Minimization of Symbolic Automata

Loris D'Antoni
University of Pennsylvania
lorisdan@cis.upenn.edu

Margus Veanes
Microsoft Research
margus@microsoft.com

Abstract

Symbolic Automata extend classical automata by using symbolic alphabets instead of finite ones. Most of the classical automata algorithms rely on the alphabet being finite, and generalizing them to the symbolic setting is not a trivial task. In this paper we study the problem of minimizing symbolic automata. We formally define and prove the basic properties of minimality in the symbolic setting, and lift classical minimization algorithms (Huffman-Moore's and Hopcroft's algorithms) to symbolic automata. While Hopcroft's algorithm is the fastest known algorithm for DFA minimization, we show how, in the presence of symbolic alphabets, it can incur an exponential blowup. To address this issue, we introduce a new algorithm that fully benefits from the symbolic representation of the alphabet and does not suffer from the exponential blowup. We provide comprehensive performance evaluation of all the algorithms over large benchmarks and against existing state-of-the-art implementations. The experiments show how the new symbolic algorithm is faster than previous implementations.

Categories and Subject Descriptors F.2.2 [*Theory of Computation*]: Automata over infinite objects, Regular languages

Keywords Minimization, Symbolic Automata

1. Introduction

Classical automata theory builds on two basic assumptions: there is a *finite state space*; and there is a *finite alphabet*. The topic of this paper is along the line of work challenging the second assumption. *Symbolic finite automata* (SFAs) are finite state automata in which the alphabet is given by a *Boolean algebra* that may have an infinite domain, and transitions are labeled with predicates over such algebra. Symbolic automata originated from the intent to support regular expressions in the context of static and dynamic program analysis [43]. Lately, they were also used for supporting regular expressions (modulo label theories) in the context of modern logical inference engines [9, 42].

Most classical automata algorithms view the size k of the alphabet as a constant and use specialized data structures that are optimized for this view [7]. Therefore, it is not clear if or how such algorithms would work when the alphabet is infinite. Understanding how operations over the finite alphabet lift to the symbolic setting is a challenging task. Some classical automata constructions are extended to SFAs in [26]. For example, the *product* (intersection) $M_1 \times M_2$ of two symbolic automata M_1 and M_2 is computed by building product transitions of the form $\langle p_1, p_2\rangle \xrightarrow{\varphi_1\wedge\varphi_2} \langle q_1, q_2\rangle$ from transitions $p_1 \xrightarrow{\varphi_1} q_1$, $p_2 \xrightarrow{\varphi_2} q_2$, in M_1, M_2, where the guards φ_1 and φ_2 are composed using *conjunction* and pruned when *unsatisfiable*. The complexity of such constructions clearly depends on the complexity of checking satisfiability in the label theory. In this particular case, constructing the product has complexity $\mathcal{O}(f(\ell)m^2)$, where m is the number of transitions, $f(l)$ is the cost of checking satisfiability of predicates of size l in the label theory, and ℓ is the size of the biggest predicate in M_1 and M_2 .

This paper focuses on the problem of minimizing automata over symbolic alphabets and, to the best of our knowledge, it is the first paper that investigates this problem. Minimizing *deterministic finite automata* (DFAs) is one of the fundamental concepts in automata theory. It occurs in numerous areas, such as programming languages, text processing, computational linguistics, graphics, etc.

Before looking for an algorithm for minimizing SFAs we first need to answer a fundamental question: what does it mean for an SFA to be minimal? Intuitively, any two distinct states p and q of a minimal SFA must be *distinguishable*, where two states p and q are distinguishable if there exists an input sequence s that starting from p leads to a final (non-final) state and starting from q leads to a non-final (final) state. This notion is similar to DFA minimality.

The original algorithms for minimizing DFAs were given by Huffman [29], Moore [34], and Hopcroft [27]. Since, all such algorithms use iterations over the finite alphabet, they do not immediately extend to the symbolic setting. In the following paragraphs we briefly describe how we extended the classical algorithms to SFAs, and how we designed a new minimization algorithm that fully takes advantage of the symbolic representation of the alphabet.

Our first algorithm is called $Min^{\mathbf{M}}_{\text{SFA}}$ and takes inspiration from a reformulation of Moore's (Huffman's) algorithm described in [28]. The key idea from the algorithm in [28] is the following:

> if two states p and q are distinguishable, and there exists a character a, and transitions $\delta(a, p') = p$ and $\delta(a, q') = q$, then p' and q' are distinguishable.

This idea nicely translates to the symbolic setting as:

> if two states p and q are distinguishable, and there exist transitions $p' \xrightarrow{\varphi} p$ and $q' \xrightarrow{\psi} q$ such that $\varphi\wedge\psi$ is satisfiable, then p' and q' are distinguishable.

Starting with the fact that final and non-final states are distinguishable, we can use a fixpoint computation for grouping states into groups of indistinguishable states. This procedure uses a number of iterations that is quadratic in the number of states. Moreover, each iteration is linear in the number of transitions.

Unfortunately, $Min^{\mathbf{M}}_{\text{SFA}}$ does not scale in the case of a performance critical application described in Section 7.1, where SFAs

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
POPL '14, January 22 - 24 2014, San Diego, CA, USA.
Copyright © 2014 ACM 978-1-4503-2544-8/14/01...$15.00.
http://dx.doi.org/10.1145/2535838.2535849

	$Min^{\mathbf{M}}_{\text{SFA}}$	$Min^{\mathbf{H}}_{\text{SFA}}$	$Min^{\mathbf{N}}_{\text{SFA}}$
Satisfiability checking, $\wedge, \vee$	✓	✓	✓
Predicate negation		✓	✓
Minterm generation		✓	

Table 1. Summary of operations needed over a given label theory in the respective minimization algorithms.

end up having thousands of states, and the complexity of $Min^{\mathbf{M}}_{\text{SFA}}$ was not acceptable. To this end, we studied a generalization of Hopcrofts algorithm, called $Min^{\mathbf{H}}_{\text{SFA}}$. Hopcroft's algorithm is based on a technique called partition refinement of states, and, with worst case complexity $O(kn \log n)$ (n = number of states, and k = number of alphabet symbols), it is the most efficient algorithm for minimizing DFAs. The main idea behind $Min^{\mathbf{H}}_{\text{SFA}}$ is to subdivide all the labels in the SFA into non-overlapping predicates called minterms. Such minterms can then be seen as atomic characters in the sense of classical DFAs, allowing us to use the classical Hopcroft's algorithm over the finite set of minterms. Even though $Min^{\mathbf{H}}_{\text{SFA}}$ performs well in the aforementioned application, it suffers from another problem: in the worst case, the number of minterms can be exponential in the number of edges of the SFA. Moreover, our experiments showed that this factor can indeed be observed when considering more complex label theories, such as the theory of pairs over $string \times int$ (§ 6.4).

This leads to our main algorithmic contribution in the paper. We designed $Min^{\mathbf{N}}_{\text{SFA}}$, a new algorithm for SFA minimization which takes full advantage of the symbolic representation of the input alphabet. $Min^{\mathbf{N}}_{\text{SFA}}$ is inspired by the idea of refining the partition of the state space used by Hopcroft's algorithm, but does not require the pre-computation of minterms as in $Min^{\mathbf{H}}_{\text{SFA}}$. The key observation is that the set of relevant predicates can be computed locally, rather than using all the transitions of the SFA. While $Min^{\mathbf{N}}_{\text{SFA}}$ is similar to $Min^{\mathbf{H}}_{\text{SFA}}$ in terms of state complexity, it does not suffer from the exponential complexity related to the minterm computation. In fact, in all our experiments, both $Min^{\mathbf{H}}_{\text{SFA}}$ and $Min^{\mathbf{M}}_{\text{SFA}}$ are outperformed by $Min^{\mathbf{N}}_{\text{SFA}}$.

Table 1 presents a summary of the minimization algorithms, illustrating their dependency on operations over the label theory.

We compared the performance of the three algorithms using: 1) the benchmark of randomly generated DFAs presented in [3], 2) the SFAs generated by common regular expressions taken from the web, 3) a set of SFAs aimed at showing the exponential minterm explosion, 4) a randomly generated set of SFAs over a complex alphabet theory, and 5) the SFAs generated during the transformation from Monadic Second Order logic to DFAs [38]. In experiments 2 and 3 we also compared the performance of our implementation of $Min^{\mathbf{H}}_{\text{SFA}}$ against the implementation of Hopcroft's algorithm in [3] and in the `brics.automaton` [1] library, and we observed similar performance characteristics that validated our implementation. In the fifth experiment we compared our results against Mona [22], the state of the art tool for deciding Monadic Second Order logic.

Contributions. In summary, our contributions are:

- a formal study of the notion of minimality of SFAs (§2);
- two algorithms for minimizing SFAs based on classical DFA algorithms (§3 and §4);
- a completely new algorithm for minimizing SFAs together with a proof of its correctness (§5);
- a comprehensive evaluation of the algorithms using a variety of different benchmarks (§6); and
- a description of several concrete applications of such minimization algorithms (§7).

2. Effective Boolean algebras and SFAs

We first formally define the notion of effective Boolean algebra and symbolic finite automata. Next, we develop a theory which explains what it means for a symbolic finite automata to be minimal.

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 $IsSat(\varphi)$ 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 the following two effective Boolean algebras in the examples, but the techniques in the paper are fully generic.

$\mathbf{2}^{\text{BV}k}$ is the powerset algebra whose domain is the finite set $\text{BV}k$, for some $k > 0$, consisting of all nonnegative integers smaller than 2^k, or equivalently, all k-bit bit-vectors. A predicate is represented by a BDD of depth k.[1] 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. For example, a formula $(x \bmod k) = 0$, say div_k, denotes the set of all numbers divisible by k. Then $div_2 \wedge div_3$ denotes the set of numbers divisible by six.

We can now define symbolic finite automata. Intuitively, a symbolic finite automaton is a finite automaton over a symbolic alphabet, where edge labels are replaced by predicates. In order to preserve the classical closure operations (intersection, complement, etc.), the predicates must form an effective Boolean algebra.

DEFINITION 1. 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*(ρ), q is the *target* state, denoted *Tgt*(ρ), and φ is the *guard* or *predicate* of the move, denoted *Grd*(ρ). 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 2. 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 $1 \leq i \leq k$, such that $p_0 = p$, and $p_k \in F$. The *language accepted by M* is $\mathscr{L}(M) \stackrel{\text{def}}{=} \mathscr{L}_{q^0}(M)$. ⊠

[1] 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.

Given a state $q \in Q$, we use the following definitions for the set transitions from and to q:

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

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 for which there exist no a-move with source state p.

- M is *deterministic*: for all $p \xrightarrow{\varphi} q, p \xrightarrow{\varphi'} q' \in \Delta$, if $\mathit{IsSat}(\varphi \wedge \varphi')$ 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 $\mathit{IsSat}(\varphi)$,
- 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 $\mathscr{L}_p(M) = \mathscr{L}_q(M)$.[2]

For the special case in which M is *deterministic and complete*, we denote the transition function using the function $\delta_M : \mathfrak{D}_\mathcal{A} \times Q \to Q$, such that for all $a \in \mathfrak{D}_\mathcal{A}$ and $p \in Q$, $\delta_M(a, p) \stackrel{\text{def}}{=} q$, where q is the state such that $p \xrightarrow{a}_M q$. Observe that, because of determinism, if $p \xrightarrow{\varphi_1}_M q_1$, $p \xrightarrow{\varphi_2}_M q_2$, and $a \in [\![\varphi_1 \wedge \varphi_2]\!]$, then $q_1 = q_2$. Moreover, due to completeness, there exists some q and φ such that $p \xrightarrow{\varphi}_M q$ and $a \in [\![\varphi]\!]$.

Determinization of SFAs is always possible and is studied in [42]. 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 $(q, \bigwedge_{\rho \in \overrightarrow{\Delta}(q)} \neg \mathit{Grd}(\rho), q_\emptyset)$. Observe that completion requires negation 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 not affect $\mathscr{L}_p(M)$ for any p.

We always assume that M is clean. Cleaning amounts to running a standard forward reachability algorithm that keeps only reachable states, and eliminates infeasible moves. Observe that infeasible moves $p \xrightarrow{\perp} q$ do not add expressiveness and might cause unnecessary state space explosion.

