Applications of Symbolic Finite Automata

Margus Veanes

Microsoft Research
margus@microsoft.com

Abstract. Symbolic automata theory lifts classical automata theory to rich alphabet theories. It does so by replacing an explicit alphabet with an alphabet described implicitly by a Boolean algebra. How does this lifting affect the basic algorithms that lay the foundation for modern automata theory and what is the incentive for doing this? We investigate these questions here. In our approach we use state-of-the-art constraint solving techniques for automata analysis that are both expressive and efficient, even for very large and infinite alphabets. We show how symbolic finite automata enable applications ranging from modern regex analysis to advanced web security analysis, that were out of reach with prior methods.

1 Introduction

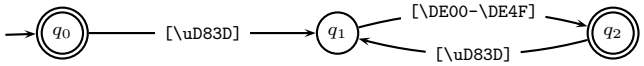
Classical automata theory makes two basic assumptions: there is a *finite state space*; and there is a *finite alphabet*. Here we challenge the second assumption by looking at how we can relax it while still maintaining all or most of the benefits of classical automata theory. One of the drawbacks of classical finite state automata is that they do not scale well for large alphabets. Although there are various techniques that address the scalability problem, such as, partial transition functions to avoid irrelevant or unused characters [3, 13], integer ranges for succinct representation of contiguous ranges of characters [1], binary decision diagrams for succinct representation of transition functions [7], as well as various extensions with registers such as register automata [10, 5] and extended finite automata [12]. Extensions with registers in general lead to infinite state systems or lack of closure properties. There is also research on register automata or automata over data words that focuses on their expressive power and decidability properties [11].

Our interest in this topic originates from the need to support regular expressions in the context of program analysis [17]. Regular expressions or regexes are stated over strings of basic Unicode characters. The runtime representation of characters in modern runtimes like JVM and .NET, as well as in scripting languages like JavaScript, uses the UTF16 encoding. From the point of view of regexes, the alphabet is the set of unsigned integers less than 2^{16} or in other words 16-bit bitvectors. For example the regex character class `[\u2639\u263A]` matches the symbols ☹ and ☺. Regexes do not directly support symbols in the supplementary Unicode planes (i.e. symbols that are formed from surrogate

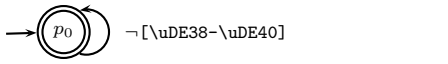
pairs and whose Unicode code point is $\geq 2^{16}$). For example, the surrogate pair `\uD83D\uDE0A` that also happens to encode a smiley symbol is treated as two separate characters by a regex, and the regex `^(\uD83D[\uDE00-\uDE4F])*$` matches a string that encodes a sequence of Unicode emoticons [2].[1]

Symbolic Finite Automata or SFAs were introduced, as an extension of classical finite state automata that allows transitions to be labeled with predicates defined in a separate alphabet algebra. The concept of automata with predicates instead of concrete symbols was first mentioned in [19] and was first discussed in [14] in the context of natural language processing. The alphabet theory in SFAs is assumed to be an *effective Boolean algebra*. The main intuition is that an SFA uses an alphabet as a plug-in through an API or interface. The only requirement is that the interface supports operations of a Boolean algebra.

To illustrate the role of the alphabet algebra consider the last regex example above. The predicate $\texttt{0xDE00} \leq x \wedge x \leq \texttt{0xDE4F}$ is an example of such a predicate in a character theory that uses linear arithmetic (modulo-2^{16}, or bitvector arithmetic) and one fixed variable x. We abbreviate it by `[\uDE00-\uDE4F]` using the standard character class notation of regexes. The following SFA is equivalent to the above regex of emoticons, say $M_{\text{emoticons}}$:



The regex character class `[\uDE38-\uDE40]` matches the set of low-surrogate halves of a "cat face" emoticon. Suppose we want to construct an SFA that accepts all strings of emoticons that contain no cat face emoticons. One way to do this is to construct the SFA $M_{\text{emoticons}} \times M_{\text{nocats}}$, where M_{nocats} is the SFA:



