPERATIONS[36]). *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 classical product construction in which transitions are "synchronized" using conjunction. For example, the intersection of $\mathbf{M}_{\text{pos}}$ and $\mathbf{M}_{\text{ev/odd}}$ from Example 2.5 is shown in Figure 2(c). To complement a deterministic partial s-FA M, M is first *completed* by adding a new nonfinal state s with loop $s \xrightarrow{\top} s$ and for each partial state p a transition $p \xrightarrow{\neg dom(p)} s$. Then, the final states and the nonfinal states are swapped in M^c. Following this procedure, the complement of $\mathbf{M}_{\text{pos}}$ from Example 2.5 is shown in Figure 2(d).

Decision procedures. s-FAs enjoy the same decidability properties of finite automata:

THEOREM 2.8 (DECIDABILITY[36]). *Given s-FAs M_1 and M_2 it is decidable to check if M_1 is empty—that is, whether $\mathcal{L}(M_1) = \emptyset$—and if M_1 and M_2 are language-equivalent—that is, whether $\mathcal{L}(M_1) = \mathcal{L}(M_2)$.*

Checking emptiness requires checking what transitions are satisfiable and, once unsatisfiable transitions are removed, any path reaching a final state from an initial state represents at least one accepting string. Equivalence can be reduced to emptiness using closure under Boolean operations—that is, $\mathcal{L}(M_1) = \mathcal{L}(M_2) \Leftrightarrow \mathcal{L}(M_1^c \times M_2) = \emptyset \wedge \mathcal{L}(M_1 \times M_2^c) = \emptyset$.

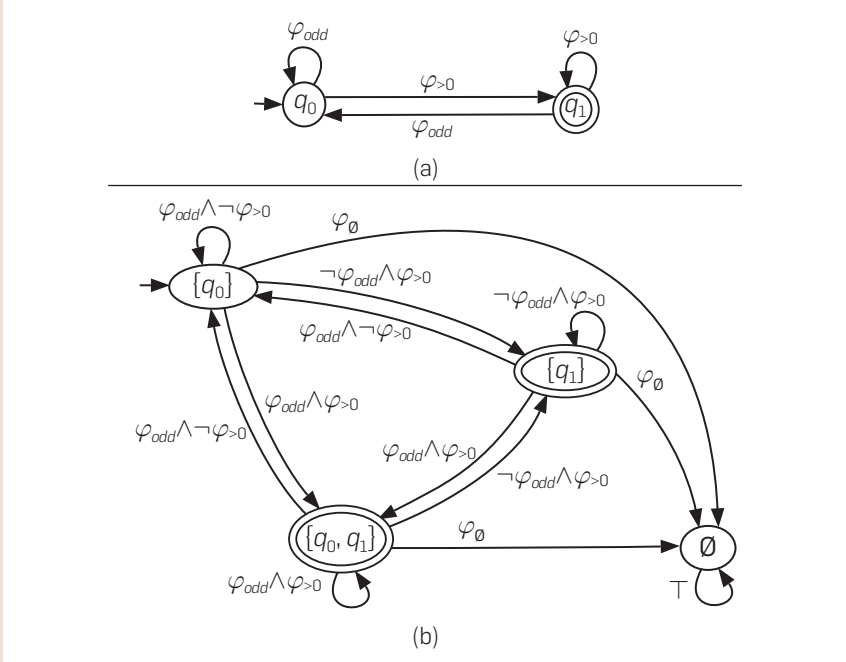
Algorithms have also been proposed for minimizing *deterministic* s-FAs,[9] for checking language inclusion,[25] and for learning s-FAs from membership and equivalence queries.[3,26,27]

Parametric complexities. Because s-FAs are parametric in an underlying alphabet theory $\mathcal{A}$, the complexities of the aforementioned algorithms must in some way depend on the complexities of performing certain operations in $\mathcal{A}$. For example, the cost of checking emptiness of an s-FA depends on the cost of checking satisfiability in $\mathcal{A}$. Another issue is representation of Ψ, how to measure the size $|\varphi|$ of $\varphi \in \Psi$, and the cost of Boolean operations whose repeated applications may cause predicates to grow in size and thus increase related costs, in particular when $\mathcal{A}$ is not extensional. This peculiar aspect of s-FAs opens up a new set of complexity questions that have not been studied in automata theory.