It is important to show that minimality of SFAs is in fact well-defined in the sense that minimal SFAs are unique up to renaming of states and equivalence of predicates. To do so we use the following construction.

Assume $M = (\mathcal{A}, Q, q^0, F, \Delta)$ to be deterministic and complete. Let $\Sigma_\mathcal{A}$ denote the *first-order language* that contains the *unary relation symbol* $\bar{F}$ and the *unary function symbol* $\bar{a}$ for each $a \in \mathfrak{D}_\mathcal{A}$. We define the $\Sigma_\mathcal{A}$*-structure of* M, denoted by $\mathfrak{M}$, to have the universe Q, and the interpretation function:

$$\bar{F}^{\mathfrak{M}} \stackrel{\text{def}}{=} F, \quad \forall \bar{a} \in \Sigma_\mathcal{A},\, p \in Q\, (\bar{a}^{\mathfrak{M}}(p) \stackrel{\text{def}}{=} \delta_M(a, p)).$$

Also, let

$$\begin{aligned} M(\epsilon) &\stackrel{\text{def}}{=} q^0, \\ M(w \cdot a) &\stackrel{\text{def}}{=} \delta_M(a, M(w)) \qquad \text{for } a \in \mathfrak{D}_\mathcal{A} \text{ and } w \in \mathfrak{D}_\mathcal{A}^*. \end{aligned}$$

In other words, $M(w)$ is the state reached in M for the word $w \in \mathfrak{D}_\mathcal{A}^*$. Recall that two Σ-structures are *isomorphic* if there exists a bijective mapping between their universes that preserves the meaning of all symbols in Σ.

THEOREM 1. *If M and N are minimal SFAs over the same alphabet $\mathcal{A}$ such that $\mathscr{L}(M) = \mathscr{L}(N)$, then $\mathfrak{M}$ and $\mathfrak{N}$ are isomorphic $\Sigma_\mathcal{A}$-structures.*

Proof: Assume M and N to be minimal SFAs over $\mathcal{A}$ such that $\mathscr{L}(M) = \mathscr{L}(N)$. We define $\imath : \mathfrak{M} \cong \mathfrak{N}$ as follows:

$$\forall w \in \mathfrak{D}_\mathcal{A}^* (\imath(M(w)) \stackrel{\text{def}}{=} N(w)).$$

To show that $\imath$ is well-defined as a function, observe first that all states of M correspond to some w because M is clean. Second, we prove that for all words v and w, if $M(v) = M(w)$ then $N(v) = N(w)$. Fix $v, w \in \mathfrak{D}_\mathcal{A}^*$ such that $M(v) = M(w)$. Then, for all $u \in \mathfrak{D}_\mathcal{A}^*$,

$$\begin{aligned} u \in \mathscr{L}_{N(v)}(N) &\Leftrightarrow v \cdot u \in \mathscr{L}(N) \\ &\overset{(\text{by } \mathscr{L}(M) = \mathscr{L}(N))}{\Leftrightarrow} v \cdot u \in \mathscr{L}(M) \\ &\Leftrightarrow u \in \mathscr{L}_{M(v)}(M) \\ &\overset{(\text{by } M(w) = M(v))}{\Leftrightarrow} u \in \mathscr{L}_{M(w)}(M) \\ &\Leftrightarrow w \cdot u \in \mathscr{L}(M) \\ &\overset{(\text{by } \mathscr{L}(M) = \mathscr{L}(N))}{\Leftrightarrow} w \cdot u \in \mathscr{L}(N) \\ &\Leftrightarrow u \in \mathscr{L}_{N(w)}(N) \end{aligned}$$

So, $\mathscr{L}_{N(v)}(N) = \mathscr{L}_{N(w)}(N)$, and thus $N(v) = N(w)$ by minimality of N. So, $\imath$ is well-defined as a function. By switching the roles of M and N we also get the opposite direction. Thus,

$$(*) \qquad \forall v, w \in \mathfrak{D}_\mathcal{A}^* (M(v) = M(w) \Leftrightarrow N(v) = N(w))$$

Next, we show that $\imath$ is an isomorphism.

First, we show that $\imath$ is bijective: $\imath$ is onto because N is clean, i.e., each state of N corresponds to $N(w)$ for some word w; $\imath$ is into because if $M(v) \neq M(w)$ then, by $(*)$, $\imath(M(v)) = N(v) \neq N(w) = \imath(M(w))$.

Finally, we show that $\imath$ is an embedding of $\mathfrak{M}$ into $\mathfrak{N}$, i.e., that $\imath$ preserves all the functions and the relations: for all $p \in Q_M$, $p \in \bar{F}^{\mathfrak{M}} \Leftrightarrow \imath(p) \in \bar{F}^{\mathfrak{N}}$, and for all $p \in Q_M$ and $a \in \mathfrak{D}_\mathcal{A}$, $\imath(\bar{a}^{\mathfrak{M}}(p)) = \bar{a}^{\mathfrak{N}}(\imath(p))$.

Let $p \in Q_M$. Let w be any word such that $p = M(w)$. Then

$$\begin{aligned} p \in \bar{F}^{\mathfrak{M}} &\Leftrightarrow M(w) \in \bar{F}^{\mathfrak{M}} \Leftrightarrow w \in \mathscr{L}(M) \Leftrightarrow w \in \mathscr{L}(N) \\ &\Leftrightarrow N(w) \in \bar{F}^{\mathfrak{N}} \Leftrightarrow \imath(M(w)) \in \bar{F}^{\mathfrak{N}} \Leftrightarrow \imath(p) \in \bar{F}^{\mathfrak{N}}. \end{aligned}$$

and, for any $a \in \mathfrak{D}_\mathcal{A}$,

$$\begin{aligned} \imath(\bar{a}^{\mathfrak{M}}(p)) &= \imath(\bar{a}^{\mathfrak{M}}(M(w))) = \imath(\delta_M(a, M(w))) \\ &= \imath(M(w \cdot a)) = N(w \cdot a) = \delta_N(a, N(w)) \\ &= \bar{a}^{\mathfrak{N}}(N(w)) = \bar{a}^{\mathfrak{N}}(\imath(M(w))) = \bar{a}^{\mathfrak{N}}(\imath(p)) \end{aligned}$$

Thus $\mathfrak{M}$ and $\mathfrak{N}$ are isomorphic. ⊠

The theorem implies that minimal SFAs are unique up to renaming of states and up to equivalence of predicates due to normalization.

DEFINITION 3. Two states $p, q \in Q$ are M*-equivalent*, $p \equiv_M q$, when $\mathscr{L}_p(M) = \mathscr{L}_q(M)$. ⊠

We have that $\equiv_M$ is an equivalence relation. If $\equiv$ is an equivalence relation over Q, then for $q \in Q$, $q_{/\equiv}$ denotes the equivalence class containing q, for $X \subseteq Q$, $X_{/\equiv}$ denotes $\{q_{/\equiv} \mid q \in X\}$, and $M_{/\equiv}$ denotes the SFA $M_{/\equiv} \stackrel{\text{def}}{=} (\mathcal{A}, Q_{/\equiv}, q^0_{/\equiv}, F_{/\equiv}, \Delta_{/\equiv})$ where:

$$\Delta_{/\equiv} \stackrel{\text{def}}{=} \{(p_{/\equiv}, \bigvee_{(p,\varphi,q) \in \Delta} \varphi, q_{/\equiv}) \mid p, q \in Q, \exists \varphi((p, \varphi, q) \in \Delta)\}$$

[2] 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.

Observe that $M_{/\equiv}$ is normalized by construction. We need the following theorem that shows that minimization of SFAs preserves their intended semantics.

THEOREM 2. *Let M be a clean, complete and deterministic SFA. Then $M_{/\equiv_M}$ is minimal and $\mathscr{L}(M) = \mathscr{L}(M_{/\equiv_M})$.*

Proof: Let $\equiv$ be $\equiv_M$. Clearly, $M_{/\equiv}$ is clean and complete because M is clean and complete. To show determinism, let $\mathbf{p} \xrightarrow{a}_{M_{/\equiv}} \mathbf{q}_1$, and $\mathbf{p} \xrightarrow{a}_{M_{/\equiv}} \mathbf{q}_2$. Take $p_1, p_2 \in \mathbf{p}$, $q_1 \in \mathbf{q}_1$ and $q_2 \in \mathbf{q}_2$ such that $p_1 \xrightarrow{a}_M q_1$ and $p_2 \xrightarrow{a}_M q_2$. Since $\mathscr{L}_{p_1}(M) = \mathscr{L}_{p_2}(M)$, and M is deterministic, it follows that $\mathscr{L}_{q_1}(M) = \mathscr{L}_{q_2}(M)$ i.e., $\mathbf{q}_1 = \mathbf{q}_2$. Thus,

$$(*) \qquad \forall a \in \mathfrak{D}_\mathcal{A}, p \in Q_M(\delta_{M_{/\equiv}}(a, p_{/\equiv}) = \delta_M(a,p)_{/\equiv}).$$

Minimality of $M_{/\equiv}$ follows from the definition. Next, we show by induction over the length of w that

$$(\star) \qquad \forall w \in \mathfrak{D}_\mathcal{A}^*(M(w)_{/\equiv} = M_{/\equiv}(w))$$

For $w = \epsilon$ we have that $M(\epsilon) = q_M^0$, $M_{/\equiv}(\epsilon) = q^0_{M_{/\equiv}}$, and $q^0_{M_{/\equiv}} = (q_M^0)_{/\equiv}$.
For $w = v \cdot a$, where $a \in \mathfrak{D}_\mathcal{A}$ and $v \in \mathfrak{D}_\mathcal{A}^*$, we have that

$$\begin{aligned} M(v\cdot a)_{/\equiv} &= \delta_M(a, M(v))_{/\equiv} \overset{(\text{by }(*))}{=} \delta_{M_{/\equiv}}(a, M(v)_{/\equiv}) \\ &\overset{(\text{by IH})}{=} \delta_{M_{/\equiv}}(a, M_{/\equiv}(v)) = M_{/\equiv}(v\cdot a). \end{aligned}$$

It follows that, for all words $w \in \mathfrak{D}_\mathcal{A}^*$, $w \in \mathscr{L}(M)$ iff $M(w) \in F_M$ iff $M(w)_{/\equiv} \in F_{M_{/\equiv}}$ iff (by $(\star)$) $M_{/\equiv}(w) \in F_{M_{/\equiv}}$ iff $w \in \mathscr{L}(M_{/\equiv})$. Thus $\mathscr{L}(M) = \mathscr{L}(M_{/\equiv})$. ⊠

Theorems 1 and 2 are classical theorems lifted to arbitrary (possibly infinite) alphabets. Theorem 2 implies that SFAs have equivalent minimal forms that, by Theorem 1, are unique up to relabeling of states, and modulo equivalence of predicates in $\mathcal{A}$. In particular, since for all Q and all equivalence relations E over Q, $|Q_{/E}| \leq |Q|$, each equivalent form has minimal number of states.

3. Moore's algorithm over symbolic alphabets

Moore's minimization algorithm [34] of DFAs (also due to Huffman [29]) is commonly known as the *standard* algorithm. Even though the classical version of Moore's algorithm depends on the alphabet being finite, the general idea can be lifted to SFAs as follows. Given an SFA M, initially, let D be the binary relation $(F \times F^c) \cup (F^c \times F)$, where F^c is $Q \setminus F$. Compute the fixpoint of D as follows: if $\langle p, q\rangle \in D$ and there exist moves (p', φ, p) and (q', ψ, q) where $\varphi \wedge \psi$ *is satisfiable*[3] then add $\langle p', q'\rangle$ to D. This process clearly terminates. Upon termination, $E = (Q \times Q) \setminus D$ is the equivalence relation $\equiv_M$, and the SFA $M_{/E}$ is therefore minimal. We refer to this algorithm as $Min^{\mathbf{M}}_{\text{SFA}}$. Observe that $Min^{\mathbf{M}}_{\text{SFA}}$ checks only *satisfiability of conjunctions* of conditions and does not depend on the full power of the alphabet algebra, in particular it does not require the ability to complement predicates (assuming the initial completion of M is viewed as a separate preprocessing step). This is in contrast with the generalization of Hopcroft's algorithm discussed in the next section.

Complexity. In the finite alphabet case Moore's algorithm can be implemented in $\mathcal{O}(kn\log\, n)$ (using the approach described in [10]), where n is the number of states of the DFA, and k the number of characters in the input alphabet. However, such implementation relies on the alphabet being finite.