There are many fundamental questions about if and how classical algorithms and techniques can be lifted to SFAs. Some algorithms depend more on the alphabet than others. For example, union of SFAs uses only disjunctions of predicates over characters while intersection uses only conjunctions. Determinization on the other hand needs all Boolean operations. Satisfiability checking of predicates is used to avoid infeasible transitions. Some tradeoffs of the algorithms, when applied to string analysis, are studied in [9]. Minimization of SFAs is studied in [15]. It differs from the classical algorithms [4] with respect to how the alphabet is being used.

Here we discuss basic properties of SFAs, the role of the alphabet, and we describe different applications of SFAs, with a focus on the role of the symbolic alphabet. Two concrete applications are: *regex processing* and *security analysis of string sanitizers*.

[1] Emoticons are symbols with code points between `0x1F600` and `0x1F64F`. As an example, the surrogate pair `\uD83D\uDE0A` encodes the Unicode code point `0x1F60A` that is the code of a smiley symbol similar to ☺.

2 Effective Boolean algebras and SFAs

An *effective Boolean algebra* $\mathcal{A}$ has components $(\mathfrak{D}, \Psi, [\![\_]\!], \bot, \top, \vee, \wedge, \neg)$. $\mathfrak{D}$ is an r.e. (recursively enumerable) set of *domain elements*. Ψ is an r.e. set of *predicates* closed under the Boolean connectives and $\bot, \top \in \Psi$. The *denotation function* $[\![\_]\!] : \Psi \to 2^{\mathfrak{D}}$ is r.e. and is such that, $[\![\bot]\!] = \emptyset$, $[\![\top]\!] = \mathfrak{D}$, for all $\varphi, \psi \in \Psi$, $[\![\varphi \vee \psi]\!] = [\![\varphi]\!] \cup [\![\psi]\!]$, $[\![\varphi \wedge \psi]\!] = [\![\varphi]\!] \cap [\![\psi]\!]$, and $[\![\neg\varphi]\!] = \mathfrak{D} \setminus [\![\varphi]\!]$. For $\varphi \in \Psi$, we write $\mathit{IsSat}(\varphi)$ when $[\![\varphi]\!] \neq \emptyset$ and say that φ is *satisfiable*. $\mathcal{A}$ is *decidable* if *IsSat* is decidable.

The intuition is that such an algebra is represented programmatically as an API with corresponding methods implementing the Boolean operations and the denotation function. We are primarily going to use two such effective Boolean algebras in the examples, but the techniques in the paper are fully generic.

$\mathbf{2}^{\mathbf{bv}k}$ is the powerset algebra whose domain is the finite set $\textsc{bv}k$, for some $k > 0$, consisting of all nonnegative integers less than 2^k, or equivalently, all k-bit bit-vectors. A predicate is represented by a BDD of depth k.[2] The Boolean operations correspond directly to the BDD operations, $\bot$ is the BDD representing the empty set. The denotation $[\![\beta]\!]$ of a BDD β is the set of all integers n such that a binary representation of n corresponds to a solution of β.

SMT^{σ} is the decision procedure for a theory over some sort σ, say integers, such as the theory of integer linear arithmetic. This algebra can be implemented through an interface to an SMT solver. Ψ contains in this case the set of all formulas $\varphi(x)$ in that theory with one fixed free integer variable x. Here $[\![\varphi]\!]$ is the set of all integers n such that $\varphi(n)$ holds. For example, a formula $(x \bmod k) = 0$, say div_k, denotes the set of all numbers divisible by k. Then $\mathit{div}_2 \wedge \mathit{div}_3$ denotes the set of numbers divisible by six.

Extending a given alphabet domain with new characters in the concrete (classical) case is more or less trivial, while in the symbolic case it may not be possible at all or is difficult. We are using the following construct for alphabet extensions.