We summarize the known complexity results for the procedures we described in this section. Let $f_{\text{sat}}(|\varphi|)$ denote the cost of checking satisfiability of a predicate $\varphi \in \Psi$ (of size $|\varphi|$) and M and M' be two *deterministic* s-FAs such that M has n states and m transitions, M' has n' states and m' transitions, and the largest predicate in M and M' has size ℓ. Then, (i) checking whether M is empty has complexity $O(n + mf(\ell))$, and (ii) checking whether M and M' are language equivalent has complexity $\mathcal{O}(mm'f(\ell)(n + n') + \alpha(n + m))$, where $\alpha(\cdot)$ is related to a functional inverse of the Ackermann function. This last complexity is obtained by a straightforward adaptation of Hopcroft and Karp's algorithm for DFA equivalence.[21]

For certain classes of problems, different algorithms can have different incomparable complexities. Consider,

Figure 3: (a) Nondeterministic partial s-FA with predicates φ_{odd} and $\varphi_{>0}$ from Example 2.3. (b) Equivalent complete s-FA after determinization where $\varphi_{\emptyset}$ denotes the predicate $\neg\varphi_{odd} \wedge \neg\varphi_{>0}$ on transitions to the "sink" state Ø.



for example, the problem of minimizing a deterministic s-FA M and let $f_{sat}(|\varphi|)$ denote the cost of checking satisfiability of a predicate $\varphi \in \Psi$. If M has n states and m transitions, and the largest predicate in M has size ℓ, then the symbolic adaptation of Moore's algorithm for minimizing DFAs has complexity $\mathcal{O}(mn \cdot f_{sat}(\ell))$, whereas the symbolic adaptation of Hopcroft's algorithm for minimizing DFAs has complexity $\mathcal{O}(m \log n \cdot f_{sat}(n\ell))$, if the cost of Boolean operations is linear.[9] Although in the world of DFAs, Hopcroft's algorithm[20] is provably better than Moore's,[29] in the world of s-FAs, the two algorithms have orthogonal complexities: Hopcroft's algorithm saves a logarithmic factor in terms of n, but this saving comes at the increased cost of satisfiability queries.

Reduction to classical automata. Although the set S of predicates appearing in a given s-FA (or finitely many s-FAs over the same alphabet algebra) operates generally over an infinite domain, the set of *minterms*, *Minterms*(S), the maximal satisfiable Boolean combinations of S induces a finite set of equivalence classes. In general, every s-FA M can be compiled into a symbolically equivalent finite automaton over alphabet *Minterms*(S), where *Predicates*(M) $\subseteq S$ and S is a finite subset of Ψ. This idea, also called predicate abstraction, is often used in program verification.[16] However, note that the size of *Minterms*(S) is in general exponential in the size of S.

Extensions of the Basic s-FA Model

Symbolic automata have been extended in various ways, as summarized in D'Antoni and Veanes[12]:

- Symbolic alternating automata (s-AFA)[8] are equivalent in expressiveness to s-FAs, but achieve succinctness by extending s-FAs with alternation and, despite the high theoretical complexity, this model can at times be more practical than s-FAs. For example, s-AFAs can efficiently check whether the intersection of multiple ASCII regular expressions is empty in cases where traditional automata suffer from state explosion.
- Symbolic extended finite automata (s-EFA)[10] allow s-FAs to read multiple input characters in a single transition and can be used to model relations between adjacent input symbols.
- Symbolic tree automata (s-TA)[13] operate over trees instead of strings. s-FAs are a special case of s-TAs in which all nonleaf nodes in the tree have one child. s-TAs have the same closure and decidability properties as s-FAs and can also be minimized efficiently.
- Symbolic visibly pushdown automata (s-VPA)[6] operate over nested words,[2] which are used to model data with both linear and hierarchical structure—for example, XML documents and recursive program traces. s-VPAs can be determinized and have the same closure and decidability properties as s-FAs.

Symbolic transducers. Automata that also emit string outputs are generally called *transducers*. Perhaps the most practical extension of s-FAs is Symbolic Finite (state) Transducers (s-FTs), which extend finite transducers by allowing outputs to functionally depend on inputs, in addition to allowing predicates over those inputs.