[3] In a concrete implementation a dictionary can be used to maintain D similar to the case of DFAs [28, Section 3.4].

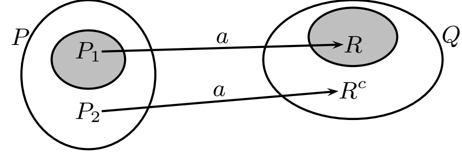


Figure 1. The idea behind an (a, R)-split of a part P.

Our implementation of the version of Moore's algorithm presented above has the following complexity. Given an SFA A, let n be the number of states of A, m the number of moves of A, and ℓ the size of the largest guard appearing in a transition of A (the biggest predicate). If the complexity of the alphabet theory for instances of size l is $f(l)$, then $Min^{\mathbf{M}}_{\text{SFA}}$ has complexity $\mathcal{O}(m^2 f(\ell))$, which is $\mathcal{O}(n^4 f(\ell))$ assuming normalized A where m is $\mathcal{O}(n^2)$.

4. Hopcroft's algorithm over symbolic alphabets

Hopcroft's algorithm [27] for minimizing DFAs is based on a technique called *partition refinement* of states. The symbolic version of Hopcroft's algorithm is given in Figure 3. Initially the set of states is partitioned into two sets: *final* states and *non-final* states, i.e., $\mathcal{P} = \{F, Q \setminus F\}$. Here we assume that the SFA M is complete and nontrivial, so that both F and $Q \setminus F$ are nonempty. The partition $\mathcal{P}$ induces the equivalence relation $\equiv_\mathcal{P}$ (or $\equiv$ when $\mathcal{P}$ is clear from the context) over Q, such that $Q_{/\equiv} = \mathcal{P}$. At a high level, $\mathcal{P}$ is refined as follows: suppose there exist parts $P, R \in \mathcal{P}$ and a character $a \in \mathfrak{D}_\mathcal{A}$ such that some a-move from P goes into R, and some a-move from P goes into R^c ($Q \setminus R$). Then the (a, R)*-split of* P is the set $\{P_1, P_2\}$, where P_1 (resp. P_2) is the set of all $p \in P$ from which there is an a-move into R (resp. R^c). P is then replaced in $\mathcal{P}$ by P_1 and P_2. See Figure 1. The (a, R)-split of P is well-defined by determinism and completeness of M. The following invariant is maintained by splitting.

LEMMA 1. *After each refinement step of $\mathcal{P}$, for all $p, q \in Q$, if $p \not\equiv q$ then $\mathscr{L}_p(M) \neq \mathscr{L}_q(M)$.*

Proof: Initially, for $p_1 \in F$: $\epsilon \in \mathscr{L}_{p_1}(M)$, and $p_2 \in Q \setminus F$: $\epsilon \notin \mathscr{L}_{p_2}(M)$. So, the statement holds for step $i = 0$. To see that the statement holds after step $i+1$, fix $P, R \in \mathcal{P}_i$ and $a \in \mathfrak{D}_\mathcal{A}$ such that (P_1, P_2) is the (a, R)-split of P. It is enough to show that, for all $p_1 \in P_1, p_2 \in P_2$, $\mathscr{L}_{p_1}(M) \neq \mathscr{L}_{p_2}(M)$. Consider $p_1 \xrightarrow{a} q_1$ and $p_2 \xrightarrow{a} q_2$. Since q_1 and q_2 belong to distinct parts of $\mathcal{P}_i$, by IH, there exists a word w such that $w \in \mathscr{L}_{q_1}(M)$ iff $w \notin \mathscr{L}_{q_2}(M)$. Moreover, by determinism of M, we have $a \cdot w \in \mathscr{L}_{p_j}(M)$ iff $w \in \mathscr{L}_{q_j}(M)$ for $j \in \{1, 2\}$. So, $a \cdot w \in \mathscr{L}_{p_1}(M)$ iff $a \cdot w \notin \mathscr{L}_{p_2}(M)$, and thus $\mathscr{L}_{p_1}(M) \neq \mathscr{L}_{p_2}(M)$. ⊠

Assuming a simple iterative refinement loop of $\mathcal{P}$, splitting is repeated until no further splits are possible, i.e., until $\mathcal{P}$ is *finest*, at which point the following property holds.

LEMMA 2. *Upon termination of refinement of $\mathcal{P}$, for all $p, q \in Q$, if $p \equiv q$ then $\mathscr{L}_p(M) = \mathscr{L}_q(M)$.*

Proof: By way of contradiction. Suppose

$(*)$ there exists a word x and states $p_1 \equiv p_2$,
such that $x \in \mathscr{L}_{p_1}(M) \Leftrightarrow x \notin \mathscr{L}_{p_2}(M)$.

Choose w to be a *shortest* such x. Since w cannot be ϵ (because $\{p_1, p_2\} \subseteq F$ or $\{p_1, p_2\} \subseteq Q \setminus F$), there are a and v such that $a \cdot v = w$, and moves $p_1 \xrightarrow{\varphi_1} q_1$ and $p_2 \xrightarrow{\varphi_2} q_2$ such that $a \in [\![\varphi_1]\!]$ and $a \in [\![\varphi_2]\!]$. The choice of q_1 and q_2 is unique for given a by determinism, thus $w \in \mathscr{L}_{p_i}(M) \Leftrightarrow v \in \mathscr{L}_{q_i}(M)$ for $i \in \{1, 2\}$. So, by $(*)$, $v \in \mathscr{L}_{q_1}(M) \Leftrightarrow v \notin \mathscr{L}_{q_2}(M)$.

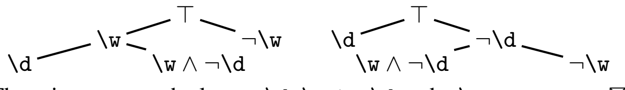
```
1  Minterms_A(ψ̄) ≝
2    tree := new Tree(⊤_A, null, null);
3    foreach (ψ in ψ̄) tree.Refine(ψ);
4      return Leaves(tree);  //return the set of all the leaf predicates
5  class Tree
6    Predicate φ; Tree left; Tree right;
7    Refine(ψ) ≝
8      if (IsSat_A(φ ∧_A ψ) and IsSat_A(φ ∧_A ¬_A ψ))
9        if (left = null)  //if the tree is a leaf then split φ into two parts
10         left := new Tree(φ ∧_A ψ, null, null);  //[[φ]] ∩ [[ψ]]
11         right := new Tree(φ ∧_A ¬_A ψ, null, null);  //[[φ]] \ [[ψ]]
12       else left.Refine(ψ); right.Refine(ψ);  //refine subtrees recursively
```

Figure 2. Minterm generation for $\bar{\psi} \subseteq \Psi_{\mathcal{A}}$ modulo $\mathcal{A}$.

We now have the following contradiction. There are two cases: 1) if $q_1 \not\equiv q_2$, say $q_1 \in R$ and $q_2 \in R^c$ for some part R, then $\mathcal{P}$ *is not finest* because we can split the part $P \in \mathcal{P}$ such that $p_1, p_2 \in P$ using the (a, R)-split of P. 2) if $q_1 \equiv q_2$ then, because $|v| < |w|$ and $(*)$ holds for $x = v$ and q_1, q_2 in place of p_1, p_2, it follows that w *is not the shortest* x *such that* $(*)$ *holds* Ψ. ⊠

In addition to the state partition refinement, in the symbolic case, we also use *predicate refinement*. Predicate refinement builds a set of *minterms*. A minterm is a minimal satisfiable Boolean combinations of all guards that occur in the SFA. The algorithm is shown in Figure 2. It uses a binary tree whose leaves define the partition. Initially the tree is the leaf $\top$. Each time a predicate ψ is used to refine the tree it may cause splitting of its leaves into finer predicates. The following example illustrates such minterm generation.

EXAMPLE 1. Consider the alphabet algebra $\mathbf{2}^{\mathrm{BV7}}$ (ASCII characters). We use standard regex notation for character classes. Suppose that the following two guards occur in the given SFA: `\w` ($[\![$`\w`$]\!] = [\![$`[a-zA-Z0-9_]`$]\!]$), and `\d` ($[\![$`\d`$]\!] = [\![$`[0-9]`$]\!]$). Then, the value of *tree* in $\mathit{Minterms}_{\mathbf{2}^{\mathrm{BV7}}}(\{\backslash\mathtt{w}, \backslash\mathtt{d}\})$ in line 4 in Figure 2 is either the first of the two trees below if `\w` is selected first in the loop of line 3, or else the second tree (note that $[\![\backslash\mathtt{d}]\!] \subsetneq [\![\backslash\mathtt{w}]\!]$ so $[\![\backslash\mathtt{d} \wedge \backslash\mathtt{w}]\!] = [\![\backslash\mathtt{d}]\!]$):



The minterms are the leaves `\d`, `\w` $\wedge\, \neg$`\d` and $\neg$`\w`. ⊠

Next, we analyze a property of the set of minterms. Given a predicate $\psi \in \Psi_{\mathcal{A}}$ and a state $p \in Q$, define:

$$\begin{aligned} \boldsymbol{\delta}(\psi, p) &\stackrel{\text{def}}{=} \{\mathit{Tgt}(\rho) \mid \rho \in \overrightarrow{\Delta}(p),\ \mathit{IsSat}(\mathit{Grd}(\rho) \wedge \psi)\} \\ \boldsymbol{\delta}^{-1}(\psi, p) &\stackrel{\text{def}}{=} \{\mathit{Src}(\rho) \mid \rho \in \overleftarrow{\Delta}(p),\ \mathit{IsSat}(\mathit{Grd}(\rho) \wedge \psi)\} \\ \boldsymbol{\delta}^{-1}(\psi, P) &\stackrel{\text{def}}{=} \bigcup_{p \in P} \boldsymbol{\delta}^{-1}(\psi, p) \quad (\text{for } P \subseteq Q) \end{aligned}$$

Let $\mathit{Minterms}(M) \stackrel{\text{def}}{=} \mathit{Minterms}_{\mathcal{A}}(\bigcup_{\rho \in \Delta} \mathit{Grd}(\rho))$. The following proposition implies that all characters that occur in one minterm are indistinguishable.

PROPOSITION 1. *Let M be deterministic and complete. For all $\psi \in \mathit{Minterms}(M)$ and $p \in Q$, $|\boldsymbol{\delta}(\psi, p)| = 1$.*

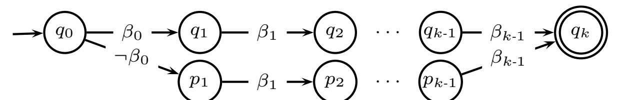
We can therefore treat $\boldsymbol{\delta}$ as a *function* from $\mathit{Minterms} \times Q$ to Q and reduce minimization of the SFA to minimization of the DFA with alphabet $\mathit{Minterms}$ and transition function $\boldsymbol{\delta}$. In particular, we can use Hopcroft's algorithm. We refer to the resulting algorithm by $\mathit{Min}^{\mathbf{H}}_{\mathrm{SFA}}$. Such an algorithm is shown in Figure 3.

```
1  Min^H_SFA(M = (A, Q, q^0, F, Δ)) ≝
2    P := {F, Q \ F};  //initial partition
3    W := {if (|F| ≤ |Q \ F|) then F else Q \ F};
4    Ψ := Minterms_A({Grd(ρ) | ρ ∈ Δ});  //compute the minterms
5    while (W ≠ ∅)  //iterate over unvisited parts
6      R := choose(W); W := W \ {R};
7      foreach (ψ in Ψ)  //iterate over all minterms
8        S := δ^{-1}(ψ, R);  //all states leading into R for given minterm
9        while (exists (P in P) where P ∩ S ≠ ∅ and P \ S ≠ ∅)
10         ⟨P, W⟩ := Split_{P,W}(P, P ∩ S, P \ S);  //split P
11   return M_{/≡P};
12   Split_{P,W}(P, P1, P2) ≝ ⟨P′, W′⟩ where
13   P′ = (P \ {P}) ∪ {P1, P2}  //refine P
14   W′ = if (P ∈ W) then (W \ {P}) ∪ {P1, P2}  //both parts
15          else W ∪ {if (|P1| ≤ |P2|) then P1 else P2}  //smaller part
```

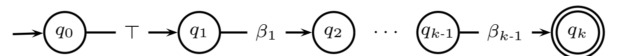
Figure 3. Hopcroft's minimization algorithm lifted to deterministic SFAs. M is assumed to be clean, complete, and nontrivial ($F \neq \emptyset$ and $Q \setminus F \neq \emptyset$).