[2] The variable order of the BDD is the reverse bit order of the binary representation of a number, in particular, the most significant bit has the lowest ordinal.

Definition 1. The *disjoint union* $\mathcal{A}+\mathcal{B}$ of two effective Boolean algebras $\mathcal{A}$ and $\mathcal{B}$, is an effective Boolean algebra where,

$$\begin{aligned}
\mathfrak{D}_{\mathcal{A}+\mathcal{B}} &\stackrel{\text{def}}{=} (\mathfrak{D}_{\mathcal{A}} \times \{1\}) \cup (\mathfrak{D}_{\mathcal{B}} \times \{2\});\\
\Psi_{\mathcal{A}+\mathcal{B}} &\stackrel{\text{def}}{=} \Psi_{\mathcal{A}} \times \Psi_{\mathcal{B}};\\
[\![\langle \alpha, \beta\rangle]\!]_{\mathcal{A}+\mathcal{B}} &\stackrel{\text{def}}{=} ([\![\alpha]\!]_{\mathcal{A}} \times \{1\}) \cup ([\![\beta]\!]_{\mathcal{B}} \times \{2\})\\
\langle \alpha, \beta\rangle \vee_{\mathcal{A}+\mathcal{B}} \langle \alpha', \beta'\rangle &\stackrel{\text{def}}{=} \langle \alpha \vee_{\mathcal{A}} \alpha', \beta \vee_{\mathcal{B}} \beta'\rangle;\\
\langle \alpha, \beta\rangle \wedge_{\mathcal{A}+\mathcal{B}} \langle \alpha', \beta'\rangle &\stackrel{\text{def}}{=} \langle \alpha \wedge_{\mathcal{A}} \alpha', \beta \wedge_{\mathcal{B}} \beta'\rangle;\\
\neg_{\mathcal{A}+\mathcal{B}}\langle \alpha, \beta\rangle &\stackrel{\text{def}}{=} \langle \neg_{\mathcal{A}}\alpha, \neg_{\mathcal{B}}\beta\rangle;\\
\bot_{\mathcal{A}+\mathcal{B}} &\stackrel{\text{def}}{=} \langle \bot_{\mathcal{A}}, \bot_{\mathcal{B}}\rangle;\\
\top_{\mathcal{A}+\mathcal{B}} &\stackrel{\text{def}}{=} \langle \top_{\mathcal{A}}, \top_{\mathcal{B}}\rangle.
\end{aligned}$$

It is straightforward to prove by using distributive laws of intersection and union that the additional conditions of the denotation function hold for the above definition, i.e., that $\mathcal{A}+\mathcal{B}$ is indeed an effective Boolean algebra. In particular, consider conjunction (we drop the indices of the algebras as they are clear from the context)

$$\begin{aligned}
[\![\langle \alpha, \beta\rangle \wedge \langle \alpha', \beta'\rangle]\!] &= [\![\langle \alpha \wedge \alpha', \beta \wedge \beta'\rangle]\!]\\
&= [\![\alpha \wedge \alpha']\!] \times \{1\} \cup [\![\beta \wedge \beta']\!] \times \{2\}\\
&= ([\![\alpha]\!] \cap [\![\alpha']\!]) \times \{1\} \cup ([\![\beta]\!] \cap [\![\beta']\!]) \times \{2\}\\
&= ((\underbrace{[\![\alpha]\!] \times \{1\}}_{A}) \cap (\underbrace{[\![\alpha']\!] \times \{1\}}_{A'})) \cup ((\underbrace{[\![\beta]\!] \times \{2\}}_{B}) \cap (\underbrace{[\![\beta']\!] \times \{2\}}_{B'}))\\
&= (A \cap A') \cup (B \cap B') \cup (\underbrace{A \cap B'}_{=\emptyset}) \cup (\underbrace{B \cap A'}_{=\emptyset})\\
&= (A \cup B) \cap (A' \cup B')\\
&= [\![\langle \alpha, \beta\rangle]\!] \cap [\![\langle \alpha', \beta'\rangle]\!]
\end{aligned}$$

Another useful construct when dealing with effective Boolean algebras is *domain restriction*. In SFAs, domain restriction can be used to limit the alphabet to only those characters that matter.

