A Symbolic Decision Procedure for Symbolic Alternating Finite Automata

Loris D'Antoni
University of Wisconsin-Madison
ldantoni@wisc.edu

Zachary Kincaid
Princeton University
zkincaid@cs.princeton.edu

Fang Wang
University of Wisconsin-Madison
fang64@wisc.edu

Abstract

We introduce Symbolic Alternating Finite Automata (S-AFA) as an expressive, succinct, and decidable model for describing sets of finite sequences over arbitrary alphabets. Boolean operations over S-AFAs have linear complexity, which is in sharp contrast with the quadratic cost of intersection and union for non-alternating symbolic automata. Due to this succinctness, emptiness and equivalence checking are PSPACE-hard.

We introduce an algorithm for checking the equivalence of two S-AFAs based on bisimulation up to congruence. This algorithm allows us to exploit the power of SAT and SMT solvers to efficiently search the state space of the S-AFAs. We evaluate our decision procedure on two verification and security applications: 1) checking satisfiability of linear temporal logic formulas over finite traces, and 2) checking equivalence of Boolean combinations of regular expressions. Our experiments show that our technique often outperforms existing techniques and it can be beneficial in both such applications.

1. Introduction

Programs that operate over sequences are ubiquitous and used for many different tasks, such as text processing [2], program monitoring [23], and deep packet inspection in networking [29]. Being able to efficiently reason about these programs is a crucial task and many techniques have been proposed to do so. However, most of these techniques share a common link: they use finite automata and the related decision procedures.

Due to this reason, recently there has been renewed interest in automata theory, especially in the fields of security and programming languages [4, 5, 11, 13, 26, 29]. Despite these improvements, existing automata-based decision procedures are not as advanced as other decision procedures such as SMT solvers yet. For example, if one wants to use classic automata techniques to check whether there exists a string s that is accepted by all regular expressions in a set $\{r_1, \dots, r_n\}$, she will incur in two problems.

- Regular expressions operate over large alphabets with thousands of characters and most existing automata formulations do not cope well with large alphabets.
- Intersecting the automata corresponding to the given regular expressions produces an automaton with number of states exponential in the number of regular expressions. Indeed, this problem is PSPACE-complete, but (as it happened in the world of SMT solvers) one might hope to find better solutions that rarely hit the worst-case complexity.

In this paper, we present a novel automata model together with a decision procedure to address both such limitations.

Symbolic finite automata The problem of handling large alphabets is not a new one. Recently a new model, called symbolic finite automata, has been proposed to address this limitation. Symbolic Finite Automata (S-FAS) [13, 33] are finite state automata in which the alphabet is given as a Boolean algebra that may operate over an infinite domain, and transitions are labeled with first-order predicates over the algebra. Although strictly more expressive than finite-state automata, S-FA are closed under Boolean operations and admit decidable equivalence, as long as it is decidable to check satisfiability of predicates in the alphabet algebra.

Symbolic alternating finite automata While S-FAS provide an elegant framework for abstracting away the alphabet structure, they have the same state complexities of classic finite automata. In particular, repeated S-FA intersections can result in S-FAS with exponentially many states. To solve this problem, we propose Symbolic Alternating Finite Automata (S-AFAs). S-AFAs add alternation to S-FAS by allowing transitions to contain Boolean formulas that describe the set of target states. For example, when an S-AFA is in state p and is reading a string $s = a_1 \dots a_n$, the transition

$$p \xrightarrow{[a-z]} q_1 \vee (q_2 \wedge q_3)$$

specifies that s is accepted from state p, if a_1 is a lower-case alphabetic character and either the string $a_2 \dots a_n$ is accepted from state q_1 or it is accepted from both q_2 and q_3. By adding alternation to S-FAS, S-AFAs obtain Boolean operations with linear complexity, which is in sharp contrast with the quadratic intersection and exponential complementation of S-FAS.

Equivalence using bisimulation The succinctness of S-AFAs comes at a cost: equivalence and emptiness are PSPACE-complete problems. In the case of S-FAS, emptiness has linear complexity while equivalence is also PSPACE-complete.

In this paper, we propose a symbolic decision procedure for checking equivalence (and emptiness) of two S-AFAs and show that the procedure is effective in practice. The algorithm extends to S-AFAs the *bisimulation up to congruence* technique for solving the language equivalence problem for nondeterministic finite automata recently proposed by Bonchi and Pous [5]. The algorithm belongs to a family of techniques based on the principle that two configurations of an automaton recognize the same language if and only if there is a bisimulation relation that relates them. Hopcroft and Karp's classical algorithm for checking equivalence of deterministic finite automata is a member of the family that employs a *bisimulation up to* technique [5, 21]. Rather than compute a bisimulation relation (which may be quadratic in the number of configurations of the DFA), Hopcroft and Karp's algorithm computes a relation R that is a bisimulation *up to* equivalence, in the sense that the equivalence relation generated by R is a bisimulation. *Bisimulation up to congruence* improves upon this technique by exploiting additional structure on the configurations of a nondeterministic automaton (the configurations of the NFA are finite disjunctions of states and if (say) if $a_1 \ R \ b_1$ and $a_2 \ R \ b_2$, then we may derive $(a_1 \vee a_2) \ R \ (b_1 \vee b_2)$.

We extend this technique in two ways. First, we show how the framework can be applied to alternating automata by exploiting the lattice structure on s-AFA configurations to compute a small relation that generates a bisimulation, and by using a propositional satisfiability solver to compute the congruence closure. Second, we give a technique for extending the algorithm to symbolic alphabets by showing how to efficiently enumerate a set of represenative characters, in a style reminiscent of the way that SAT solvers enumerate all satisfying assignments to a propositional formula.

We implemented our algorithm and evaluated it on a comprehensive set of verification and security benchmarks. First, we used s-AFAs to check satisfiability of more than 10,000 LTL formulas appearing in [17] using the semantics of LTL over finite traces from [15] and compared our implementation against the tool Mona [19]. Second, we used s-AFAs to check equivalence of Boolean combinations of complex regular expressions appearing in [reg] and compare against existing solutions based on non-alternating finite automata. Our experiments show that s-AFAs and our bisimulation technique often outperforms existing techniques and can be beneficial in both such applications.

Contributions. In summary our contributions are:

- Symbolic Alternating Finite Automata, s-AFAs, an automata model that can describe languages of strings operating over large and potentially infinite alphabets and for which Boolean operations have linear time complexity (Section 3.
- An algorithm for checking equivalence of two s-AFAs, which integrates bisimulation up to congruence with propositional SAT solving (Section 4).
- A modular, open-source implementation of s-AFAs, which allows the users to provide custom definitions of the alphabet Boolean algebra, and an efficient implementation of our equivalence algorithm (Section 5)
- A comprehensive evaluation of our model and its decision procedures on more than 40,000 benchmarks from real world applications (Section 6).