Lifting the transition relation to be over *minterms* is a powerful hammer that can be used to adapt most classical automata algorithms to the symbolic setting. However, one drawback of minterm generation is that, in the worst case, the number of minterms of an SFA is exponential in the number of guards occurring in the SFA. The following example illustrates a worst case scenario in which, due to such a problem, $\mathit{Min}^{\mathbf{H}}_{\mathrm{SFA}}$ runs in exponential time.

EXAMPLE 2. Let the character domain be nonnegative integers $< 2^k$. Suppose $\beta_i(x)$ is a predicate that is true for x iff the i'th bit of the binary representation of x is 1, e.g. $\beta_3(8)$ is true and $\beta_3(7)$ is false. Predicate β_3 can be defined as $\neg((x \& 8) = 0)$, provided that, besides equality, the bitwise-and operator $\&$ is a built-in function symbol of $\Psi_{\mathcal{A}}$ (e.g., consider the bit-vector theory of an SMT solver). Similarly, we may also use the algebra $\mathbf{2}^{\mathrm{BV}k}$, where the size of the concrete BDD representation for β_i is linear in k, it has one node that is labeled by i and whose left child (case bit is 0) is false and whose right child (case bit is 1) is true. The point is that predicates are small (essentially constant) in size. Consider the following SFA M_k with such an alphabet $\mathcal{A}$.



Then $\mathit{Minterms}(M_k) = \mathit{Minterms}_{\mathcal{A}}(\{\neg\beta_i, \beta_i\}_{i<k}) = \{\hat{n}\}_{n<2^k}$ has 2^k elements, where $[\![\hat{n}]\!] = \{n\}$. For example, suppose $k = 3$, then $[\![\beta_2 \wedge \neg\beta_1 \wedge \beta_0]\!] = \{5\}$. The minimal automaton is

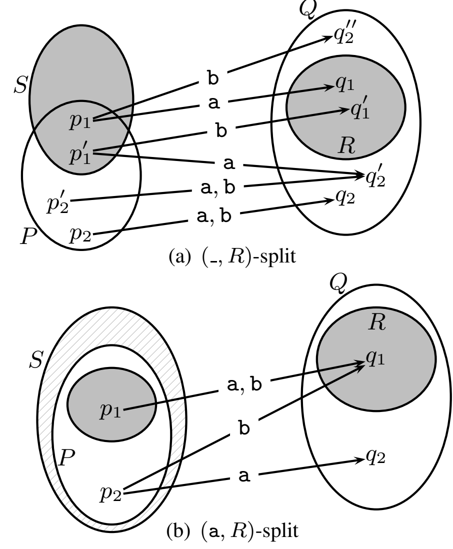


The dead-end state $q_\emptyset$ necessary for completion is implicit. ⊠

Complexity. In the finite alphabet case, Hopcroft's algorithm has complexity $\mathcal{O}(kn \log n)$, where n is the number of states of the DFA and k is the number of characters in the input alphabet [27], assuming k is treated as a constant. It is shown in [31] that if k is $\mathcal{O}(n)$ then the complexity of the algorithm presented in [27] is

1 $Min^{\mathbf{N}}_{\text{SFA}}(M = (\mathcal{A}, Q, q^0, F, \Delta)) \stackrel{\text{def}}{=}$
2 $\quad \mathcal{P} := \{F, Q \setminus F\}$; *//initial partition*
3 $\quad W := \{$**if** $(|F| \leq |Q \setminus F|)$ **then** F **else** $Q \setminus F\}$;
4 $\quad$**while** $(W \neq \emptyset)$ *//main loop*
5 $\quad\quad R :=$ **choose**(W); $W := W \setminus \{R\}$;
6 $\quad\quad S := \{Src(\tau) \mid \tau \in \overleftarrow{\Delta}(R)\}$; *//all states leading into R*
$\quad\quad$ *//$\Gamma(p)$ denotes the set of all characters leading from $p \in S$ into R*
7 $\quad\quad \Gamma := \{p \mapsto \bigvee_{\tau \in \overleftarrow{\Delta}(R), Src(\tau)=p} Grd(\tau) \mid p \in S\}$;
8 $\quad\quad$**while** (**exists** $(P$ **in** $\mathcal{P})$ **where** $P \cap S \neq \emptyset$ **and** $P \setminus S \neq \emptyset)$
9 $\quad\quad\quad \langle \mathcal{P}, W \rangle := Split_{\mathcal{P},W}(P, P \cap S, P \setminus S)$; *//$(_, R)$-split*
10 $\quad\quad$**while** (**exists** $(P$ **in** $\mathcal{P})$ **where** $P \cap S \neq \emptyset$ **and**
11 $\quad\quad\quad\quad$**exists** $(p_1, p_2$ **in** $P)$ **where** $IsSat(\neg(\Gamma(p_1) \Leftrightarrow \Gamma(p_2))))$
12 $\quad\quad\quad a :=$ **choose**$([\![\neg(\Gamma(p_1) \Leftrightarrow \Gamma(p_2))]\!])$;
13 $\quad\quad\quad P_1 := \{p \in P \mid a \in [\![\Gamma(p)]\!]\}$;
14 $\quad\quad\quad \langle \mathcal{P}, W \rangle := Split_{\mathcal{P},W}(P, P_1, P \setminus P_1)$; *//$(a, R)$-split*
15 $\quad$**return** $M_{/\equiv_{\mathcal{P}}}$;

Figure 4. Minimization of deterministic SFAs. M is assumed to be clean, complete and nontrivial. $Split_{\mathcal{P},W}$ is defined in Figure 3.



(a) $(_, R)$-split

(b) $(\mathtt{a}, R)$-split

Figure 5. Split cases of P in $Min^{\mathbf{N}}_{\text{SFA}}$. Suppose $\mathfrak{D}_{\mathcal{A}} = \{\mathtt{a}, \mathtt{b}\}$.

$O(n^3)$, but it is also shown that $\mathcal{O}(kn \log n)$ complexity can be maintained with a more careful implementation.

Given an SFA, let n be the number of states, let m be the number of moves, and let ℓ be the size of the largest label (guard) of a move. In a normalized SFA there are at most n^2 moves. Let μ be the number of minterms of the SFA; μ is bounded by 2^m. Each minterm is of the form $\phi_1 \wedge \ldots \wedge \phi_m$, where each ϕ_i is a guard or a negated guard, and thus minterms have size $O(m\ell)$. The computation of minterms is $\mathcal{O}(2^m f(m\ell))$ where f is the complexity to decide satisfiability of formulas of given size. The rest of the algorithm (after line 4) can be seen as a standard implementation of Hopcroft's algorithm, where the SFA is first transformed into a DFA with an alphabet of minterm identifiers where each SFA label has been replaced by identifiers of all relevant minterms. Then Ψ is viewed as a concrete alphabet (of such minterm identifiers). In the final result, the identifiers are mapped back to minterms and the resulting SFA is normalized. The overall complexity is then $\mathcal{O}(2^m f(m\ell) + 2^m n \log n)$.

5. Minimization without minterm generation

When inspecting the worst case complexity of $Min^{\mathbf{H}}_{\text{SFA}}$, the factor that stands out the most is the exponential blowup caused by the predicate refinement (minterm generation). In this section we investigate a new technique which is based on the symbolic representation of the alphabet, and that avoids minterm generation. Figure 4 shows $Min^{\mathbf{N}}_{\text{SFA}}$, a new minimization algorithm, that does not require predicate refinement.

The intuition behind $Min^{\mathbf{N}}_{\text{SFA}}$ is the following: when splitting a partition's part, it is not necessary to provide the exact witness that defines the split, instead it is enough to check if some witness (or witness set) exists. The main steps are the two inner while loops in Figure 4, they both refine the partition $\mathcal{P}$ with respect to R. In the first loop (lines 8–9) a part P is split into $P \cap S$ and $P \setminus S$ without using a fixed witness (see Figure 5(a) where $\{\{p_1, p'_1\}, \{p_2, p'_2\}\}$ is a $(_, R)$-split of P, but it is neither an $(\mathtt{a}, R)$-split nor a $(\mathtt{b}, R)$-split of P). The second loop (lines 10–14) splits P if there *exists* some a that produces an (a, R)-split of P. Such an element a must somehow distinguish at least two states p_1 and p_2 of P, (see Figure 5(b)), so that the split is indeed proper and guarantees progress.

The first loop (lines 8–9) is an optimization that can be omitted without sacrificing correctness, provided that for $p \notin S$ we let $\Gamma(p) \stackrel{\text{def}}{=} \bot$. The conditions $P \setminus S \neq \emptyset$ and $P \cap S \neq \emptyset$ together imply that there exist $p_1 \in P \cap S$ and $p_2 \in P \setminus S$ and thus $[\![\Gamma(p_1)]\!] \neq \emptyset$ but $[\![\Gamma(p_2)]\!] = \emptyset$, so $\neg(\Gamma(p_1) \Leftrightarrow \Gamma(p_2))$ is satisfiable.

The concrete implementation is shown in Figure 6. It differs from the abstract algorithm in that it computes local minterms, and does not compute any concrete witnesses (the element a is not computed) in the second loop. In the concrete implementation it is important to keep the first loop for the following reasons. One is efficiency, the first loop is cheaper. Second is simplicity, it is useful to work with Γ as a dictionary or array whose index set is S, and it is practical to assume that the invariant $P \subseteq S$ holds during the second loop.

Next, we formally prove the correctness of $Min^{\mathbf{N}}_{\text{SFA}}$. The proof provides more intuition on how the algorithm works. We then provide more details on how the concrete implementation works.

THEOREM 3. *$Min^{\mathbf{N}}_{\text{SFA}}(M)$ is minimal, and $\mathscr{L}(Min^{\mathbf{N}}_{\text{SFA}}(M)) = \mathscr{L}(M)$.*

Proof: We show first that the invariant of Lemma 1 holds. The invariant clearly holds initially. We show that it is preserved by each split (lines 8–9, and lines 10–14).

First, consider the first splitting loop in Figure 4. Fix $R \in \mathcal{P}$ and let $S = \{Src(\tau) \mid \tau \in \overleftarrow{\Delta}(R)\}$, and choose $P \in \mathcal{P}$ such that $P_1 = P \cap S \neq \emptyset$ and $P_2 = P \setminus S \neq \emptyset$. Fix $p_1 \in P_1$ and $p_2 \in P_2$. Then, there is a move $p_1 \xrightarrow{\varphi_1} q_1$ for some $q_1 \in R$. Let $a \in [\![\varphi_1]\!]$. Since $p_2 \notin S$ and M is complete, for some $q_2 \in R^c$ there exist a move $p_2 \xrightarrow{\varphi_2} q_2$ such that $a \in [\![\varphi_2]\!]$. The situation is illustrated in Figure 5(a). By using the invariant, $\mathscr{L}_{q_1}(M) \neq \mathscr{L}_{q_2}(M)$, there is a word w such that $w \in \mathscr{L}_{q_1}(M) \Leftrightarrow w \notin \mathscr{L}_{q_2}(M)$. Thus, by using the fact that M is deterministic, $a \cdot w \in \mathscr{L}_{p_1}(M) \Leftrightarrow a \cdot w \notin \mathscr{L}_{p_2}(M)$. Therefore, $\mathscr{L}_{p_1}(M) \neq \mathscr{L}_{p_2}(M)$.