Definition 2. The *domain restriction* of an effective Boolean algebra $\mathcal{A}$ with respect to a nonempty r.e. set $V \subseteq \mathfrak{D}_{\mathcal{A}}$, denoted $\mathcal{A}{\restriction}V$, is the same effective Boolean algebra as $\mathcal{A}$ except that $\mathfrak{D}_{\mathcal{A}{\restriction}V} \stackrel{\text{def}}{=} \mathfrak{D}_{\mathcal{A}} \cap V$ and $[\![\psi]\!]_{\mathcal{A}{\restriction}V} \stackrel{\text{def}}{=} [\![\psi]\!]_{\mathcal{A}} \cap V$.

It is easy to check that $\mathcal{A}{\restriction}V$ is well-defined. In particular, consider disjunction:

$$\begin{aligned}
[\![\psi \vee \varphi]\!]_{\mathcal{A}{\restriction}V} &= [\![\psi \vee \varphi]\!]_{\mathcal{A}} \cap V = ([\![\psi]\!]_{\mathcal{A}} \cup [\![\varphi]\!]_{\mathcal{A}}) \cap V = ([\![\psi]\!]_{\mathcal{A}} \cap V) \cup ([\![\varphi]\!]_{\mathcal{A}} \cap V)\\
&= [\![\psi]\!]_{\mathcal{A}{\restriction}V} \cup [\![\varphi]\!]_{\mathcal{A}{\restriction}V}
\end{aligned}$$

and complement:

$$\begin{aligned}
[\![\neg\psi]\!]_{\mathcal{A}{\restriction}V} &= [\![\neg\psi]\!]_{\mathcal{A}} \cap V = (\mathfrak{D}_{\mathcal{A}} \setminus [\![\psi]\!]_{\mathcal{A}}) \cap V = (\mathfrak{D}_{\mathcal{A}} \cap V) \setminus ([\![\psi]\!]_{\mathcal{A}} \cap V)\\
&= \mathfrak{D}_{\mathcal{A}{\restriction}V} \setminus [\![\psi]\!]_{\mathcal{A}{\restriction}V}
\end{aligned}$$

Definition 3. A *symbolic finite automaton* (*SFA*) M is a tuple $(\mathcal{A}, Q, q^0, F, \Delta)$ where $\mathcal{A}$ is an effective Boolean algebra, called the *alphabet*, 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 *moves* or *transitions*.

Elements of $\mathfrak{D}_{\mathcal{A}}$ are called *characters* and finite sequences of characters, elements of $\mathfrak{D}_{\mathcal{A}}^*$, are called *words*; ϵ denotes the empty word. A move $\rho = (p, \varphi, q) \in \Delta$ is also denoted by $p \xrightarrow{\varphi}_M q$ (or $p \xrightarrow{\varphi} q$ when M is clear) where p is the *source* state, denoted $Src(\rho)$, q is the *target* state, denoted $Tgt(\rho)$, and φ is the *guard* or *predicate* of the move, denoted $Grd(\rho)$. A move is *feasible* if its guard is satisfiable. Given a character $a \in \mathfrak{D}_{\mathcal{A}}$, an *a-move* of M is a move $p \xrightarrow{\varphi} q$ such that $a \in [\![\varphi]\!]$, also denoted $p \xrightarrow{a}_M q$ (or $p \xrightarrow{a} q$ when M is clear). In the following let $M = (\mathcal{A}, Q, q^0, F, \Delta)$ be an SFA.