2. Motivating example

In this section we present an application of s-AFAs and their equivalence algorithm in the context of spam detection.

Spam detection is a notoriously hard task and spam filters are continuously modified to either take into account novel malicious behaviours or to relax existing assumption to handle overly restricting behaviours. While machine learning is the typical choice for spam detection, certain companies prefer using custom filters created using regular-expression-based black- and white-listing. The number of such filters can be very large and redundant expressions that cover already considered behaviours are often mistakenly added to the set of filters. Efficiently processing all such expressions can become a complicated task and it is therefore undesirable to avoid adding redundant filters to the list of processed ones.

For the sake of this example, we assume that a spam filter is given as a set of regular expressions $R = \{r_1, \dots, r_n\}$ with the following string: a string is an instance of the spam filter R if it belongs to the language of each regular expression $r_i \in R$. A simple example of a spam scenario is given in Fig. 1.[1]

When a new spam filter $R' = \{r'_1, \dots, r'_n\}$ is added to the set of all spam filters S, we might want to see whether there exists a spam filter R'' that is already in S and that subsumes R'. Similarly, if we have a regular expression W describing known good inputs, we might want to check that none of the strings

[1] The example is inspired from `https://theadminzone.com/threads/list-of-spam-email-addresses.27175/`.

Contains `.ru` email	`.*@.*\.ru`
Contains the word `free`	`free`
Contains a Cyrillic character	`[U+0400U+04FF]`

Figure 1. Regexes forming a spam scenario.

in W is classified as spam. As is well known, we can perform these checks by building the finite automaton corresponding to each regular expression and by using the appropriate automata operations. However, as mentioned in the introduction we face some problems.

- Regular expressions operate over very large alphabets with up to 2^{16} characters that can make classic automata operations highly impractical.
- Repeated Boolean automata operations can cause an exponential blow-up in the number of states of the resulting automaton.
- Checking equivalence of non-deterministic automata is a PSPACE-complete problem and in general requires automata determinization.

s-AFAs, the model proposed in this paper explicitly addresses the first two issues and provides a practical algorithm for solving the third one. s-AFAs combine two existing automata models.

Symbolic finite automata extend classic automata to large and potentially infinite alphabets by allowing transitions to carry predicates over a given alphabet theory. For example, the third regular expression in Figure 1 can be succinctly represented using the symbolic finite automaton with transitions

$$q_0 \xrightarrow{true} q_0 \quad q_0 \xrightarrow{[U+0400U+04FF]} q_1 \quad q_1 \xrightarrow{true} q_1$$

where q_0 is an initial state and q_1 is the only final state. Transitions carry a predicates and can be traversed when the processed symbol is a model of the predicate. For example a *true*-labeled transition can be traversed when reading any input character in the alphabet. This model enables succinct representation of large input alphabets.

Alternating finite automata extend classic automata by allowing Boolean operations to be performed at the transition level. For example, assume we are given three finite automata A_1, A_2, and A_3 for the regular expressions in Figure 1 with initial states p_1, p_2, and p_3. One can build an alternating finite automaton A accepting the intersection of the languages accepted by the three automata by simply taking the disjoint union of the three automata A_1, A_2, and A_3 and setting $p_1 \wedge p_2 \wedge p_3$ as the initial state. Informally, the initial state says that the alternating finite automaton A accepts a string s iff s is in the language of the state p_1, of the state p_2, and of the state p_3. We defer the formalization of this concept to Section 3. Thanks to this transition structure, Boolean operations over alternating automata can be performed in linear time and space. These complexities are in sharp contrast with those of classic automata where intersection has quadratic complexity and complementation has exponential complexity.

Our model combines these two models and can handle large alphabets and perform Boolean operations in linear time and space. While s-AFA inherit the efficient Boolean operations from alternating finite automata, they also inherit the complexity of deciding equivalence and emptiness: both are PSPACE-complete. (In fact these two problems are reducible to each other in linear time.) However in this paper we propose an equivalence algorithm that can efficiently solve problems like the regular expression analysis described in this section on many practical instances that existing models cannot handle.

We considered real regular expressions taken from [reg] and we were able to prove equivalences of the form $r_1 \cap r_2 \cap r_3 = r_4 \cap r_5$ in milliseconds for instances for which the classic decision procedure based on non-alternating automata could not terminate in 20 seconds. We expand on this evaluation in Section 6.

3. Symbolic alternating finite automata

This section gives a formal description of symbolic alternating finite automata (S-AFA). The two key features of S-AFAs are that (1) the alphabet is symbolic (as in a symbolic finite automaton), and (2) the automaton may make use of both existential and universal nondeterminism (as in an alternating finite automaton).

As in an S-FA, the symbolic alphabet of an S-AFA is manipulated algorithmically via an effective Boolean algebra. An *effective Boolean algebra* $\mathcal{A}$ has components $(\mathfrak{D}, \Psi, [\![\_]\!], \bot, \top, \vee, \wedge, \neg)$. $\mathfrak{D}$ is a set of *domain elements*. Ψ is a set of *predicates* closed under the Boolean connectives and $\bot, \top \in \Psi$. The *denotation function* $[\![\_]\!] : \Psi \to 2^{\mathfrak{D}}$ is such that, $[\![\bot]\!] = \emptyset$, $[\![\top]\!] = \mathfrak{D}$, for all $\varphi, \psi \in \Psi$, $[\![\varphi \vee \psi]\!] = [\![\varphi]\!] \cup [\![\psi]\!]$, $[\![\varphi \wedge \psi]\!] = [\![\varphi]\!] \cap [\![\psi]\!]$, and $[\![\neg\varphi]\!] = \mathfrak{D} \setminus [\![\varphi]\!]$. For $\varphi \in \Psi$, we write $\mathit{IsSat}(\varphi)$ when $[\![\varphi]\!] \neq \emptyset$ and say that φ is *satisfiable*. In the following we will assume that IsSat is a computable function and that, for every domain element $a \in \mathfrak{D}$ and predicate φ, it is decidable to check whether $a \in [\![\varphi]\!]$.

The intuition is that such an algebra is represented programmatically as an API with corresponding methods implementing the Boolean operations and the denotation function. The following are examples of decidable effective Boolean algebras.

$\mathbf{2}^{\mathrm{BV}k}$ is the powerset algebra whose domain is the finite set $\mathrm{BV}k$, 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. The Boolean operations correspond to the BDD operations and $\bot$ 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 $\mathit{div}_2 \wedge \mathit{div}_3$ denotes the set of numbers divisible by six.