Second, consider the second splitting loop. Fix P and $p_1, p_2 \in P$ that satisfy the loop condition. All parts in $\mathcal{P}$ that intersect with S must be subsets of S due to the first splitting loop, so, since M is clean, $P \subseteq S$, $IsSat(\Gamma(p_1))$ and $IsSat(\Gamma(p_2))$. The condi-

```
MinSFA(Automaton<S> fa)
{
  var fB  = new Block(fa.GetFinalStates());
  var nfB = new Block(fa.GetNonFinalStates());
  var blocks = new Dictionary<int, Block>();
  foreach (var q in fa.GetFinalStates()) blocks[q] = fB;
  foreach (var q in fa.GetNonFinalStates()) blocks[q] = nfB;
  var W = new BlockStack();
  if (nfB.Count < fB.Count) W.Push(nfB); else W.Push(fB);

  while (!W.IsEmpty) {
    var R = W.Pop();
    var G = ...  //Γ in Figure 4
    var S = G.Keys;
    var relevant = ...  //blocks intersecting with S
    foreach (var P in relevant){  //lines 8-9 in Figure 4
      var P1 = ...  //P ∩ S
      if (P1.Count < P.Count) {  //(_,R)-split of P
        foreach (var p in P1) { P.Remove(p); blocks[p] = P1;}
        if (W.Contains(P)) W.Push(P1);
        else if (P.Count <= P1.Count) W.Push(P);
        else W.Push(P1);
    }}
    bool iterate = true;
    while (iterate) {  //lines 10-14 in Figure 4
      iterate = false;
      relevant = ...  //blocks intersecting with S
      foreach (var P in relevant) {
        var P1 = new Block();
        var psi = G[P.Current];   //start with some element of P
        bool splitterFound = false;
        P1.Add(P.Current);

        while (P.MoveNext()) {
          var q = P.Current;
          var phi = G[q];
          if (splitterFound) {
            if (IsSat(psi & phi)) { P1.Add(q); psi = psi & phi;}
          } else {
            if (IsSat(psi & !phi)) {
              psi = psi & !phi;   //refine the local minterm
              splitterFound = true;
            } else {  //psi implies phi
              if (IsSat(phi & !psi)) {
                P1.Clear(); P1.Add(q);  //set P1 to {q}
                psi = phi & !psi;  //swap the local minterm
                splitterFound = true;
              } else P1.Add(q);  //psi is equivalent to phi
        }}}
        if (P1.Count < P.Count) {  //(a,R)-split of P for some a
          iterate = (iterate || (P.Count > 2));
          foreach (var p in P1) { P.Remove(p); blocks[p] = P1; }
          if (W.Contains(P)) W.Push(P1);
          else if (P.Count <= P1.Count) W.Push(P);
          else W.Push(P1);
  }}}}
  ...  //construct the result using blocks and normalize it
}
```

Figure 6. Concrete implementation of $\mathit{Min}^{\mathbf{N}}_{\text{SFA}}$.

tion $\mathit{IsSat}(\neg(\Gamma(p_1) \Leftrightarrow \Gamma(p_2)))$ means that either $\mathit{IsSat}(\Gamma(p_1) \wedge \neg\Gamma(p_2))$ or $\mathit{IsSat}(\Gamma(p_2) \wedge \neg\Gamma(p_1))$. Assume the former case and choose $a \in [\![\Gamma(p_1) \wedge \neg\Gamma(p_2)]\!]$. By definition of Γ, we know that there is a move $p_1 \xrightarrow{\varphi_1} q_1$ where $q_1 \in R$ such that $a \in [\![\varphi_1]\!]$. Moreover, since $a \notin [\![\Gamma(p_2)]\!]$, and $\Gamma(p_2)$ covers *all* the characters that lead from p_2 to R, there must be (by completeness and determinism of M) a move $p_2 \xrightarrow{\varphi_2} q_2$ where $q_2 \in R^c$ and $a \in [\![\varphi_2]\!]$. See Figure 5(b). It follows as above, by using $\mathscr{L}_{q_1}(M) \neq \mathscr{L}_{q_2}(M)$, that $\mathscr{L}_{p_1}(M) \neq \mathscr{L}_{p_2}(M)$. Since each step properly refines the partition, Lemma 1 follows.

We now show that Lemma 2 holds. The proof is by way of contradiction.

(*) Assume there exists a word x, a part $P \in \mathcal{P}$, and two states $p_1, p_2 \in P$ such that $x \in \mathscr{L}_{p_1}(M) \Leftrightarrow x \notin \mathscr{L}_{p_2}(M)$.

Let w be *shortest* such x. Since w cannot be ϵ, there exist a and v such that $w = a \cdot v$. So, there are, by determinism and completeness of M, unique q_1 and q_2 such that $p_1 \xrightarrow{a} q_1$ and $p_2 \xrightarrow{a} q_2$. It follows that $v \in \mathscr{L}_{q_1}(M) \Leftrightarrow v \notin \mathscr{L}_{q_2}(M)$. So $q_1 \not\equiv q_2$ or else v satisfies (*) and v is shorter that w.

Consider any fixed computation of $\mathit{Min}^{\mathbf{N}}_{\text{SFA}}$. It follows from the definition of $\mathit{Split}_{\mathcal{P},W}$, that W is always a subset of $\mathcal{P}$ and (due to the first condition of the update to W), if W ever contains a part containing a state q, then W will keep containing a part that contains q until such a part is removed from W in line 5.

Next, we show that the following W*-invariant* must hold at all times: for all $R \in W$, $q_1 \in R \Leftrightarrow q_2 \in R$. Let $\{\iota, \hat{\iota}\} = \{1, 2\}$. Suppose, by way of contradiction, that at some point in line 5 a part R is chosen from W such that $q_\iota \in R$, and $q_{\hat{\iota}} \notin R$. So, $p_\iota \in S$, with S as in line 6. We then have two following cases.

1. If $p_{\hat{\iota}} \notin S$, then p_ι and $p_{\hat{\iota}}$ are split apart in the first splitting loop. This contradicts the fact that $p_1 \equiv p_2$.
2. Assume $p_{\hat{\iota}} \in S$ and consider the second splitting loop. By choice of the character a above, we know that there exist moves $p_\iota \xrightarrow{\varphi_\iota} q_\iota$ and $p_{\hat{\iota}} \xrightarrow{\varphi_{\hat{\iota}}} q_{\hat{\iota}}$ where $a \in [\![\varphi_\iota]\!]$ and $a \in [\![\varphi_{\hat{\iota}}]\!]$. It follows that $a \notin [\![\Gamma(p_{\hat{\iota}})]\!]$ (because M is deterministic and $q_{\hat{\iota}} \notin R$) while $a \in [\![\Gamma(p_\iota)]\!]$. So $a \in [\![\Gamma(p_\iota)]\!] \setminus [\![\Gamma(p_{\hat{\iota}})]\!]$, or, in other words, $a \in [\![\Gamma(p_\iota) \wedge \neg\Gamma(p_{\hat{\iota}})]\!]$, and therefore $\mathit{IsSat}(\neg(\Gamma(p_\iota) \Leftrightarrow \Gamma(p_{\hat{\iota}})))$ holds. Consequently, p_ι and $p_{\hat{\iota}}$ end up in distinct parts upon termination of the second splitting loop. This again contradicts the fact that $p_1 \equiv p_2$.

So, initially $q_1 \in F \Leftrightarrow q_2 \in F$, or else the initial part of W violates the invariant. But now consider the point when the part containing both q_1 and q_2 is split into two parts containing q_1 and q_2 respectively. But at this point, at least one of those parts will be added to W by definition of $\mathit{Split}_{\mathcal{P},W}$. Thus, we have reached the desired contradiction, because the W-invariant is violated at that point.

We have shown that, upon termination of $\mathit{Min}^{\mathbf{N}}_{\text{SFA}}(M)$, $\equiv_{\mathcal{P}}$ coincides with $\equiv_M$. It follows from Theorem 2 that $M_{/\equiv_{\mathcal{P}}}$ is minimal and accepts $\mathscr{L}(M)$. ⊠

Implementation. A simplified version of our concrete C# implementation of $\mathit{Min}^{\mathbf{N}}_{\text{SFA}}$ is shown in Figure 6. Parts of a state partition are represented by mutable sets called *blocks*, i.e., objects of type `Block`, and states are represented by integers. Each block contains a `HashSet` of states. The search frontier W is maintained as a stack of blocks, and the partition $\mathcal{P}$ is an array of blocks, called `blocks`, that is indexed by states.

The first inner while loop (lines 8–9 in Figure 4) is implemented by iterating over all blocks `P` that intersect with `S`. The content of block `P1` becomes `P` ∩ `S`, while the content of block `P` is updated to `P` \ `S`. Observe that blocks are objects (pointers), thus if `W` contains `P`, after the (_,`R`)-split it will still contain `P` as well as the new block `P1`.

The second inner while loop (lines 10–14 in Figure 4) is implemented by en efficient encoding of the search for p_1 and p_2 in line 11 of Figure 4. Moreover, no concrete witness a is computed. Instead, a "local minterm" (called `psi` or ψ in Figure 6), is computed using Γ. This avoids the use of *model generation* that is more expensive than satisfiability checking (i.e. checking if there *exists* a model).[4] Thus, the implementation does not rely on model generation. Observe also that the second inner while loop relies on the fact that all remaining (`relevant`) blocks that intersect with S must be *contained* in S due to the first inner loop. The split of P

[4] Although $\mathit{IsSat}_{\mathcal{A}}(\varphi)$ is formally defined as $[\![\varphi]\!]_{\mathcal{A}} \neq \emptyset$, satisfiability checking is a more lightweight operation than actual model generation, in particular, in the context of SMT solvers.

then happens with respect to ψ, which, by construction, is a member of $Minterms(\{\Gamma(p) \mid p \in P\})$, so some member a of $[\![\psi]\!]$ would have produced the same split.

Relation to classical techniques. Implementation of Hopcroft's algorithm is discussed in detail in [7]. In the classical setting, the notions of blocks and (a, R)-splits are standard, the pair (R, a) is called a *splitter* in classical algorithms, where it plays a key role. The idea of splitting a block into two and keeping only the smaller block in the waiting set W is a core classical feature. In the case of *partial* DFAs, the algorithms are similar except that W must be initialized with both F and F^c [40].

In classical implementations the waiting set W consists of *splitters* rather than blocks, where characters have been *preselected*. The same is true for partial DFAs except that only those characters that are relevant for a given block are being used, which is beneficial for DFAs with sparse transition graphs. In the symbolic setting the character selection is indirect and achieved only through the satisfiability checks using local minterms during the second loop. The alphabet algebra may be infinite in the case of SFAs, this is not possible in the classical setting. As far as we know, the idea behind the first inner loop of $Min^{\mathbf{N}}_{\text{SFA}}$ and the notion of $(_, R)$-splits (recall Figure 5), say *free splitters* $(R, _)$, have not been studied in the classical setting.

Complexity. The complexity of $Min^{\mathbf{N}}_{\text{SFA}}$ depends on several factors, most importantly on the representation of predicates and the concrete representation of the partition refinement data structure, that is explained above.

First of all, observe that each $\Gamma(p)$ has size at most $\mathcal{O}(n\ell)$, and the total size of Γ is $\mathcal{O}(n^2\ell)$. Since the split operator always adds to W only the smallest partition, the outer loop is run at most $\log n$ times. The two internal loops have different running times:

- the first loop is run at most n times, with an internal complexity of $\mathcal{O}(n)$, due to the split operation, and
- the second loop is run at most n times, and if the complexity of the label theory for instances of size l is $f(l)$, the complexity of each iteration is at most $\mathcal{O}(nf(n\ell))$. This is due to the n internal iterations over the current part P. We also notice that each iteration calls the solver on a predicate of size at most $\mathcal{O}(n\ell)$.

We can conclude that $Min^{\mathbf{N}}_{\text{SFA}}$ has complexity $\mathcal{O}(n^2\log n \cdot f(n\ell))$. As we will observe in the next section, the quadratic behavior is not observed in practice.

6. Evaluation

We evaluate the performance of $Min^{\mathbf{N}}_{\text{SFA}}$, $Min^{\mathbf{H}}_{\text{SFA}}$, and $Min^{\mathbf{M}}_{\text{SFA}}$ with the following experiments:

1. We minimize the randomly generated DFAs from the benchmark presented in [3]. This experiment analyzes the performance in the presence of small finite alphabets and allows us to validate our implementation of $Min^{\mathbf{H}}_{\text{SFA}}$ against the results shown in [3];
2. We minimize the SFAs generated by common regular expressions taken from the web. This experiment measures performance in the case of typical character alphabets (Unicode);
3. We minimize the SFAs M_k from Example 2 to show the worst case exponential behavior of $Min^{\mathbf{H}}_{\text{SFA}}$. In this experiment we also compare against the `brics` library, a state-of-the-art automata library that uses character ranges as a symbolic representations of finite alphabets;

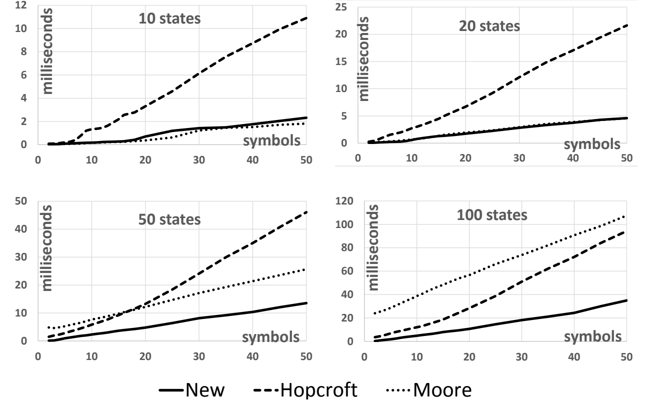


Figure 7. Running times on benchmark of randomly generated DFAs. The corresponding SFAs are over the theory of bitvectors and each set of input symbols is represented as a predicate using a binary decision diagram (BDD).