Definition 4. A word $w = a_1 a_2 \cdots a_k \in \mathfrak{D}_{\mathcal{A}}^*$, is *accepted at state p of M*, denoted $w \in \mathscr{L}_p(M)$, if there exist $p_{i-1} \xrightarrow{a_i}_M p_i$ for $1 \leq i \leq k$ where $p_0 = p$ and $p_k \in F$. The *language accepted by M* is $\mathscr{L}(M) \stackrel{\text{def}}{=} \mathscr{L}_{q^0}(M)$.

For $q \in Q$, we use the definitions

$$\overrightarrow{\Delta}(q) \stackrel{\text{def}}{=} \{\rho \in \Delta \mid Src(\rho) = q\}, \quad \overleftarrow{\Delta}(q) \stackrel{\text{def}}{=} \{\rho \in \Delta \mid Tgt(\rho) = q\}.$$

The definitions are lifted to sets in the usual manner. The following terminology is used to characterize various key properties of M. A state p of M is called *partial* if there exists a character a such that there is no a-move from p.

- M is *deterministic*: for all $p \xrightarrow{\varphi} q, p \xrightarrow{\varphi'} q' \in \Delta$, if $IsSat(\varphi \wedge \varphi')$ then $q = q'$.
- M is *complete*: there are no partial states.
- M is *clean*: for all $p \xrightarrow{\varphi} q \in \Delta$, p is reachable from q^0 and $IsSat(\varphi)$,
- M is *normalized*: for all $p, q \in Q$, there is at most one move from p to q.
- M is *minimal*: M is deterministic, complete, clean, normalized, and for all $p, q \in Q$, $p = q$ if and only if $\mathscr{L}_p(M) = \mathscr{L}_q(M)$.[3]

Determinization of SFAs is always possible and is studied in [16]. Completion is straightforward: if M is not complete then add a new state $q_\emptyset$ and the self-loop $q_\emptyset \xrightarrow{\top} q_\emptyset$ and for each partial state q add the move $(q, \bigwedge_{\rho \in \overrightarrow{\Delta}(q)} \neg Grd(\rho), q_\emptyset)$. Observe that completion requires complementation of predicates.

Normalization is obvious: if there exist states p and q and two distinct transitions $p \xrightarrow{\varphi} q$ and $p \xrightarrow{\psi} q$ then replace these transitions with the single transition $p \xrightarrow{\varphi \vee \psi} q$. This does clearly not affect $\mathscr{L}_p(M)$ for any p.

Cleaning amounts to running standard forward reachability that keeps only reachable states, and eliminates infeasible moves. Observe that having infeasible moves $p \xrightarrow{\bot} q$ is semantically useless and may cause unnecessary state space explosion.

[3] It is sometimes convenient to define minimality over incomplete SFAs, in which case the *dead-end* state q ($q \neq q^0$ and $\mathscr{L}_q(M) = \emptyset$) is eliminated if it is present.

3 Applications

The development of the theory of symbolic automata has been driven by several concrete practical problems. Here we discuss two such applications. In each case we illustrate what kind of character theory we are working with, and focus on the benefits of the symbolic representation.

3.1 Regex processing

Practical applications of regular expressions or *regexes* is ubiquitous. What distinguishes practical regexes from schoolbook regular expressions (besides non-regular features that go beyond capabilities of finite state automata representations) are certain constructs that make them appealing (more succinct) than their classical counterparts such as *bounded quantifiers* and *character classes*.

The size of the alphabet is 2^{16} due to the widely adopted UTF16 standard of Unicode characters, e.g., as a somewhat unusual example, the regex `^[\uFF10-\uFF19]$` matches the set of digits in the so-called Wide Latin range of Unicode. We let the alphabet algebra be $\mathbf{2}^{\mathrm{BV16}}$. Let the BDD β_w^7 represent all ASCII word characters (letters, digits, and underscore) as the set of character codes $\{$`'0'`$, \ldots,$ `'9'`, `'A'`$, \ldots,$ `'Z'`, `'_'`, `'a'`$, \ldots,$ `'z'`$\}$. (We write `'0'` for the code 48, `'a'` for the code 97, etc.) Let also β_d^7 represents the set of all decimal digits $\{$`'0'`$, \ldots,$ `'9'`$\}$ and let $\beta_{\_}$ represent underscore $\{$`'_'`$\}$. By using the Boolean operations, e.g., $\beta_w^7 \wedge \neg(\beta_d^7 \vee \beta_{\_})$ represents the set of all upper- and lower-case ASCII letters. As a regex character class it is expressible as `[\w-[\d_\x7F-\uFFFF]]`.