Nondeterministic automata generalize deterministic automata by allowing a state to have multiple outgoing transitions labelled with the same character. A word is accepted by the nondeterministic automaton when *some* run leads to an accepting state (i.e., choice is interpreted *existentially*). One may naturally consider the dual interpretation of choice, wherein a word is accepted when *all* runs lead to an accepting state (i.e., choice is interpreted *universally*). An *alternating* finite automaton supports both types of nondeterminism. Nested combinations of existential and universal choices can naturally be represented by positive Boolean formulas. Formally, for any set X, we use $\mathcal{B}^+(X)$ to denote the set of *positive Boolean formulas over* X (that is, Boolean formulas built from *true*, *false*, and the members of X using the binary connectives $\wedge$ and $\vee$).

Definition 3.1 (Symbolic alternating finite automaton). *A symbolic alternating finite automaton (S-AFA) is a tuple $M = \langle \mathcal{A}, Q, p_0, F, \Delta \rangle$ where $\mathcal{A}$ is a decidable effective Boolean algebra, Q is a finite set of states, $p_0 \in \mathcal{B}^+(Q)$ is a positive Boolean formula over Q, $F \subseteq Q$ is a set of accepting states, and $\Delta \subseteq Q \times \Psi_{\mathcal{A}} \times \mathcal{B}^+(Q)$ is a finite set of transitions.*

An S-AFA over an effective Boolean algebra $\mathcal{A}$ recognizes a language of words over the set of characters $\mathfrak{D}_{\mathcal{A}}$, which we will define presently. Let $M = \langle \mathcal{A}, Q, p_0, F, \Delta \rangle$ be an S-AFA. We define a function $\mathcal{L}_M(\cdot) : \mathcal{B}^+(Q) \to 2^{\mathfrak{D}_{\mathcal{A}}^*}$ mapping each positive Boolean formula to the language accepted by that formula to be the least function (in pointwise inclusion ordering) that satisfies:

$$\begin{aligned}
w \in \mathcal{L}_M(\mathit{true}) &\quad \text{always} \\
w \in \mathcal{L}_M(\mathit{false}) &\quad \text{never} \\
\epsilon \in \mathcal{L}_M(s) &\iff s \in F \\
aw \in \mathcal{L}_M(s) &\iff \exists \langle s, \varphi, q \rangle \in \Delta \text{ s.t. } a \in [\![\varphi]\!] \wedge w \in \mathcal{L}_M(q) \\
w \in \mathcal{L}_M(p \wedge q) &\iff w \in \mathcal{L}_M(p) \wedge w \in \mathcal{L}_M(q) \\
w \in \mathcal{L}_M(p \vee q) &\iff w \in \mathcal{L}_M(p) \vee w \in \mathcal{L}_M(q)
\end{aligned}$$

Finally, we define the language $\mathcal{L}(M)$ recognized by M as $\mathscr{L}(M) \triangleq \mathcal{L}_M(p_0)$.

Unsurprisingly, the relationship between S-AFAs and S-FAs is analogous to the relationship between AFAs and NFAs: S-AFAs and S-FAS recognize the same family of languages, but converting an S-AFA with n states to an S-FA can require up to 2^n states. Interestingly, the symbolic alphabet is another source of complexity in the conversion from S-AFA to S-FA. Consider an S-AFA with two states $Q = \{x, y\}$ and two outgoing transitions per state

$$\Delta = \{\langle x, \varphi_1, x \rangle, \langle x, \varphi_2, y \rangle, \langle y, \psi_1, y \rangle, \langle y, \psi_2, \mathit{true} \rangle\}$$

The states of the equivalent S-FA can be identified with the positive Boolean formula over Q. The state $p \wedge q$ must have *four* outgoing transitions, one for each combination of the guards of x and y ($\varphi_1 \wedge \psi_1$, $\varphi_1 \wedge \psi_2$, $\varphi_2 \wedge \psi_1$, and $\varphi_2 \wedge \psi_2$). In general, for an S-AFA with n states each with m outgoing transitions, the equivalent S-FA can have states with up to m^n outgoing transitions.

3.1 Boolean operations on S-AFAs

One of the critical features of S-AFAs is that Boolean operations have linear complexity in the number of states. The constructions for S-AFA union, intersection, and complementation follow the standard ones for AFA, with the exception that S-AFA complementation (like S-FA complementation) requires a preprocessing step. For the sake of completeness, we will recall these constructions below.

Suppose that $M = \langle \mathcal{A}, Q, p_0, F, \Delta \rangle$ and $M' = \langle \mathcal{A}, Q', p'_0, F', \Delta' \rangle$ are S-AFAs over the same effective Boolean algebra and (without loss of generality) disjoint state spaces. Their union and intersection are defined simply by:

- $M \cup M' = \langle \mathcal{A}, Q \cup Q', p_0 \vee p'_0, F \cup F', \Delta \cup \Delta' \rangle$
- $M \cap M' = \langle \mathcal{A}, Q \cup Q', p_0 \wedge p'_0, F \cup F', \Delta \cup \Delta' \rangle$

(i.e., the set of states, set of final states, and transitions of the union/intersection S-AFAs are just the union of the component S-AFAs; they differ only in the initial state, which is either the disjunction (union) or conjunction (intersection) of the initial states of the components.

The complement construction for an S-AFA $M = \langle \mathcal{A}, Q, p_0, F, \Delta \rangle$ relies on M satisfying the property that for all states $x \in Q$, the set $\{[\![\varphi]\!] : \exists p. \langle x, \varphi, p \rangle \in \Delta\}$ forms a partition of $\mathfrak{D}_{\mathcal{A}}$. An S-AFA that satisfies this condition is called *normal*. Any S-AFA can be converted into an equivalent normal S-AFA using the *normalization* procedure pictured in Algorithm 1. (The algorithm is similar to the one in [27] for computing all satisfiable assignments to a propositional formula, and the one in [32] for computing satisfiable Boolean combinations of a set of predicates, and also the representative character enumeration algorithm that we will present in the next section – there it will be explained in greater detail). Normalization may (in the worst case) cause an exponential blow-up

```
Procedure Normalize($M$)
    Input  : S-AFA $M = \langle \mathcal{A}, Q, p, F, \Delta \rangle$
    Output: Equivalent normal S-AFA M'
    $\Delta' \leftarrow \emptyset$
    foreach $x \in Q$ do
        $chars \leftarrow \top$
        while IsSat($chars$) do
            $a \leftarrow$ Witness($chars$)
            $p = false$
            $class \leftarrow \top$
            foreach $\langle x, \varphi, q \rangle \in \Delta$ do
                if $a \in [\![\varphi]\!]$ then
                    $class \leftarrow class \wedge \varphi$
                    $p \leftarrow p \vee q$
                else
                    $class \leftarrow class \wedge \neg\varphi$
            $chars \leftarrow chars \wedge \neg class$
            Add $\langle x, class, p \rangle$ to $\Delta'$
    return $\langle \mathcal{A}, Q, p, F, \Delta' \rangle$
```

Algorithm 1: Normalization algorithm for S-AFA

in the number of outgoing transitions of any one state in an S-AFA (note, however, that the exponential factor does not depend on the number of states).