4. We minimize randomly generated SFAs over the theory of pairs of integers and strings. Here, we analyze the performance in the case in which the alphabet is infinite; and
5. We implemented the classical procedure for transforming Monadic Second Order (MSO) logic formulae into DFAs [38] and measure how the running time of such transformation is affected by different minimization algorithms. This last test aims at understanding the performance in the case of very large alphabets. We also compare our implementation against the tool Mona [22], the state of the art library for deciding MSO formulae.

All the experiments are run on a 64 bits Intel(R) Xeon(R) 3.60GHz processor with 8 GB of RAM memory.

6.1 Small randomly generated DFAs

In [3] the performance of several minimization algorithms are compared against a set of randomly generated DFAs. Such benchmark contains 5 million DFAs with number of states ranging between 5 and 1000, and alphabet sizes ranging between 2 and 50. The set of DFAs in [3] is uniformly generated at random, and therefore it offers good statistical coverage of the set of all possible DFAs with such number of states and alphabet sizes.

We run $Min^{\mathbf{N}}_{\text{SFA}}$, $Min^{\mathbf{M}}_{\text{SFA}}$ and $Min^{\mathbf{H}}_{\text{SFA}}$ on such set of DFAs. Figure 7 shows the results. For simplicity we only plot the results for DFAs with 10, 20, 50, and 100 states. Each plot contains the running time for each algorithm, where the x axis represents the number of symbols in the alphabet. In [3], Moore's algorithm is not considered and we weren't therefore able to validate the accuracy of our implementation of $Min^{\mathbf{M}}_{\text{SFA}}$. However, we were able to replicate the behavior of Hopcroft's algorithm shown [3] using $Min^{\mathbf{H}}_{\text{SFA}}$.

Figure 7 shows how the complexity of $Min^{\mathbf{M}}_{\text{SFA}}$ highly depends on the number of states, while the complexity of $Min^{\mathbf{H}}_{\text{SFA}}$ is mainly affected by the number of input alphabets. We can indeed see that, already for 100 states, $Min^{\mathbf{M}}_{\text{SFA}}$ performs worse than $Min^{\mathbf{H}}_{\text{SFA}}$. In this experiment, for most of the input DFAs the number of minterms was the same as the number of input symbols.

It is not a surprise that $Min^{\mathbf{N}}_{\text{SFA}}$ is much faster than both $Min^{\mathbf{M}}_{\text{SFA}}$ and $Min^{\mathbf{H}}_{\text{SFA}}$. Indeed, $Min^{\mathbf{N}}_{\text{SFA}}$'s performance seems to be resistant to both bigger state space and bigger alphabets. Moreover, for no single input $Min^{\mathbf{N}}_{\text{SFA}}$ is slower than $Min^{\mathbf{M}}_{\text{SFA}}$ or $Min^{\mathbf{H}}_{\text{SFA}}$.

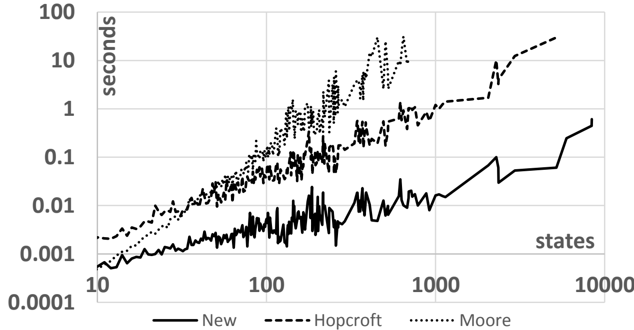


Figure 8. Running times on regexes from regexlib. Both axes are in log-scale.

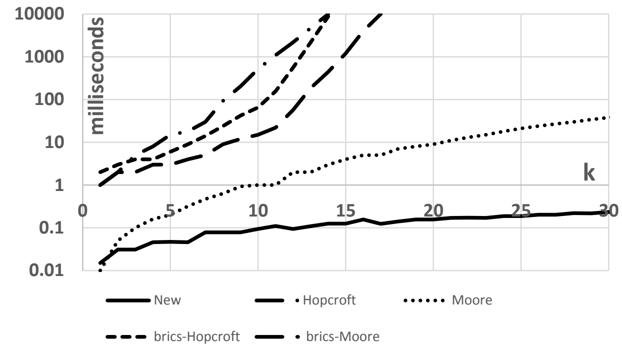


Figure 9. Running times on the M_k SFAs of Example 2. The y-axis is in log-scale.

6.2 Unicode regular expressions

Next, we compared the performance of different minimization algorithms over a sample set of 1850 SFAs over the alphabet $\mathbf{2}^{\mathrm{BV16}}$ (of Unicode characters) constructed from typical regexes (taken from a public web-site[5] of popular regexes). Figure 8 shows the running times. The figure clearly shows how $Min^{\mathbf{N}}_{\mathrm{SFA}}$ is faster than both $Min^{\mathbf{H}}_{\mathrm{SFA}}$ and $Min^{\mathbf{M}}_{\mathrm{SFA}}$ for every input instance. Moreover, it can be appreciated how for bigger state sizes (70-80 states) $Min^{\mathbf{H}}_{\mathrm{SFA}}$ starts outperforming $Min^{\mathbf{M}}_{\mathrm{SFA}}$.

In all cases, the number of minterms turned out to be *smaller* (by a factor between 2 and 3) than the total number of predicates, and the exponential blowup of minterms never occurred. The size of the generated SFAs ranged from 2 states to 15800 states with an average of 100 states. After the minimization each SFA was 32% smaller (number of states) than the original SFA.

The following is a typical regex from the sample set:

```
[NS]\d{1,}(\:[0-5]\d){2}.{0,1}\d{0,},[EW]\d{1,}(\:[0-5]\d){2}
```

The generated SFA uses 16 predicates (such as `\d`)[6] while there are only 7 minterms (such as `[:]`). For this regex, the determinized SFA has 47 states and the minimized SFA has 23 states.

Since the number of minterms does not blowup, is there a performance incentive in using $Min^{\mathbf{N}}_{\mathrm{SFA}}$ in this context? The answer is yes since the total time used to minimize all 1700 SFAs was 20 seconds when using $Min^{\mathbf{H}}_{\mathrm{SFA}}$, and only 0.8 seconds when using $Min^{\mathbf{N}}_{\mathrm{SFA}}$, thus showing a 24x speedup.

We also measured the performance of the minimization algorithm implemented in the `brics.automaton` library (version 1.11-8) that uses *symbolic integer ranges* to represent Unicode characters, but does not implement them as a Boolean algebra (since ranges are not closed under complement and union). We observed that the `brics` implementation of Hopcroft's algorithms is comparable to our implementation of $Min^{\mathbf{H}}_{\mathrm{SFA}}$, and for 95% of the regexes the running times of `brics` Hopcroft's algorithm and $Min^{\mathbf{H}}_{\mathrm{SFA}}$ were at most 5% apart. The trend of $Min^{\mathbf{H}}_{\mathrm{SFA}}$ in Figure 8 is very similar to that of the Hopcroft's implementation in the `brics` library. We decided not to include the latter in the plot for the following reasons: 1) `brics` is implemented in Java, while our implementations are all in C#, and 2) in `brics`, the code corresponding to the minterm computation is not part of the minimization algorithm, therefore the comparison would not be completely fair (especially for big instances).

[5] `http://regexlib.com/`

[6] We use standard regex character class notation for character predicates.

6.3 Alphabet corner cases

The next experiment shows the importance of avoiding the minterm generation used by $Min^{\mathbf{H}}_{\mathrm{SFA}}$. We consider the SFAs M_k from Example 2, for values of k ranging between 1 and 31, over the alphabet $\mathbf{2}^{\mathrm{BV32}}$ of 32-bit bit-vectors. Figure 9 shows the running times for $Min^{\mathbf{N}}_{\mathrm{SFA}}$, $Min^{\mathbf{H}}_{\mathrm{SFA}}$, and $Min^{\mathbf{M}}_{\mathrm{SFA}}$.

As expected, the performance of $Min^{\mathbf{H}}_{\mathrm{SFA}}$ degrades exponentially. Already for $k = 11$, minimizing the SFA using $Min^{\mathbf{H}}_{\mathrm{SFA}}$ took 1 second due to minterm generation, while both $Min^{\mathbf{M}}_{\mathrm{SFA}}$ and $Min^{\mathbf{N}}_{\mathrm{SFA}}$ took less than 1ms. For $Min^{\mathbf{M}}_{\mathrm{SFA}}$ the time increased from 1ms to .1sec with $k = 31$, while for $Min^{\mathbf{N}}_{\mathrm{SFA}}$ the time remained below 1ms for all the values of k.

Similarly to the previous experiment, we also compared the running time of the sample set against Hopcroft's and Moore's minimization algorithm in the `brics.autmaton` library (version 1.11-8). For this experiment, we decided to show the `brics` performance in order to appreciate how this particular example causes the number of ranges to grow exponentially, causing Hopcroft's algorithm to be very slow. It is interesting to notice that the `brics` implementation of Moore's algorithm is also slow. This is again due to the alphabet's blowup caused by the ranges algebra.

6.4 Complex theories

We compare the performance of $Min^{\mathbf{N}}_{\mathrm{SFA}}$, $Min^{\mathbf{H}}_{\mathrm{SFA}}$ and $Min^{\mathbf{M}}_{\mathrm{SFA}}$ over a sample set of 220 randomly generated SFAs over the theory of pairs over $string \times int$. The guards of each SFAs are conjunctions of simple predicates over strings and integers. In the theory of strings we only allow comparison against a constant string, while for integers we generate unary predicates containing $+, -, <, =$ and integer constants. The set of generated SFAs is created as follows. We first generated a set S of 10 SFAs with at most 10 states. For each SFA $a \in S$ we also compute the complement, and for every pair $a, b \in S$ we compute the union, intersection, and difference of a and b.

Figure 10 shows the running time of each algorithm for different numbers of states. We first observe how the performance of $Min^{\mathbf{H}}_{\mathrm{SFA}}$ quickly degrades when increasing the number of states. This is mainly due to the large number of minterms. Since the predicates are randomly generated, many overlaps are possible and the number of minterms grows quickly. Next, we can see how the performance of $Min^{\mathbf{M}}_{\mathrm{SFA}}$ is affected by the increasing number of states. Finally, $Min^{\mathbf{N}}_{\mathrm{SFA}}$ can quickly minimize SFAs with up to 96 states in less than 1.5 seconds. This experiment shows how $Min^{\mathbf{N}}_{\mathrm{SFA}}$ is not affected by complex theories, while $Min^{\mathbf{H}}_{\mathrm{SFA}}$ and $Min^{\mathbf{M}}_{\mathrm{SFA}}$ are both impractical in this setting.

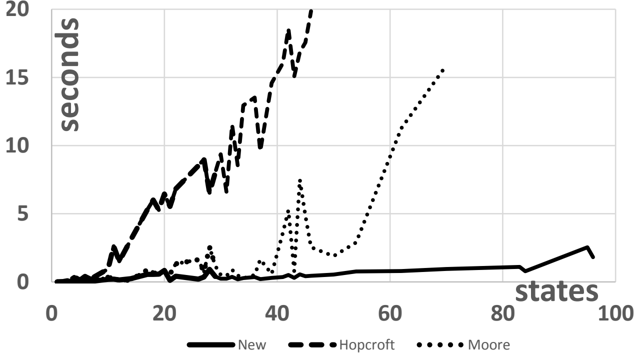


Figure 10. Running times for SFAs over the theory $String \times Int$ with 20 seconds timeout.