Regexes are used in many different contexts. A common use of regexes is as a constraint language over strings for *checking* presence or absence of different patterns, e.g., for *security validation* of packet headers in network protocols. Another application, is the use of regexes for *generating* strings that match certain criteria, e.g., for *fuzz testing* applications that use regexes. A further application is *password generation* based on constraints given in form of regexes. Here is a scenario:[4]

1. Length is k and characters are in visible ASCII range: `^[\x21-\x7E]{`k`}$`
2. There are at least two letters: `[a-zA-Z].*[a-zA-Z]`
3. There is at least one digit: `\d`
4. There is at least one non-word character: `\W`

Consider SFAs for each case and build their product. The product is constructed by using depth-first search. Unsatisfiable predicates are eliminated so that the result is clean. Dead-end states are also eliminated. Random strings accepted by the automaton can be generated uniformly from its minimized or determinized form. Here the canonical structure of BDDs can be exploited to achieve uniformly random selection of characters from predicates.

[4] Recall the standard convention: a regex without the start-anchor `^` matches any prefix and a regex without the end-anchor `$` matches any suffix.

3.2 Sanitizer analysis

Sanitizers are string transformation routines (special purpose encoders) that are extensively used in web applications, in particular as the first line of defense against cross site scripting (XSS) attacks. There are at least three different string sanitizers involved in a single web page (CssEncoder, UrlEncoder, HtmlEncoder) that have very different semantics and sometimes use other basic encoders, e.g., UrlEncoder uses Utf8Encoder as the first step, while the raw input strings are in fact Utf16 encoded during runtime. A large class of sanitizers (including all the ones mentioned above) can be described and analyzed by using *symbolic finite state transducers* (SFTs) [8]. SFAs are used in that context for certain operations over SFTs, for example for checking domain equivalence of SFTs [18].

The character algebra here is modular integer linear arithmetic (or bitvector arithmetic of an SMT solver, the SMT solver used in our implementation is Z3 [6]). The main advantage of this choice is that it makes it possible to seamlessly combine the guards over characters with expressions over *yields* that are the *symbolic outputs* of SFT moves. A concrete example of a yield is the following transformation that takes a character and encodes it as a sequence of other characters:

$$f: \quad \lambda x.[\texttt{`\&'},\ \texttt{`\#'},\ (((x \div 10) \bmod 10) + 48),\ ((x \bmod 10) + 48),\ \texttt{`;'}]$$

In general, a yield denotes a function from an input character to an output word (of length that is independent of the input character). For example, a yield can be a function $\lambda x.[x, x]$ that duplicates the input character. Thus, an image of an SFTs is not necessarily SFA-recognizable, which is unlike the classical case where the image of a finite state transducer is always regular. In the above example, for example $f(\texttt{`a'})$ is the sequence [`&', `#', `9', `7', `;'] (or the string `"a"`). A typical SFT move ρ looks like:

$$\rho: \quad q \xrightarrow{(\lambda x.0<x<32)/\lambda x.[\texttt{`\&'},\ \texttt{`\#'},\ (((x \div 10) \bmod 10)+48),\ ((x \bmod 10)+48),\ \texttt{`;'}]} q$$