Assuming that M is normal, the complement can be constructed by "De Morganization." We define the complement to be

$$\overline{M} \triangleq \langle \mathcal{A}, Q, \overline{p_0}, Q \setminus F, \{\langle x, \varphi, \overline{p} \rangle : \langle x, \varphi, p \rangle \in \Delta\rangle,$$

where $\bar{\cdot}$ denotes the positive Boolean formula tranformation that replaces every $\wedge$ with $\vee$ (and vice versa).

3.2 An algebraic view of S-AFA

This section describes S-AFA in a more algebraic style, which will be useful in the next section.

A *bounded lattice* $\mathcal{L} = \langle L, \sqsubseteq, \sqcup, \sqcap, \bot, \top \rangle$ is a partially ordered set $\langle L, \sqsubseteq \rangle$ such that every finite set of elements has a least upper bound and greatest lower bound. For any pair of elements $x, y \in L$, we use $x \sqcup y$ to denote their least upper bound and $x \sqcap y$ to denote the greatest lower bound. The least element of the lattice (the least upper bound of the empty set) is denoted by $\bot$ and the greatest (the greatest lower bound of the empty set) by $\top$. We say that $\mathcal{L}$ is *distributive* if for all $a, b, c \in L$, we have $a \sqcap (b \sqcup c) = (a \sqcap b) \sqcup (a \sqcap c)$.

Our first example of a bounded lattice is the (distributive) bounded lattice of positive Boolean formulas. Operating (as we do) under the assumption that we do not distinguish between logically equivalent positive Boolean formulas, for any set X, $\mathcal{B}^+(X)$ is a bounded lattice where the order is logical entailment, the least upper bound is disjunction, the greatest lower bound is conjunction, $\bot$ is *false*, and $\top$ is *true*. A second important example is the Boolean lattice $\mathbf{2} \triangleq \langle \{0, 1\}, \leq, \vee, \wedge, 0, 1 \rangle$ (which is also bounded and distributive).

Let X be a set. A *model* over X is a function $m : X \to \mathbf{2}$ that assigns each $x \in X$ a Boolean value. The model m can be extended to evaluate any positive Boolean formula by defining:

$$m(\mathit{false}) \triangleq 0$$
$$m(\mathit{true}) \triangleq 1$$
$$m(p \wedge q) \triangleq m(p) \wedge m(q)$$
$$m(p \vee q) \triangleq m(p) \vee m(q)\ .$$

Thus, we say that the model $m : X \to \mathbf{2}$ *extends uniquely to a lattice homomorphism* $\mathcal{B}^+(X) \to \mathbf{2}$. In fact, there is nothing special about the bounded lattice $\mathbf{2}$ in this regard: if $\mathcal{L} = \langle L, \sqcup, \sqcap, \bot, \top \rangle$ is a bounded lattice, then any function $f : X \to \mathcal{L}$ extends uniquely to a lattice homomorphism $\mathcal{B}^+(X) \to \mathcal{L}$.[2] In the following, our notation will not distinguish between a function $f : X \to \mathcal{L}$ and its extension.

Let $M = \langle \mathcal{A}, Q, p_0, F, \Delta \rangle$ be an S-AFA. The set $F \subseteq Q$ of final states defines a model $F : Q \to \mathbf{2}$ over Q as follows:

$$F(s) \triangleq \begin{cases} 1 & \text{if } s \in F \\ 0 & \text{otherwise}\ . \end{cases}$$

Note that for any $p \in \mathcal{B}^+(Q)$, we have $F(p) = 1$ if and only if $\mathcal{L}_M(p)$ contains the empty word. Any character $a \in \mathfrak{D}_\mathcal{A}$ can be associated with a transition *function* $\Delta_a : Q \to \mathcal{B}^+(Q)$, where

$$\Delta_a(s) \triangleq \bigvee \{q : \exists \varphi \in \Psi_\mathcal{A}. \langle s, \varphi, q \rangle \in \Delta \wedge a \in [\![\varphi]\!]\}\ .$$

Recall that since Δ_a is a function into a bounded lattice (namely $\mathcal{B}^+(Q)$ itself) it extends uniquely to a lattice homomorphism $\mathcal{B}^+(Q) \to \mathcal{B}^+(Q)$. Similarly, any word $w = a_1...a_n \in \mathfrak{D}_\mathcal{A}^*$ can be associated with a transition function $\Delta_a : Q \to \mathcal{B}^+(Q)$ where

$$\Delta_w \triangleq \Delta_{a_n} \circ \cdots \circ \Delta_{a_1}\ .$$

Finally observe that we can characterize the language recognized by an S-AFA succinctly using the algebraic machinery described in this section: for any $p \in \mathcal{B}^+(Q)$, we have

$$w \in \mathcal{L}_M(p) \iff F(\Delta_w(p)) = 1\ .$$

4. Equivalence checking for S-AFAs

This section describes an algorithm for checking whether two symbolic alternating finite automata recognize the same language. Recently, Bonchi and Pous introduced the *bisimulation up to congruence* technique for solving the language equivalence problem for non-deterministic finite automata [5]. We extend this technique in two ways: (1) we show how the framework can be applied to alternating automata, using a propositional satisfiability solver to compute congruence closure; (2) we give a technique for extending the technique to symbolic alphabets.

4.1 Bisimulation up to congruence

We will begin by recalling some of the details of bisimulation up to congruence, adpated to our setting of symbolic alternating finite automata.