Example 3.1. Consider an *HTML* encoder that is often used as a *string sanitizer* in a Web browser. At a high level, its purpose is to replace all nonwhitelisted characters by their equivalent HTML encoded substrings in an HTML document, for example, the character < can be encoded as the string "<" (or as the string "<" using its character code 60). It is prohibitively expensive to view such an encoder as a classical Finite (state) Transducer (FT) because the standard *Unicode alphabet*[c] has 1,112,064 characters. So the FT would in general need over a million transitions, one for each character c: namely, a transition $q \xrightarrow{c/c} q$ if c is whitelisted (safe), and a transition $q \xrightarrow{c/encode(c)} q$ otherwise where *encode*(c) denotes the HTML encoding of c.

Instead, the corresponding s-FT would need only *two* transitions: a transition $q \xrightarrow{\varphi_{safe}/[\lambda x.x]} q$ for *safe* characters (this includes all ASCII uppercase and lowercase letters and digits), and a transition $q \xrightarrow{\neg\varphi_{safe}/[\lambda x.'\&', \lambda x.'\#', \pi_6, \pi_5, \pi_4, \pi_3, \pi_2, \pi_1, \pi_0, \lambda x.';']} q$ where $\pi_k = \lambda x.((x \div 10^k) \bmod 10) + 48$ yields the kth decimal digit from the character x (recall that the standard character code of a decimal digit d is d +48). Note that the output of an s-FT transition is a sequence of functions—in the safe case, the sequence contains a single identity function (outputting the input character). ☒

We discuss next some of the main properties of s-FTs and the kinds of analysis they enable. A more formal treatment is covered in D'Antoni and Veanes.[12]

A transducer T denotes a relation $\mathcal{T}_T \subseteq \mathfrak{D}^* \times \mathfrak{D}^*$ from input to output sequences. When T is deterministic (at most one transition can be triggered by an input symbol), this relation is a function. If T is nondeterministic but each input sequence is mapped to only one output sequence, the relation is also a function. In these last two cases, we call the transducer *functional* or *single-valued*. We call the first and second projections of the relation describing the semantics of the transducer T, its domain (**dom**(T)) and range (**ran**(T)), respectively. We write $\mathcal{T}_T(u)$ for $\{v \mid (u,v) \in \mathcal{T}_T\}$.

Although both the domain and the range of a finite state transducer are regular, this is not true for s-FTs. By a *regular language*, here we mean a language accepted by an s-FA. Given an s-FT T, one can compute an s-FA $DOM(T)$ such that $\mathcal{L}(DOM(T)) = \mathbf{dom}(T)$. The range of an s-FT is in general not regular: 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. The range **ran**(T) of the s-FT T only accepts sequences of numbers where each number at position 0 (respectively, 2,4,...) has to be the same as the number at position 1 (respectively, 3,5,....). Then, **ran**(T) is not regular because an s-FA cannot enforce the equality constraint between two symbols at different positions in the sequence. Also, because the alphabet is *infinite*, states cannot be used to remember what symbol is at each position such as it is normally done for finite automata over finite alphabets.

The most important property of s-FTs is that they are closed under *sequential composition*: Given two s-FTs S and T, one can compute an s-FT $S(T)$ such that:

$$\forall u, v \in \mathfrak{D}^* : (u,v) \in \mathcal{T}_{S(T)} \Leftrightarrow$$
$$\exists w \in \mathfrak{D}^* : (u,w) \in \mathcal{T}_T \wedge (w,v) \in \mathcal{T}_S.$$

c https://home.unicode.org/

This enables several interesting program analyses[19] and optimizations, such as (*symbolic regular*) *type-checking* that is decidable when combined with closure properties of s-FAs:

Given an s-FT T and s-FAs M_I and M_O, decide if for all $v \in \mathcal{L}(M_I)$: $\mathscr{T}_T(v) \subseteq \mathcal{L}(M_O)$.

Treat any s-FA M implicitly as an s-FT by treating each transition $p \xrightarrow{\varphi} q$ as $p \xrightarrow{\varphi(x)/[x]} q$. Then, $M(T)$ restricts *outputs* of T to $\mathcal{L}$(M) and $T(M)$ restricts inputs of T to $\mathcal{L}$(M). So type-checking reduces to emptiness of $\mathscr{T}_{M_O^c(T(M_I))}$.