that is an HtmlEncoder rule for encoding control characters in state q and remaining in that state. For analyzing say idempotence of an encoder with such rules, the encoder is composed with itself. As a result, this leads to more complex guards and outputs of the resulting composed SFT (SFTs are closed under such composition). Imagine for example composing the move ρ with itself, i.e., roughly speaking, feeding the five output characters as its inputs again five times in a row. Then the guard of the composed rule will have subconditions such as $0 < (((x \div 10) \bmod 10) + 48) < 32$ involving potentially nontrivial arithmetic operations. (In this particular case the guard of the composed move will be infeasible.) One task of idempotence checking is *domain equivalence* of SFTs that reduces to *language equivalence* of SFAs whose guards now involve arithmetic operations of the above kind. Domain equivalence of SFTs essentially means that they accept/reject the same input sequences. Note that not all inputs sequences are valid. Perhaps a bit surprising, but even raw input strings may have misplaced characters (e.g. singleton occurrences of surrogates), assuming the standard Utf16 encoding of characters.

References

1. BRICS finite state automata utilities. http://www.brics.dk/automaton/.
2. Emoticons, Unicode standard, v. 6.2. http://unicode.org/charts/PDF/U1F600.pdf.
3. M.-P. Béal and M. Crochemore. Minimizing incomplete automata. In *Finite-State Methods and Natural Language Processing, 7th International Workshop*, pages 9–16, 2008.
4. J. Berstel, L. Boasson, O. Carton, and I. Fagnot. Minimization of automata. To appear in Handbook of Automata, 2011.
5. M. Bojanczyk, A. Muscholl, T. Schwentick, L. Segoufin, and C. David. Two-variable logic on words with data. In *LICS*, pages 7–16, 2006.
6. L. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *TACAS'08*, LNCS. Springer, 2008.
7. J. Henriksen, J. Jensen, M. Jørgensen, N. Klarlund, B. Paige, T. Rauhe, and A. Sandholm. Mona: Monadic second-order logic in practice. In *TACAS '95*, volume 1019 of *LNCS*. Springer, 1995.
8. P. Hooimeijer, B. Livshits, D. Molnar, P. Saxena, and M. Veanes. Fast and precise sanitizer analysis with Bek. In *USENIX Security*, August 2011.
9. 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.
10. M. Kaminski and N. Francez. Finite-memory automata. *TCS*, 134(2):329–363, 1994.
11. L. Segoufin. Automata and logics for words and trees over an infinite alphabet. In *CSL*, pages 41–57, 2006.
12. R. Smith, C. Estan, S. Jha, and S. Kong. Deflating the big bang: fast and scalable deep packet inspection with extended finite automata. SIGCOMM '08, pages 207–218. ACM, 2008.
13. A. Valmari and P. Lehtinen. Efficient minimization of DFAs with partial transition functions. In S. Albers and P. Weil, editors, *25th International Symposium on Theoretical Aspects of Computer Science (STACS 2008)*, pages 645–656, Dagstuhl, 2008.
14. G. van Noord and D. Gerdemann. Finite state transducers with predicates and identities. *Grammars*, 4(3):263–286, 2001.
15. M. Veanes. Minimization of symbolic automata. Technical Report MSR-TR-2013-48, Microsoft Research, 2013.
16. M. Veanes, N. Bjørner, and L. de Moura. Symbolic automata constraint solving. In C. Fermüller and A. Voronkov, editors, *LPAR-17*, volume 6397 of *LNCS/ARCoSS*, pages 640–654. Springer, 2010.
17. M. Veanes, P. de Halleux, and N. Tillmann. Rex: Symbolic Regular Expression Explorer. In *ICST'10*, pages 498–507. IEEE, 2010.
18. M. Veanes, P. Hooimeijer, B. Livshits, D. Molnar, and N. Bjørner. Symbolic finite state transducers: Algorithms and applications. In *POPL'12*, pages 137–150, 2012.
19. B. W. Watson. Implementing and using finite automata toolkits. In *Extended finite state models of language*, pages 19–36, New York, NY, USA, 1999. Cambridge University Press.