Definition 4.1 (Bisimulation)**.** *Let $M = \langle \mathcal{A}, Q, p_0, F, \Delta \rangle$ be an* S-AFA*, and let $R \subseteq \mathcal{B}^+(Q) \times \mathcal{B}^+(Q)$ be a binary relation on positive Boolean formulas over M's states. We say that R is a* ***bisimulation*** *if for all p, q such that $p\ R\ q$, we have*

- Consistency*: $F(p) = F(q)$*
- Compatibility*: For all $a \in \mathfrak{D}_\mathcal{A}$, $\Delta_a(p)\ R\ \Delta_a(q)$.*

Consistency and compatibility are useful notions outside of the context of bisimulations, so we will provide more general definitions. For any relation $R \subseteq X \times X$ and any function $f : X \to X$, we say that f is *compatible* with R if $x\ R\ y$ implies $f(x)\ R\ f(y)$. A function $f : X \to \mathbf{2}$ is *consistent* with R if $x\ R\ y$ implies $f(x) = f(y)$. Clearly, compatible functions are closed under composition, and the composition of a compatible function with a consistent function is consistent.

The following proposition states the soundness and completeness of the principle of bisimulations for language equivalence checking.

Proposition 4.2. *Let $M = \langle \mathcal{A}, Q, p_0, F, \Delta \rangle$ be an* S-AFA.

[2] Succinctly, $\mathcal{B}^+(X)$ is the *free* bounded distributive lattice generated by X.

1. *For any bisimulation $R \subseteq \mathcal{B}^+(Q) \times \mathcal{B}^+(Q)$ and any $p, q \in \mathcal{B}^+(Q)$ such that $p\ R\ q$, we have $\mathcal{L}_M(p) = \mathcal{L}_M(q)$.*
2. *The relation $\sim$ defined by*
$$p \sim q \iff \mathcal{L}_M(p) = \mathcal{L}_M(q)$$
is a bisimulation.

We will now define bisimulation up to congruence for S-AFAs.

Definition 4.3 (Congruence closure)**.** *Let Q be a finite set and let $R \subseteq \mathcal{B}^+(Q) \times \mathcal{B}^+(Q)$ be a binary relation. The* **congruence closure** *of R, denoted $\equiv_R$, is the smallest congruence relation that contains R. That is, $\equiv_R$ is the smallest reflexive, transitive, and symmetric relation that contains R and such that for all $p_1, p_2, q_1, q_2 \in \mathcal{B}^+(Q)$ such that $p_1 \equiv_R q_1$ and $p_2 \equiv_R q_2$, we have $p_1 \wedge p_2 \equiv_R q_1 \wedge q_2$ and $p_1 \vee p_2 \equiv_R q_1 \vee q_2$.*

Definition 4.4 (Bisimulation up to congruence)**.** *Let $M = \langle \mathcal{A}, Q, p_0, F, \Delta \rangle$ be an S-AFA, and let $R \subseteq \mathcal{B}^+(Q) \times \mathcal{B}^+(Q)$ be a binary relation. We say that R is a* bisimulation up to congruence *if for all p, q such that $p\ R\ q$, we have*

- Consistency: $F(p) = F(q)$
- Compatibility: *For all $a \in \mathfrak{D}_{\mathcal{A}}$, $\Delta_a(p) \equiv_R \Delta_a(q)$.*

Bisimulation up to congruence allows us to solve language equivalence queries as follows: if R is a bisimulation up to congruence such that $p\ R\ q$, then p and q recognize the same language. This follows from Proposition 4.2 and the following:

Proposition 4.5. *Let $M = \langle \mathcal{A}, Q, p_0, F, \Delta \rangle$ be an S-AFA. Let $R \subseteq \mathcal{B}^+(Q) \times \mathcal{B}^+(Q)$ be a binary relation on positive Boolean formulas over M's states. R is a bisimulation up to congruence if and only if $\equiv_R$ is a bisimulation.*

We delay the proof of this proposition to the next section, after we have developed some technical machinery for checking whether a given relation is a bisimulation up to congruence.

4.2 Congruence checking

First, we will address an algorithmic challenge: *how may one check whether a given relation is a bisimulation up to congruence*? Generating the congruence closure $\equiv_R$ explicitly is intractable, since the cardinality of $\equiv_R$ may be double-exponentially larger than that of R. However, for the purpose of checking whether a relation R is a bisimulation up to congruence, we need only to be able to check membership within the congruence closure. Thus, we are interested in the CONGRUENCE problem, which is stated as follows: given a finite set Q, a finite relation $R \subseteq \mathcal{B}^+(Q) \times \mathcal{B}^+(Q)$, and two positive Boolean formulas $p, q \in \mathcal{B}^+(Q)$, determine whether $p \equiv_R q$. In the following, we will show that the CONGRUENCE problem is NP-complete, but it can be solved in practice by exploiting propositional satisfiability solvers. Towards this end, we define the *logical closure* of a relation as follows:

Definition 4.6. *Let Q be a finite set and let $R \subseteq \mathcal{B}^+(Q) \times \mathcal{B}^+(Q)$ be a binary relation. Define $cl(R)$ as the* logical closure *of R as follows:*
$$\Phi(R) \triangleq \bigwedge_{pRq} p \iff q$$
$$cl(R) \triangleq \{\langle p, q \rangle : \Phi(R) \models p \iff q\}.$$

Observe that for any R, p, and q, we have $p\ cl(R)\ q$ if and only if $\Phi(R) \wedge \neg(p \iff q)$ is unsatisfiable. Thus, membership in $cl(R)$ reduces to a propositional satisfiability problem.

Note that every propositional model $m : Q \to \mathbf{2}$ extends uniquely to a bounded lattice homomorphism $\mathcal{B}^+(Q) \to \mathbf{2}$ (where for all $p \in \mathcal{B}^+(Q)$, $m(p) = 1 \iff m \models p$). In light of this, m is a model of $\Phi(R)$ if and only if m is consistent with R, and $p\ cl(R)\ q$ if and only if $h(p) = h(q)$ for every bounded lattice homomorphism $h : \mathcal{B}^+(Q) \to \mathbf{2}$ that is consistent with R. That is, $cl(R)$ is the *largest* relation such that that every bounded lattice homomorphism that is consistent with R is consistent with $cl(R)$.

The question now is what is the relationship between the congruence closure and the logical closure. We will show that they are identical. First, a lemma:

Lemma 4.7. *Let $\mathcal{L} = \langle L, \sqcup, \sqcap, \bot, \top \rangle$ be a finite bounded lattice. For any two distinct elements a, b in L, there is a homomorphism $f : \mathcal{L} \to \mathbf{2}$ such that $f(a) \neq f(b)$.*

Proof. Without loss of generality, suppose that $a \not\sqsubseteq b$. Let c be a join-irreducible element c of L such that $c \sqsubseteq a$ and $c \not\sqsubseteq b$. The existence of such a c can be proved by contradiction: suppose there exists some $a \not\sqsubseteq b$ such that there is no join-irreducible element c of L such that $c \sqsubseteq a$. Then there exists a least such element a. By assumption, a is join-reducible so there exists d and d' such that $d \sqsubset a$, $d' \sqsubset a$, and $d \sqcup d' = a$. It cannot be the case that both $d \sqsubseteq b$ and $d' \sqsubseteq b$ (if so, then $d \sqcup d' \sqsubseteq b$, but by construction we have $a = d \sqcup d' \not\sqsubseteq b$). Suppose without loss of generality that $d \not\sqsubseteq b$. By minimality of a, there is some join-irreducible element $c \sqsubseteq d$ such that $c \not\sqsubseteq b$. Since $d \sqsubset a$, we have $c \sqsubseteq a$ and we are done.