Example 3.2. By using the type-checking algorithm one can prove that, for every input sequence $v \in \mathfrak{D}^*$, avn HTML-Encoder T always produces a *valid* output: $\mathscr{T}_T(v) \subseteq \mathcal{L}(M_O)$, where M_O disallows all unsafe characters besides '&', and allows '&' only as an encoding prefix—for example, as described by the regular expression "([#0-~]|&(amp|lt|gt|#[0-9]+);)*". Observe that M_O^c accepts all invalid outputs and thus $M_O^c(T)$ represents the restriction of T to unwanted behaviors. ☒

Although equivalence of finite transducers is in general undecidable,[17] equivalence of single-valued s-FTs is decidable and single-valuedness is itself a decidable property.[37] When combined with closure under composition, decidable equivalence is a powerful verification tool. For example, one can check whether composing two s-FTs E and D, representing a string encoder and decoder, respectively, yields the identity function I—that is, if $\mathscr{T}_{D(E)} = \mathscr{T}_I$.

A transducer T is *injective* if for all distinct $u, v \in \mathfrak{D}^*$, we have $\mathscr{T}_T(u) \cap \mathscr{T}_T(v) = \emptyset$. Although injectivity of finite transducers is decidable, injectivity is *undecidable* for s-FTs.[23]

Extensions of symbolic transducers. Similarly to how s-EFAs extend s-FAs, Symbolic Extended Finite Transducers (s-EFT)[10] are symbolic transducers in which each transition can read more than a single character. A similar extension, called s-RTs, incorporates the notion of bounded *look-back* and *roll-back* in form of roll-backtransitions, not present in any other transducer formalisms, to accommodate default or exceptional behavior.

s-FTs have also been extended with *registers* and are called *symbolic transducers*.[10,38] The key motivation is to support loop-carried data state, such as the maximal number seen so far. This model is closed under composition,

Sanitizers provide a first line of defense against cross site scripting attacks.

but most decision problems for it are undecidable, even emptiness.

Symbolic *tree* transducers (s-TT)[13] 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.

Applications of Symbolic Automata and Their Variants

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 alphabet can contain upward of 2^{16} characters due to the widely adopted UTF16 standard of Unicode characters. s-FAs can model characters as bitvectors and use various representations for predicates—for example, Binary Decision Diagrams (BDDs) or bitvector arithmetic. These representations turned out to be a viable way to model regular expression constraints in parameterized unit tests[36] and for random password generation.[9]

Analysis of string encoders and sanitizers. The original motivation for s-FTs was to enable static analysis for string *sanitizers*.[19] String sanitizers are particular string to string functions over Unicode (a very large alphabet that cannot be handled by traditional automata models) 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 first line of defense 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—that is, if $\mathscr{T}_{A(B)} = \mathscr{T}_{B(A)}$—if a sanitizer A is idempotent—that is, if $\mathscr{T}_{A(A)} = \mathscr{T}_A$—or if A cannot be compromised with an input attack vector—that is, if $\mathbf{ran}(A) \subseteq$ *SafeSet*. Checking such properties can help ensure the correct usage of sanitizers. s-EFTs have been used to prove that efficient implementations of BASE64 or UTF encoders and decoders correctly

invert each other.[10] Recently, s-EFTs have been used to automatically synthesize inverses of encoders that are correct by construction.[23]

Analysis of functional programs. Symbolic transducers have been used to perform static analysis of functional programs that operate over lists and trees.[13] In particular, symbolic tree transducers were used to verify HTML sanitizers and to perform deforestation, a technique used to speedup function composition in functional language compilation. These programs cannot be modeled using traditional automata models because they operate over infinite alphabets—for example, lists of integers.

Code generation and parallelization. Symbolic transducers can be used to expose data parallelism in computations that may otherwise seem inherently sequential. A DFA transition function can be viewed as a particular kind of *matrix multiplication*. Therefore, DFA-based pattern matching can be executed in parallel by virtue of associativity of multiplication. This idea can be lifted to symbolic transducers and applied to many common string transformations. Symbolic transducers can also be extended with *registers* and *branching rules*, which are transitions with multiple target states in the form of if-then-else statements. The main purpose of a branching rule is to support built-in determinism and to enable a way to control evaluation order of transition guards for serial code generation, so that *hot paths* can be optimized. Moreover, symbolic transducers can be composed in a manner that is similar to *loop fusion* in order to avoid intermediate data structures. The main context where these ideas have been evaluated is in log/data processing pipelines.[31]