6.5 From Monadic Second-Order logic predicates to DFAs

The relation between regular languages and logic has been extensively investigated in the past [38]. Particularly, every regular language can be expressed as a formula in Monadic Second-Order logic (MSO) over strings, and vice versa. In the rest of the section we assume the alphabet to be a finite set Σ, say $\{a, b\}$. The following is an example of an MSO formula: $\phi = \exists x.\hat{a}(x)$ where $\hat{a}$ is a unary relation symbol. A string $s \in \Sigma^*$ is a model of ϕ, iff there exists a *position* x in the string with label a. Transforming MSO formulas to automata gives us an algorithm for deciding satisfiability of MSO formulas.

The procedure for converting an MSO formula into a DFA inductively transforms each sub-formula into the corresponding DFA and then combines such DFAs using classical regular language operations. The complexity arises from the presence of free variables, that range over positions. For example, in the formula $\hat{a}(x)$, x occurs free. In order to represent such a language, the alphabet Σ is extended with one extra bit to $\Sigma \times \{0, 1\}$. The string $\langle a, 0\rangle\langle b, 0\rangle\langle a, 1\rangle$ over the extended alphabet will then represent that the third position of the string aba is assigned to variable x. Following this intuition, every sub-formula ψ is compiled into a DFA over the alphabet $\Sigma \times \{0, 1\}^n$ where n is the number of quantified variables around ψ. For each existential quantification a projection is performed which leads to a non-deterministic automaton that must be determinized when a negation on it is performed. This means that each quantifier alternation might therefore lead to an exponential blowup, and in general the procedure has non-elementary complexity.

Despite the non-elementary complexity, practical algorithms that can translate non-trivial MSO formulas are presented in [22]. The tool implementing such algorithms is called Mona. The two key-features of such algorithms are:

1. determinizing and minimizing the intermediate DFA at every step in the transformation, and
2. using BDDs for representing the lifted bit-vector alphabets.

We implemented the same transformation using the BDD solver in our library, and compared the performance using different minimization algorithms.

Figure 11 shows the performance of the transformation for different MSO formulas. The running time for Mona are also shown. The four sub-figures depict the running time for the MSO to DFA transformation for the following formulas:

a) $\exists x_1, \ldots, x_k.\ x_1 < \ldots < x_k$, for k between 2 and 40,

b) $\exists x_1, \ldots, x_k.\ x_1 < \ldots < x_k \wedge a(x_1) \wedge \ldots \wedge a(x_k)$, for k between 2 and 40,

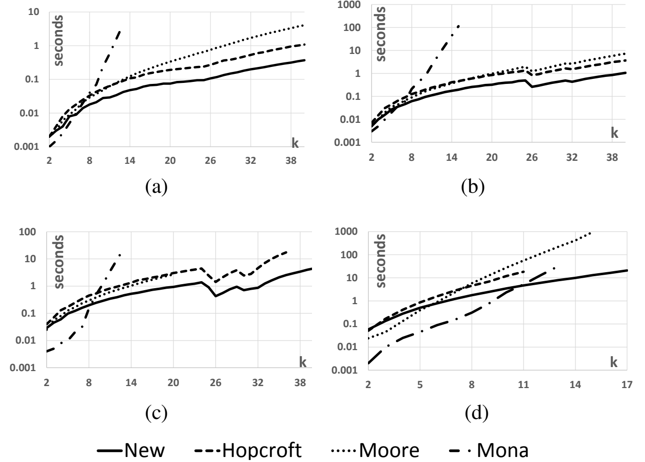


Figure 11. MSO to DFA running times. The y-axes are in log-scale.

c) $(\exists x_1, \ldots, x_k.\ x_1 < \ldots < x_k \wedge a(x_1) \wedge \ldots \wedge a(x_k)) \vee \exists y.c(y)$, for k between 2 and 40,

d) $\exists x_1, \ldots, x_k.\ f_k$ where $f_2 = (x_1 < x_2 \wedge a(x_1)) \vee c(x_1)$, and for $n > 2$, $f_n = f_{n-1} \wedge ((x_{n-1} < x_n \wedge a(x_{n-1})) \vee c(x_{n-1}))$, for k between 2 and 17.

All the values missing from the graphs are due to the algorithms running out of memory. The figure shows the following behaviors:

- for all of the four classes of formulas, the transformation based on $Min^{\mathbf{N}}_{\text{SFA}}$ is able to create the DFAs for higher values of k (number of nested variables) than those supported by Mona;
- for small instances Mona is faster than our implementation. However for higher values of k our implementation is faster, even when using $Min^{\mathbf{H}}_{\text{SFA}}$ or $Min^{\mathbf{M}}_{\text{SFA}}$;
- while Mona immediately shows an exponential behavior (very steep slope), we couldn't observe such trend in our implementation; and
- $Min^{\mathbf{N}}_{\text{SFA}}$ is faster than $Min^{\mathbf{H}}_{\text{SFA}}$ and $Min^{\mathbf{M}}_{\text{SFA}}$, and it is also less memory intensive (it runs out of memory for higher k).

Even though such formulas are only a small benchmark, they are quite representative of the kind of inputs that cause Mona's transformation to be impractical. We believe that the performance improvement of our implementation with respect to Mona is primarily due to a better use of the BDD solver, rather than due to the minimization algorithm.

7. Applications

The development of the theory of symbolic automata, including use of minimization, is motivated by several concrete practical problems. Here we discuss three such applications. In each case we illustrate what kind of character theory we are working with, the role of minimization, and focus on the benefits of the symbolic representation.

7.1 Regex processing

Practical applications of regular expressions or *regexes* is ubiquitous. Practical regexes differ from schoolbook regular expressions in the following aspects:

- they support, besides non-regular features that go beyond capabilities of finite state automata representations, constructs such

as *bounded quantifiers* and *character classes*, that make them more appealing (more succinct) than their classical counterparts, and

- the size of the alphabet is 2^{16} due to the widely adopted UTF16 standard of Unicode characters.

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 as the set of character codes $\{$`'0'`$,\dots,$`'9'`$,$`'A'`$,\dots,$`'Z'`$,$`'_'`$,$`'a'`$,\dots,$`'z'`$\}$. We write `'0'` for the code 48, `'a'` for the code 97, etc. Let also β_d^7 represent the set of all decimal digits $\{$`'0'`$,\dots,$`'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 lowercase 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. As another application consider the *password generation* problem based on constraints given in form of regexes. Here is a concrete set of constraints adopted in password generation:[7]

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`

The smallest feasible value for k is obviously 4. Consider SFAs A_1, A_2, A_3, and A_4 for each case and let A be their product. For $k = 4$ the minimized version of A has 12 states, and minimizing A with any of $Min^{\mathbf{N}}_{\mathrm{SFA}}$, $Min^{\mathbf{H}}_{\mathrm{SFA}}$, or $Min^{\mathbf{M}}_{\mathrm{SFA}}$ takes a few ms. When k is increased to 40,[8] the number of states increases to 444 and minimizing using $Min^{\mathbf{N}}_{\mathrm{SFA}}$ or $Min^{\mathbf{H}}_{\mathrm{SFA}}$ takes respectively 15 and 90 ms, while $Min^{\mathbf{M}}_{\mathrm{SFA}}$ becomes impractical, taking more than 43 sec (450x slower).

The minimal (canonical) form of A plays an important role here. Together with finiteness of the language accepted by A, minimality is used to guarantee a *uniform distribution* of the passwords generated from A. Uniformity is achieved by using the canonical form of the state graph together with the canonical form of the guards. While uniform sampling of $a \in [\![\varphi]\!]$ is not possible for arbitrary alphabet theories, it is a well-known feature of BDDs.

7.2 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, semantically very different, string sanitizers involved in a single web page: CssEncoder, UrlEncoder, and HtmlEncoder. A large class of sanitizers (including all the ones mentioned above) can be described and analyzed using *symbolic finite state transducers* (SFTs) [25]. SFAs are used in that context for certain operations over SFTs, for example for checking domain equivalence of SFTs [44]. Since several basic operations are quadratic in the number of states (product, union), minimization comes in handy as a tool for reducing the state space size,

In this setting we choose a different character theory. We use integer linear modular arithmetic (or bit-vector arithmetic of an SMT solver, in our case Z3 [19]) in place of BDDs. The main advantage of this choice is that it makes it possible to seamlessly combine the guards over characters with expressions over *yields*, i.e. 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 five other characters:

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

In general, a yield denotes a function from $\mathfrak{D}_{\mathcal{A}}$ to $\mathfrak{D}_{\mathcal{A}}^*$ (or to $\mathfrak{D}_{\mathcal{B}}$ when the output alphabet $\mathcal{B}$ is different from the input alphabet $\mathcal{A}$). In the yield above, for example, $f($`'a'`$)$ is the sequence $[$`'&'`$,$`'#'`$,$`'9'`$,$`'7'`$,$`';'`$]$ (or the string `"a"`). Given f as above, a typical SFT move ρ looks like:

$$\rho: \quad q \xrightarrow{(\lambda x.0<x<32)/\lambda x.f(x)} q$$

This specific rule happens to be an HtmlEncoder rule for encoding control characters in state q and remaining in that state. What is the connection with SFAs and the theory mentioned above? For analyzing the idempotence of an encoder with such rules, the encoder is sequentially composed with itself. As a result, this leads to an SFT with more complex guards and outputs (SFTs are closed under composition). When composing the move ρ with itself (i.e., roughly speaking, feeding the five output characters as its inputs again five times in a row), the guard of the composed rule will have sub-conditions such as $0 < (((x \div 10) \bmod 10) + 48) < 32$, which may involve nontrivial arithmetic operations (in this particular case the guard of the composed move will be infeasible). One of the operations performed during *idempotence* analysis is checking whether the original SFT and the composed one have the same *domain*. This reduces to *language equivalence* of SFAs for which the guards involve arithmetic operations of the above kind, that are not readily expressible using the earlier BDD based approach. Domain equivalence of two SFTs checks that both the SFTs accept/reject the same input sequences. Maintaining the SFAs minimal speeds up the equivalence check, and in general provides a better understanding of the structure of the domain language.

In general, the alphabet theory may be a Boolean combination of other decidable theories that are available for example in state-of-the-art SMT solvers. In the context of sanitizers, encoders, and decoders, the alphabet theory is a combination of lists, tuples, bit-vectors and integer linear arithmetic. Lists are used for example to represent composite characters or characters that represent "lookahead" [17]. We demonstrated in the evaluation section that, when the alphabet theory is complex the algorithm $Min^{\mathbf{N}}_{\mathrm{SFA}}$ outperforms both $Min^{\mathbf{H}}_{\mathrm{SFA}}$ and $Min^{\mathbf{M}}_{\mathrm{SFA}}$ by several orders of magnitude enabling analysis of bigger sanitizers/encoders.

7.3 Solving Monadic Second Order logic

We already anticipated in Section 6.5 how Monadic Second Order (MSO) logic predicates can be transformed into equivalent DFAs using a non-elementary algorithm. Such algorithm also provides a decision procedure for MSO.

We already discussed how several techniques have been introduced by the tool Mona [22] in order to make such transformation practical. Keeping the DFA minimal at any step in the transformation is one of the key techniques. We gave experimental evidence of the fact that the new algorithm presented in this paper, and in general the use of symbolic automata, can further move the barrier of solvable MSO formulas, in terms of both formula's size (number of nested quantifiers) and running times.

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

[8] Relevant values of k in the above scenario depend on the concrete context, but range between 4 and 128

8. Related work

DFA Minimization: The classical algorithms for DFA minimization have been studied and analyzed extensively in several different aspects. In particular, Moore's algorithm is studied in [10] where it is shown that, by a clever change of data structures, its complexity can be reduced from $O(kn^2)$ to $O(kn \log n)$. The bound $O(kn \log n)$ has been shown to be tight for Hopcroft's algorithm [6, 8]. Brzozowski [13] observed that a DFA can be minimized by reversing its transitions, then determinizing, then reversing its transitions again, and finally determinizing again. However, due to the determinization steps, this procedure is exponential and we decided not to consider it in this paper. Linear time algorithms have been studied for minimizing acyclic automata [32, 36] and automata containing only simple cycles [2]. The book chapter [7] provides an in-depth study of the state-of-the-art techniques for automata minimization, including the approaches mentioned above and several other ones. Watson [45] also provides an elegant classification of the classical minimization algorithms based on their structural properties.