Construct a function $f : L \to \mathbf{2}$ by defining
$$f(x) \triangleq \begin{cases} 1 & \text{if } c \leq x \\ 0 & \text{otherwise} \end{cases}$$
One may check that f is a bounded lattice homomorphism with $f(a) = 1$ and $f(b) = 0$. □

Proposition 4.8. *Let Q be a finite set and let $R \subseteq \mathcal{B}^+(Q) \times \mathcal{B}^+(Q)$ be a binary relation. Then $cl(R)$ coincides with $\equiv_R$.*

Proof. Clearly $cl(R)$ is a congruence relation containing R, so $\equiv_R$ is a subset of $cl(R)$. It remains to show that $cl(R)$ is a subset of $\equiv_R$, or equivalently that if p and q are *not* related by $\equiv_R$, then they are not related by $cl(R)$.

Let p and q be such that $p \not\equiv_R r$. Then the equivalence classes $[p]$ and $[q]$ are distinct in the quotient lattice $\mathcal{B}^+(Q)/_{\equiv_R}$. By Lemma 4.7, there is a bounded lattice homomorphism $f : \mathcal{B}^+(Q)/_{\equiv_R} \to \mathbf{2}$ such that $f([p]) \neq f([q])$. Then clearly $f \models \Phi(R)$ (viewing f as a propositional model), but (since $f([p]) \neq f([q])$), $f \not\models p \iff q$. Therefore, $\Phi(R) \not\models p \iff q$, and p and q are not related by $cl(R)$. □

Proposition 4.8 yields a simple candidate algorithm for the CONGRUENCE problem: simply check whether the pair of positive Boolean formulas belongs to the logical closure of a relation using a SAT solver. The following proposition states that we cannot hope for an asymptotically superior algorithm.

Proposition 4.9. *CONGRUENCE is NP-complete.*

Proof. Membership in NP follows immediately from Proposition 4.8. We prove NP-hardness of CONGRUENCE by giving polytime reduction from SAT. The key insight is that the relation R can be used to axiomatize negation, so that arbitrary Boolean formulas can be encoded into positive Boolean formulas.

Let φ be a Boolean formula in conjunctive normal form over a set of propositional variables P. Form a new set of propositional variables
$$Q \triangleq P \cup \{\overline{p} : p \in P\}$$
consisting of the original propositional variables P plus a disjoint set of "barred" copies, intended to represent negative literals. Define a relation $R \subseteq \mathcal{B}^+(Q) \times \mathcal{B}^+(Q)$ as follows:
$$R \triangleq \{\langle p \wedge \overline{p}, \mathit{false} \rangle : p \in P\} \cup \{\langle p \vee \overline{p}, \mathit{true} \rangle : p \in P\}$$

Finally, let $\widehat{\varphi}$ be the formula obtained by replacing every negative literal $\neg p$ with $\overline{p}$. Then φ is satisfiable if and only if $\widehat{\varphi} \not\equiv_R$ *false*. $\square$

Finally, using the technical machinery we have developed in this section, we will re-state and prove Proposition 4.5.

Proposition 4.5. *Let $M = \langle \mathcal{A}, Q, p_0, F, \Delta \rangle$ be an* S-AFA*. Let $R \subseteq \mathcal{B}^+(Q) \times \mathcal{B}^+(Q)$ be a binary relation on positive Boolean formulas over M's states. R is a bisimulation up to congruence if and only if $\equiv_R$ is a bisimulation.*

Proof. Consistency: Suppose that $p \equiv_R q$. Since $cl(R)$ coincides with $\equiv_R$, we have $p\ cl(R)\ q$, and thus $\Phi(R) \models p \iff q$. Since R is a bisimulation up to congruence, F is consistent with R, and thus $F \models \Phi(R) \models p \iff q$. Thus $F(p) = F(q)$.

Compatibility: Towards the converse, suppose that there exists positive Boolean formulas $p, q \in \mathcal{B}^+(Q)$ and a character $a \in \mathfrak{D}_\mathcal{A}$ such that $\Delta_a(p) \not\equiv_R \Delta_a(q)$, and show that $p \not\equiv_R q$.

Since $\Delta_a(p) \not\equiv_R \Delta_a(q)$, and $cl(R)$ coincides with $\equiv_R$, there exists a model $m : Q \to \mathbf{2}$ such that $m \models \Phi(R)$ but $m \not\models \Delta_a(p) \iff \Delta_a(q)$. Since $m(\Delta_a(p)) \neq m(\Delta_a(q))$, it is sufficient to show that $m \circ \Delta_a$ is consistent with R (since if p and q can be distinguished by a homomorphism consistent with R, then it cannot be the case that $p\ cl(R)\ q$, and thus $p \not\equiv_R q$). Towards proving that $m \circ \Delta_a$ is consistent with R, let r, s be such that $r\ R\ s$, and prove that $m(\Delta_a(r)) = m(\Delta_a(s))$. Since R is a bisimulation up to congruence, we have $\Delta_a(r) \equiv_R \Delta_a(s)$. Since m is consistent with R, we have $m \models \Delta_a(r) \iff \Delta_a(s)$ and thus $m(\Delta_a(r)) = m(\Delta_a(s))$. $\square$

4.3 Equivalence algorithm

We will now show how the theory of bisimulation up to congruence can be leveraged in a decision procedure for S-AFA language equivalence. The algorithm addresses two challenges raised by bringing Bonchi and Pous' NFA equivalence algorithm to bear on symbolic alternating finite automata: (1) how to efficiently check membership in the congruence closure of a relation, and (2) how to efficiently enumerate a sufficient finite set of characters on which to verify the bisimuation conditions.

The equivalence decision procedure is pictured in Algorithm 2. The idea is simple: given an S-AFA $M = \langle \mathcal{A}, Q, p_0, F, \Delta \rangle$ and two positive Boolean formulas p_0 and q_0, the algorithm attempts to synthesize a bisimulation up to congruence R such that $p_0\ R\ q_0$ or show that no such R exists.