Symbolic regex matcher (SRM). Analogously to s-FAs, *regular expressions* can also be defined *modulo* an alphabet theory $\mathcal{A}$, instead of using a finite alphabet. One can develop a corresponding theory of *symbolic derivatives* of such symbolic regular expressions. Symbolic derivatives enable lazy unfolding and on-the-fly creation of s-FAs that leads to a new class of more predictable matching algorithms that avoid backtracking. Such algorithms and a tool called *SRM*[32] have been developed at Microsoft Research and are being deployed daily in Azure for scanning credentials and other sensitive content in cloud service software. The input to SRM is a .NET regular expression over a restricted set of features. SRM's linear matching complexity has helped avoid unpredictable performance in the built-in .NET regular expression engine that was susceptible to catastrophic backtracking on files with long lines, such as minified JavaScript and SQL server seeding files.

Open Problems and Future Directions

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

Theoretical directions. *Adapting DFA algorithms to s-FAs.* Several algorithms for finite automata are based on efficient data structures that take advantage of the alphabet being finite. For example, Hopcroft's and Karp's algorithm for DFA minimization iterates over the alphabet to refine state partitions through splitting. This iteration can be avoided in s-FAs using satisfiability checks on certain carefully crafted predicates.[9] Paige-Tarjan's algorithm for computing forward bisimulations of NFAs is similar to Hopcroft's algorithm.[30] The algorithm can compute the partition of forward-bisimilar states in time $O(km \log n)$. However, unlike Hopcroft's algorithm, Paige-Tarjan's algorithm is hard to adapt to the symbolic setting and the current adaptation has complexity $O(2^m \log n + 2^m f_{\text{sat}}(n\ell))$.[11] By contrast, the simpler $O(km^2)$ algorithm for forward bisimulations can be easily turned into a symbolic $O(m^2\ f_{\text{sat}}(\ell))$ algorithm.[11] Another example of this complexity of adaptation arises in checking equivalence of two unambiguous NFAs.[34]

The problem of learning symbolic automata has only received limited attention.[3, 15, 26, 27] Classical learning algorithms 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 finite number of queries to the oracle. 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.

Properties of new models. Some symbolic models are still not well understood because they do not have finite automata counterparts. In particular, s-EFAs[10] do not enjoy many good properties, but it is possible that they have practical subclasses—for example deterministic, unambiguous, etc.—with good properties.

A new model, called Symbolic Register Automata (s-RA),[7] combines the symbolic aspect of s-FA with the ability of comparing elements at different positions in a string—for example, equality. This model is strictly more expressive than its components and it enjoys the same decidability properties of its nonsymbolic counterpart[24]—for example, equivalence is decidable for deterministic s-RAs. s-RAs could be used to represent complex list properties and improve static-analysis procedures. However, their theoretical treatment is limited so far and an exciting avenue for those interested in symbolic automata.

Complexity and expressiveness. In classical automata theory, the complexities of the algorithms are given with respect to the number of states and transitions in the automaton. We discussed previously how the algorithm complexities in the symbolic setting depend on trade-offs made in the alphabet theory. Exactly understanding these trade-offs is an interesting research question.

New potential applications. *Extracting automata from recurrent neural networks.* Due to the widespread adoption of machine learning modes, there has been renewed interest in the problem of extracting "explanations" from opaque black-box models. The work that is most relevant to ours is that of using automata learning to extract finite automata from recurrent neural networks.[40] Existing works have used automata learning for extracting automata from very simple synthetic RNNs. In particular, these techniques can only handle small-valued features (2-4 values) and cannot handle complex real-valued feature sets. s-FAs can be used to address this limitation. In particular, recent work on learning s-FAs[3] could potentially be used for

extracting automata from complex real-valued neural networks.

SMT solving with sequences. SMT solvers[14] 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. Solvers that are able to handle strings typically use automata. Most solvers only handle strings over finite small alphabets and s-FAs have the potential to impact the way in which such solvers for SMT are built. Recently, some SMT solvers such as Z3 have started incorporating s-FAs in the context of supporting regular expressions in the sequence theory.[33]

Static analysis. Dalla Preda et al. recently investigated how to use s-FAs to model program binaries.[5] 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 reflective code—that is, code that can self-modify itself at runtime.

Argyros et al.[4] have used s-FA learning to extract models of Web applications and detecting inconsistencies among different applications of the same logical application. Extending this approach to the more powerful extensions of s-FAs could lead to the ability to detect complex bugs in various kinds of software.