In the case of DFAs, it matters whether the DFA to be minimized is partial (incomplete), because it may be useful to avoid completion of DFAs with sparse transition graphs. Minimization of partial DFAs is studied in [5, 39, 40]. In this setting the complexity of the algorithm depends on the number of transitions rather than on the size of the alphabet, however, in the case of DFAs these two quantities are generally related. In contrast, in a normalized SFA the number of transitions is independent of the alphabet size, and it is at most n^2 where n is the number of states. One concrete difference between the minimization algorithms of complete DFAs versus partial DFAs is that the initial value of the waiting set W (see Figure 3) for the partial case must contain both the sets F and F^c [40]. We believe that similar modifications may be applied to $Min^{\mathbf{N}}_{\text{SFA}}$, even though we expect that in the case of SFAs the benefit of using partiality might not be as visible. In fact, a complete SFA has at most n transitions more than a partial one. Moreover, in the case of SFAs, there are different ways for completing an SFA, e.g., one can effectively restrict the alphabet to only those characters that are mentioned in the trimmed SFA (SFA without dead-end states) prior to completion.

Different Notions of Minimization: A notion of incremental minimization is investigated in [47]. An incremental minimization algorithm can be halted at any point and produce a partially minimal DFA which is equivalent to the starting DFA, and has less or equal number of states. If the algorithm runs till completion, a minimal DFA is computed. In this paper we did not address incremental computation, however it would be interesting to identify variants of the presented algorithms with such a property. A similar idea, based on intermediate computation results, is used for a modular minimization algorithm in [14]. An algorithm for building a minimal DFA accepting a given finite set of strings is presented in [15]. The same paper investigates the problem of "maintaining" a minimal DFA when applying modification such as node, edge, or string deletion and insertion to an already minimal DFA. This class of problems is called dynamic minimization.

A parallel version of Moore's algorithm is presented in [37]. We are not aware of a parallel version of Hopcroft's algorithm. We leave as an open problem identifying parallel algorithms corresponding to $Min^{\mathbf{H}}_{\text{SFA}}$ and $Min^{\mathbf{N}}_{\text{SFA}}$.

A variant of minimization called hyper-minimization is investigated in [24]. Given an input DFA A, a DFA A' is hyper minimal with respect to A if it is the smallest DFA that differs from A only on a finite number of strings. In the case of symbolic automata, this definition doesn't extend naturally due to potentially infinite alphabet. A less restrictive notion called k-minimization is studied in [21]. Given an input DFA A, a DFA A' is k-minimal with respect to A if it is the smallest DFA that differs from A only on strings of length smaller or equal than k. This second restriction naturally extends to symbolic automata. A more general notion is that of minimization up to E-equivalence, where given a regular language E, the DFA A' is allowed to differ from A on a set of strings $L \subseteq E$ [23]. Extending the results of [21, 23] to symbolic automata is an interesting open research direction. Relationships between minimality and selection of final states are studied in [35] where *uniform* minimality is defined as minimality for any choice of final states.

Automata with Predicates: The concept of automata with predicates instead of concrete symbols was first mentioned in [46] and was first discussed in [41] in the context of natural language processing. A symbolic generalization of Moore's algorithm was first discussed in [43]. To the best of our knowledge, no other minimization algorithms have been studied for SFAs. The MONA implementation [22] provides decision procedures for monadic second-order logic, and it relies on a highly-optimized BDD-based representation for automata which has seen extensive engineering effort [30]. Therefore, the use of BDDs in the context of automata is not new, but is used here as an example of a Boolean algebra that seems particularly well suited for working with Unicode alphabets.

Minimization of Other Automata: The problem of automata minimization has been investigated in several other settings. A new approach for minimizing *nondeterministic* automata and nondeterministic Büchi automata has been studied in [33]. The problem of minimizing weighted automata (automata computing functions from strings to a semi-ring) is studied in [20]. Classical minimization algorithms are used in this setting, and we hope that the results shown in this paper can extend to such domain. The minimization problem has also been studied for timed automata [11] and register automata [16]. These models are able to represent infinite domains, but not arbitrary theories. An orthogonal direction is to extend the techniques presented in this paper to minimization of symbolic *tree* automata [18].

Other Applications: The problem of learning of symbolic transducers has recently been studied in [12]. Classical automata learning is based on Angluin's algorithm [4], which fundamentally relies on DFA minimality. An almost unexplored topic is whether such learning techniques can be extended to SFAs.

9. Conclusions

We presented three algorithms for minimizing Symbolic Finite Automata (SFAs). We first extended Moore's algorithm and Hopcroft's algorithm to the symbolic setting. We then show that, while in the classical setting Hopcroft's algorithm is the fastest known minimization algorithm, in the presence of symbolic alphabets, it might incur in an exponential blowup. To address this issue, we introduced a new minimization algorithm that fully benefits from the symbolic representation of the alphabet and does not suffer from the exponential blowup. The new algorithm is a refinement of Hopcroft's one in which splits are computed locally without having to consider the entire input alphabet. The new algorithm can also be adopted in the classical setting. Finally, we implemented all the algorithms and provided experimental evidence that the new minimization algorithm is the most efficient one in practice.

Acknowledgments. We thank Marco Almeida for helping us accessing his benchmark of randomly generated DFAs. Loris D'Antoni was supported by the NSF Expeditions in Computing award CCF 1138996, and this work was done as part of an internship at Microsoft Research. We also thank the anonymous reviewers for their insightful comments.

References

[1] BRICS finite state automata utilities. http://www.brics.dk/automaton/.

[2] J. Almeida and M. Zeitoun. Description and analysis of a bottom-up DFA minimization algorithm. *Information Processing Letters*, 107(2):52–59, 2008.

[3] M. Almeida, N. Moreira, and R. Reis. On the performance of automata minimization algorithms. Technical Report DCC-2007-03, University of Porto, 2007.

[4] D. Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, 1987.

[5] M.-P. Béal and M. Crochemore. Minimizing incomplete automata. In *Finite-State Methods and Natural Language Processing, 7th International Workshop*, pages 9–16, 2008.

[6] J. Berstel, L. Boasson, and O. Carton. Hopcroft's automaton minimization algorithm and Sturmian words. In *DMTCS'2008*, pages 355–366, 2008.

[7] J. Berstel, L. Boasson, O. Carton, and I. Fagnot. Minimization of automata. To appear in Handbook of Automata, 2011.

[8] J. Berstel and O. Carton. On the complexity of Hopcroft's state minimization algorithm. In *CIAA'2004*, volume 3317, pages 35–44, 2004.

[9] N. Bjørner, V. Ganesh, R. Michel, and M. Veanes. An SMT-LIB format for sequences and regular expressions. In P. Fontaine and A. Goel, editors, *SMT'12*, pages 76–86, 2012.

[10] N. Blum. An $0(n \log n)$ implementation of the standard method for minimizing n-state finite automata. *Information Processing Letters*, 57:65–69, 1996.

[11] M. Bojaczyk and S. Lasota. A machine-independent characterization of timed languages. In A. Czumaj, K. Mehlhorn, A. Pitts, and R. Wattenhofer, editors, *Automata, Languages, and Programming*, volume 7392 of *LNCS*, pages 92–103. Springer Berlin Heidelberg, 2012.

[12] M. Botinčan and D. Babić. Sigma*: symbolic learning of input-output specifications. In *POPL '13*, pages 443–456, New York, NY, USA, 2013. ACM.

[13] J. A. Brzozowski. Canonical regular expressions and minimal state graphs for definite events. In *Proc. Sympos. Math. Theory of Automata*, pages 529–561, New York, 1963.

[14] D. Bustan. Modular minimization of deterministic finite-state machines. In *In 6th International Workshop on Formal Methods for Industrial Critical Systems*, pages 163–178, 2001.

[15] R. C. Carrasco and M. L. Forcada. Incremental construction and maintenance of minimal finite-state automata. *Comput. Linguist.*, 28(2):207–216, June 2002.

[16] S. Cassel, B. Jonsson, F. Howar, and B. Steffen. A succinct canonical register automaton model for data domains with binary relations. In S. Chakraborty and M. Mukund, editors, *ATVA 2012*, volume 7561 of *LNCS*, pages 57–71. Springer, 2012.

[17] L. D'Antoni and M. Veanes. Equivalence of extended symbolic finite transducers. In N. Sharygina and H. Veith, editors, *CAV 2013*, volume 8044 of *LNCS*, pages 624–639. Springer, 2013.

[18] L. D'Antoni, M. Veanes, B. Livshits, and D. Molnar. Fast: A transducer-based language for tree manipulation. Technical Report MSR-TR-2012-123, Microsoft Research, November 2012.

[19] L. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *TACAS'08*, LNCS. Springer, 2008.

[20] M. Droste, W. Kuich, and H. Vogler. *Handbook of Weighted Automata*. Springer Publishing Company, Incorporated, 1st edition, 2009.

[21] P. Gawrychowski, A. Jež, and A. Maletti. On minimising automata with errors. In *MFCS'11*, pages 327–338, Berlin, Heidelberg, 2011. Springer.

[22] 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.

[23] M. Holzer and S. Jakobi. From equivalence to almost-equivalence, and beyond – minimizing automata with errors. In H.-C. Yen and O. Ibarra, editors, *Developments in Language Theory*, volume 7410 of *LNCS*, pages 190–201. Springer, 2012.

[24] M. Holzer and A. Maletti. An nlogn algorithm for hyper-minimizing a (minimized) deterministic automaton. *Theor. Comput. Sci.*, 411(38-39):3404–3413, 2010.

[25] P. Hooimeijer, B. Livshits, D. Molnar, P. Saxena, and M. Veanes. Fast and precise sanitizer analysis with Bek. In *USENIX Security*, August 2011.

[26] 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.

[27] 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, New York, 1971. Academic Press.

[28] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, 1979.

[29] D. Huffman. The synthesis of sequential switching circuits. *Journal of the Franklin Institute*, 257(3–4):161–190,275–303, 1954.

[30] N. Klarlund, A. Møller, and M. I. Schwartzbach. MONA implementation secrets. *International Journal of Foundations of Computer Science*, 13(4):571–586, 2002.

[31] T. Knuutila. Re-describing an algorithm by hopcroft. *Theor. Comput. Sci.*, 250(1-2):333–363, 2001.

[32] S. L. Krivol. Algorithms for minimization of finite acyclic automata and pattern matching in terms. *Cybernetics*, 27:324–331, 1991.

[33] R. Mayr and L. Clemente. Advanced automata minimization. In *POPL'13*, pages 63–74, 2013.

[34] E. F. Moore. Gedanken-experiments on sequential machines. *Automata studies, Annals of mathematics studies*, (34):129–153, 1956.

[35] A. Restivo and R. Vaglica. Some remarks on automata minimality. In G. Mauri and A. Leporati, editors, *DLT'2011*, volume 6795 of *LNCS*, pages 15–27. Springer, 2011.

[36] D. Revuz. Minimisation of acyclic deterministic automata in linear time. *Theoret. Comput. Sci.*, 92:181–189, 1992.

[37] A. Tewari, U. Srivastava, and P. Gupta. A parallel DFA minimization algorithm. In *HiPC 2002*, volume 2552 of *LNCS*, pages 34–40. Springer, 2002.

[38] W. Thomas. Languages, automata, and logic. In *Handbook of Formal Languages*, pages 389–455. Springer, 1996.

[39] A. Valmari. Fast brief practical DFA minimization. *Information Processing Letters*, 112:213–217, 2012.

[40] A. Valmari and P. Lehtinen. Efficient minimization of DFAs with partial transition functions. In S. Albers and P. Weil, editors, *STACS 2008*, pages 645–656, Dagstuhl, 2008.

[41] G. van Noord and D. Gerdemann. Finite state transducers with predicates and identities. *Grammars*, 4(3):263–286, 2001.

[42] 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.

[43] M. Veanes, P. de Halleux, and N. Tillmann. Rex: Symbolic regular expression explorer. In *ICST'10*, pages 498–507. IEEE, 2010.

[44] 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.

[45] B. W. Watson. A taxonomy of finite automata minimization algorithms. Computing Science Report 93/44, Eindhoven University of Technology, January 1995.

[46] 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.

[47] B. W. Watson and J. Daciuk. An efficient incremental DFA minimization algorithm. *Nat. Lang. Eng.*, 9(1):49–64, Mar. 2003.