Checking congruence closure membership The algorithm implicitly maintains a relation R such that any bisimulation that contains $\langle p_0, q_0 \rangle$ *must* contain R. The congruence closure of R is represented using an incremental SAT solver. An incremental SAT solver s internally maintains a *context formula* (initially *true*) and supports two operations: $s.push(\varphi)$ conjoins the constraint φ to s's context, and $s.isSat(\varphi)$ checks whether the conjunction of φ and s's context is satisfiable. The congruence closure of $R = \{\langle p_1, q_1 \rangle, ..., \langle p_n, q_n \rangle\}$ is represented by a SAT solver with context $\Phi(R) = (p_1 \iff q_1) \wedge \cdots \wedge (p_n \iff q_n)$ (cf. Definition 4.6 and Proposition 4.8). We may add a pair $\langle p, q \rangle$ to the relation R by calling $s.push(p \iff q)$ and we may check whether a given pair $\langle p, q \rangle$ belongs to $\equiv_R$ by issuing the query $s.isSat(p \iff q)$.

The relation R is initialized to the singleton set containing the pair of $\langle p_0, q_0 \rangle$ of formulas for which we wish to decide equivalence (line 4). Recall that R is a bisimulation up to congruence if and only if it satisfies the *consistency* and *compatibility* conditions given in Definition 4.4. Thus, if at any point the relation R contains a pair $\langle p, q \rangle$ such that $F(p) \neq F(q)$ (i.e., R fails the consistency condition), the algorithm returns false (lines 22-23), having proved that $\mathcal{L}_M(p_0) \neq \mathcal{L}_M(q_0)$. Towards the consistency condition, the algorithm maintains a *worklist* such that every pair $\langle p, q \rangle \in R$ such that $\Delta_a(p) \not\equiv_R \Delta_a(q)$ for some character $a \in \mathfrak{D}_\mathcal{A}$ belongs to *worklist*. The algorithm returns *true* when *worklist* is empty (equivalently, when the consistency condition holds).

Enumeration of representative characters Each iteration of the main loop (lines 5-26) removes a pair $\langle p, q \rangle$ from the worklist and adds pairs to R that are implied by the membership of $\langle p, q \rangle$ in R and the compatibility condition. A naive way to do this is to iterate over the alphabet $\mathfrak{D}_\mathcal{A}$, and for each character $a \in \mathfrak{D}_\mathcal{A}$ add $\langle \Delta_a(p), \Delta_a(q) \rangle$ to R and *worklist* if $\Delta_a(p) \not\equiv_R \Delta_a(q)$. However, iterating over the alphabet is not effective because the set of characters may be infinite.

The algorithm overcomes this problem by iterating over a finite set of *representative* characters, such that if $\Delta_a(p) \equiv_R \Delta_a(q)$ holds for all representative characters a then it holds for all characters. One natural candidate for the set of representative characters is to choose one member of each equivalence class of the relation $\simeq$ defined by

$$a \simeq b \iff \forall x \in Q. \Delta_a(x) = \Delta_b(x)\ .$$

(That is, $a \simeq b$ if the transition functions Δ_a and Δ_b are equal). Observe that any set B containing one member of each equivalence class is a valid choice for a representative set of characters:

1. B is finite: recalling that for any character a, Δ_a is defined by
$$\Delta_a(x) \triangleq \bigvee \{q : \exists \varphi \in \Psi_\mathcal{A}. \langle x, \varphi, q \rangle \in \Delta \wedge a \in [\![\varphi]\!]\}$$
and that Δ is a finite set, there are only finitely many equivalence classes of $\simeq$, and so B is finite.
2. B is representative: Suppose $\Delta_b(p) \equiv_R \Delta_b(q)$ for all $b \in B$, and let $a \in \mathfrak{D}_\mathcal{A}$. There is some $b \in B$ such that $a \simeq b$, so we have
$$\Delta_b(p) = \Delta_a(p) \equiv_R \Delta_a(q) = \Delta_b(q)$$
and thus $\Delta_a(p) \equiv_R \Delta_a(q)$.

A practical refinement of this idea is to employ an equivalence relation with fewer equivalence classes. Towards this end, for any set of states $S \subseteq Q$, define an equivalence relation $\simeq_S$ on the set of characters $\mathfrak{D}_\mathcal{A}$ as follows:

$$a \simeq_S b \iff \forall s \in S. \Delta_a(s) = \Delta_b(s)\ .$$

Note that the relation $\simeq_Q$ coincides with $\simeq$. The argument that any set of equivalence class representatives of $\simeq$ is a valid set of representative characters applies also to equivalence class representatives of $\simeq_S$ for any set S that contains every state appearing in p or q. Since S is typically smaller than Q, this yields a smaller representative set.

Algorithm 2 iterates over the set of representative characters (lines 7-26) by manipulating sets of characters symbolically via the effective Boolean algebra $\mathcal{A}$. The enumeration is reminiscent of the way that AllSat solvers enumerate satisfying assignments to propositional formula [27]. The variable *chars*, initially $\top$, holds a predicate representing the set of characters that remain to be processed. At each iteration of the loop, we select a character $a \in [\![\mathit{chars}]\!]$ that has not yet been processed (line 10), and compute a predicate *class* representing its equivalence class in the relation $\simeq_S$:

$$\begin{aligned}\mathit{class} \triangleq &\bigwedge \{\varphi : \exists x, q. \langle x, \varphi, q \rangle \in \Delta \wedge a \in [\![\varphi]\!]\} \\ &\wedge \bigwedge \{\neg\varphi : \exists x, q. \langle x, \varphi, q \rangle \in \Delta \wedge a \notin [\![\varphi]\!]\}\ .\end{aligned}$$

We then remove every character in a's equivalence class from the set *chars* by conjoining *chars* with the negation of *class* (line 21). This ensures that on the next iteration of the loop, we choose a character that is not equivalent to any character seen so far (in the context of AllSat, $\neg$*class* is sometimes called a *blocking clause*).