Conclusion

Symbolic automata and their variants have proven to be a versatile and powerful model to reason about practical applications that were beyond the reach of finite-alphabet automata models. In this article, we summarized what theoretical results are known for symbolic models, described the numerous extensions of symbolic automata, and clarified why these models are different from their finite-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 classical algorithms for automata over finite alphabets to the symbolic setting? Can we use symbolic automata algorithms to design decision procedures for the SMT theory of sequences? C

References

1. Alur, R, D'Antoni, L., Raghothaman, M. DReX: A declarative language for efficiently evaluating regular string transformations. *ACM SIGPLAN Notices – POPL'15 50*, 1 (2015), 125–137.
2. Alur, R., Madhusudan, P. Adding nesting structure to words. In *Developments in Language* Theory, O.H. Ibarra and Z. Dang, eds. Springer, Berlin, Heidelberg, 2006, 1–13.
3. Argyros, G., D'Antoni, L. The learnability of symbolic automata. In *Computer Aided Verification*, H. Chockler and G. Weissenbacher, eds. Springer International Publishing, Cham, 2018, 427–445.
4. Argyros, G., Stais, I., Jana, S., Keromytis, A.D., Kiayias, A. SFADiff: Automated evasion attacks and fingerprinting using black-box differential automata learning. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria, October 24-28, 2016), 1690–1701.
5. Dalla Preda, M., Giacobazzi, R., Lakhotia, A., Mastroeni, I. Abstract symbolic automata: Mixed syntactic/semantic similarity analysis of executables. *ACM SIGPLAN Notices – POPL'15 50*, 1 (2015), 329–341.
6. D'Antoni, L., Alur, R. Symbolic visibly pushdown automata. In *CAV'14* (2014), Springer International Publishing, Cham, 209–225.
7. D'Antoni, L., Ferreira, T., Sammartino, M., Silva, A. Symbolic register automata. In *Computer Aided Verification*, I. Dillig and S. Tasiran, eds, Springer International Publishing, Cham, 2019, 3–21.
8. D'Antoni, L., Kincaid, Z., Wang, F. A Symbolic Decision Procedure for Symbolic Alternating Finite Automata, *Electron. Notes Theor. Comput. Sci.* 336 (2018), 79–99.
9. D'Antoni, L., Veanes, M. Minimization of symbolic automata. *ACM SIGPLAN Notices – POPL'14 49*, 1 (2014), 541–553.
10. D'Antoni, L., Veanes, M. Extended symbolic finite automata and transducers. *Formal Methods Syst. Des. 47*, 1 (Aug. 2015), 93–119.
11. D'Antoni, L., Veanes, M. Forward bisimulations for nondeterministic symbolic finite automata. In *TACAS'17, LNCS* (2017), Springer, Berlin, Heidelberg, 518–534.
12. D'Antoni, L., Veanes, M. The Power of Symbolic Automata and Transducers. In: *Computer Aided Verification. Computer Aided Verification. CAV. Lecture Notes in Computer Science*. R. Majumdar, V. Kunčak, eds. Springer, Cham, 2017, vol. 10426, 47–67.
13. D'Antoni, L., Veanes, M., Livshits, B., Molnar, D. Fast: A transducer-based language for tree manipulation. *ACM TOPLAS 38*, 1 (2015), 1–32.
14. de Moura, L., Bjørner, N. Satisfiability modulo theories: Introduction & applications. *Commun. ACM 54*, 9 (Sept. 2011), 69–77.
15. Drews, S., D'Antoni, L. Learning symbolic automata. In *TACAS*'17 (2017), 173–189.
16. Flanagan, C., Qadeer, S. Predicate abstraction for software verification. In *Predicate Abstraction for Software Verification* (2002), vol. 37, ACM, 191–202.
17. Griffiths, T. The unsolvability of the equivalence problem for Λ-free nondeterministic generalized machines. *J. ACM* 15, 1968, 409–413.
18. Holzmann, G.J. The model checker spin. *IEEE Trans. Softw. Eng.* 23, 5 (May 1997), 279–295.
19. Hooimeijer, P., Livshits, B., Molnar,D., Saxena, P., Veanes, M. Fast and precise sanitizer analysis with BEK. In *Proceedings of the 20th USENIX Conference on Security, SEC'11* (Berkeley, CA, USA, 2011), USENIX Association, San Francisco, CA.
20. Hopcroft, J. An *nlogn* algorithm for minimizing states in a finite automaton. In *Proceedings of International Symposium on Theory of Machines and Computations*, (Technion, Haifa, 1971), Z. Kohavi, ed., 189–196.
21. Hopcroft, J.E., Karp, R.M. A linear algorithm for testing equivalence of finite automata. Technical Report 114, Cornell University, Ithaca, NY, December 1971.
22. Hopcroft, J.E., Ullman, J.D. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, Reading, Mass, 1979.
23. Hu, Q., D'Antoni, L. Automatic program inversion using symbolic transducers. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2017), ACM, New York, NY, USA, 376–389.
24. Kaminski, M., Francez, N. Finite-memory automata. *TCS 134*, 2 (1994), 329–363.
25. Keil, M., Thiemann, P. Symbolic solving of extended regular expression inequalities. In *FSTTCS'14* (2014), LIPIcs, Dagstuhl, Germany, 175–186.
26. Maler, O., Mens, I.-E. A generic algorithm for learning symbolic automata from membership queries. *Logical Methods Comput. Sci. 11*, 3:13 (2014), 2015.
27. Maler, O., Mens, I.-E. Learning regular languages over large ordered alphabets. *Log. Methods Comput. Sci.* 11, 3(2015)
28. Mamouras, K., Raghotaman, M., Alur, R., Ives, Z.G., Khanna, S. StreamQRE: Modular specification and efficient evaluation of quantitative queries over streaming data. ACM, New York, NY, USA, 2017, 693–708.
29. Moore. E.F. Gedanken-experiments on sequential machines. *In: Automata Studies, Annals of Mathematics Studies*. C. E. Shannon, J. McCarthy, eds. Litho-Printed, Princeton University Press, Princeton, vol. 34 (1956), 129–153.
30. Paige, R., Tarjan, R.E. Three partition refinement algorithms. *SIAM J. Comput. 16*, 6 (1987), 973–989.
31. Saarikivi, O., Veanes, M., Mytkowicz, T., Musuvathi, M. Fusing effectful comprehensions. In *ACM SIGPLAN Notices – PLDI'17* (2017), ACM, New York, NY.
32. Saarikivi, O., Veanes, M., Wan, T., Xu, E. Symbolic regex matcher. In *TACAS'19, LNCS* (2019), Springer, Cham, Switzerland.
33. C. Stanford, M. Veanes, and N. Bjørner. Symbolic Boolean Derivatives for Efficiently Solving Extended Regular Expression Constraints. Technical Report MSR-TR-2020-25, Microsoft, August 2020.
34. Stearns, R.E., Hunt, H.B. On the equivalence and containment problems for unambiguous regular expressions, grammars, and automata. *SIAM J. Comput. 14*, 3, 598–611.
35. Vardi, M.Y. Linear-time model checking: Automata theory in practice. In *Implementation and Application of Automata, 12th International Conference, CIAA 2007* (Prague, Czech Republic, July 16-18, 2007), Revised Selected Papers, 5–10.
36. Veanes, M., de Halleux, P., Tillmann, N. Rex: symbolic regular expression explorer. In: *Third international conference on software testing, verification and validation* (2010), IEEE, Paris, 498–507.
37. Veanes, M., Hooimeijer, P., Livshits, B., Molnar, D., Bjørner, N. Symbolic finite state transducers: Algorithms and applications. *ACM SIGPLAN Notices – POPL'12 47*, 1 (2012), 137–150.
38. Veanes, M., Mytkowicz, T., Molnar, D., Livshits, B. Data-parallel string-manipulating programs. *ACM SIGPLAN Notices – POPL'15 50*, 1 (2015), 139–152.
39. Watson, B.W. Implementing and using finite automata toolkits. In *Extended Finite State Models of Language*. Cambridge University Press, Cambridge, England, 1999, 19–36.
40. Weiss, G., Goldberg, Y., Yahav, E. Extracting automata from recurrent neural networks using queries and counterexamples. In *Proceedings of the 35th International Conference on Machine Learning*. J. Dy and A. Krause, eds. Volume 80, PMLR, Stockholmsmassan, Stockholm Sweden, 10–15 Jul 2018, 5247–5256

Loris D'Antoni, University of Wisconsin-Madison, WI, USA.

Margus Veanes, Microsoft Research, Redmond, WA, USA.