Illustrative example Let $M = \langle \mathcal{A}, Q, p_0, F, \Delta \rangle$ be an S-AFA over the theory of linear integer arithmetic where there are five states $Q = \{v, w, x, y, z\}$, all states are final, and the transitions are as follows:

$$v \xrightarrow{c \leq 0} x \vee y \quad v \xrightarrow{c>0} z \wedge w \quad w \xrightarrow{c \leq 0} z$$

$$w \xrightarrow{c>0} (y \vee x) \wedge v \quad x \xrightarrow{c=1} v \quad y \xrightarrow{c \neq 1} w \quad z \xrightarrow{true} v$$

Assume that we want to prove that the state v is equivalent to the state w. The algorithm initializes the relation R to $\{\langle v, w\rangle\}$, the worklist to $[\langle v, w \rangle]$, and enters the main loop:

1. ($R = \{\langle v, w\rangle\}$, *worklist* $= [\langle v, w\rangle]$) We pick $\langle v, w\rangle$ off the worklist, and enter the inner loop:
   (a) ($R = \{\langle v, w\rangle\}$, *worklist* $= [\,]$, *chars* $= \top$) We compute a witness to the satisfiability of *chars* – this may be any integer, but let's suppose that we choose 0. We compute the equivalence class of 0 to be *class* $= c \leq 0$ and set *chars* $\leftarrow$ *chars* $\wedge \neg(c \leq 0)$. We add $\langle \Delta_0(v), \Delta_0(w)\rangle = \langle x \vee y, z\rangle$ to R and the worklist.
   (b) ($R = \{\langle v, w\rangle, \langle x \vee y, z\rangle\}$, *worklist* $= [\langle x \vee y, z\rangle]$, *chars* $= \neg(c \leq 0)$) We generate 5 as a witness to satisfiability of *chars*. We compute the equivalence class of 5 to be *class* $= c > 0$ and set *chars* $\leftarrow$ *chars* $\wedge \neg(c > 0)$. We find that
   $$\Delta_5(v) = z \wedge w \equiv_R (y \vee x) \wedge v = \Delta_5(w)$$
   and therefore, do not add $\langle \Delta_5(v), \Delta_5(w)\rangle$ to R or the worklist.
   (c) ($R = \{\langle v, w\rangle, \langle x \vee y, z\rangle\}$, *worklist* $= [\langle x \vee y, z\rangle]$, *chars* $= \neg(c \leq 0) \wedge \neg(c > 0)$). We find that *chars* is unsatisfiable and exit the inner loop.
2. ($R = \{\langle v, w\rangle, \langle x \vee y, z\rangle\}$, *worklist* $= [\langle x \vee y, z\rangle]$) We pick $\langle x \vee y, z\rangle$ off the worklist, and enter the inner loop:
   (a) ($R = \{\langle v, w\rangle, \langle x \vee y, z\rangle\}$, *worklist* $= [\,]$, *chars* $= \top$): we compute 1 as a witness of satisfiability of *chars*. We compute the equivalence class of 1 to be *class* $= (c = 1)$ and set *chars* $\leftarrow$ *chars* $\wedge \neg(c = 1)$. We find that
   $$\Delta_1(x \vee y) = v \vee \mathit{false} \equiv_R v = \Delta_1(z)$$
   and therefore do not add $\langle \Delta_1(x \vee y), \Delta_1(z)\rangle$ to R or the worklist.
   (b) ($R = \{\langle v, w\rangle, \langle x \vee y, z\rangle\}$, *worklist* $= [\,]$, *chars* $= \neg(c = 1)$) We generate 2 as a witness to satisfiability of *chars*. We compute the equivalence class of 2 to be *class* $= c \neq 1$ and set *chars* $\leftarrow$ *chars* $\wedge \neg(c \neq 1)$. We find that
   $$\Delta_2(x \vee y) = \mathit{false} \vee w \equiv_R v = \Delta_2(z)$$
   and therefore do not add $\langle \Delta_2(x \vee y), \Delta_2(z)\rangle$ to R or the worklist.
   (c) ($R = \{\langle v, w\rangle, \langle x \vee y, z\rangle\}$, *worklist* $= [\,]$, *chars* $= \neg(c = 1) \wedge \neg(c \neq 1)$). We find that *chars* is unsatisfiable and exit the inner loop.
3. ($R = \{\langle v, w\rangle, \langle x \vee y, z\rangle\}$, *worklist* $= [\,]$) Since the worklist is empty, the algorithm terminates: R is a bisimulation up to congruence containing $\langle v, w\rangle$, so $\mathcal{L}_M(v) = \mathcal{L}_M(w)$.

5. Implementation

We implemented S-AFAs and their decision procedure in an existing Java automata library. [3]

[3] Name and link omitted for double blind.

```
Procedure IsEquivalent($(M, p_0, q_0)$)
    Input  : S-AFA $M = \langle \mathcal{A}, Q, p, F, \Delta \rangle$
             positive Boolean formulas $p_0, q_0 \in \mathcal{B}^+(Q)$
    Output: true if $\mathcal{L}_M(p_0) = \mathcal{L}_M(q_0)$, false otherwise
    $s \leftarrow$ new solver
    worklist $\leftarrow [(p_0, q_0)]$
    $s.push(p_0 \iff q_0)$
    while worklist is not empty do
        Pick $(p, q)$ off worklist
        $S \leftarrow$ set of states in $p$ or $q$
        /* $[\![chars]\!]$ is the set of characters that remain to be
           processed                                              */
        chars $\leftarrow \top$
        while IsSat(chars) do
            $a \leftarrow$ Witness(chars)
            /* Compute the transition function $\Delta_a$ and a
               predicate representing the equivalence class
               of $a$ in $\simeq_S$.                              */
            class $\leftarrow$ true
            $\Delta_a \leftarrow \lambda x.false$
            for $x \in S, \langle x, \varphi, q\rangle \in \Delta$ do
                if $a \in [\![\varphi]\!]$ then
                    class $\leftarrow$ class $\wedge \varphi$
                    $\Delta_a(x) \leftarrow \Delta_a(x) \vee q$
                else
                    class $\leftarrow$ class $\wedge \neg\varphi$
            $p' \leftarrow \Delta_a(p)$
            $q' \leftarrow \Delta_a(q)$
            /* Remove $a$'s equivalence class from chars      */
            chars $\leftarrow$ chars $\wedge \neg$class
            if $F(p') \neq F(q')$ then
                return false
            /* If $p' \not\equiv_R q'$, add $\langle p', q'\rangle$ to $R$   */
            if $s.isSat((p' \wedge \neg q') \vee (\neg p' \wedge q'))$ then
                Add $(p', q')$ to worklist
                $s.push(p \iff q)$
    return true
```
Algorithm 2: Equivalence algorithm for S-AFAs

The implementation provides an interface for specifying custom Boolean algebras for both the alphabet theory and the positive Boolean formulas over the automaton states and it can be easily integrated with externally specified alphabet theories. To represent the positive Boolean formulas over the automaton states we implemented two algebras: one which simply maintains the explicit Boolean representations of formulas (referred to as DAG in the experiments) and one which instead maintains a BDD corresponding to each formula. We use the JDDFactory implementation in JavaBDD as our BDD library (`http://javabdd.sourceforge.net/`). To check membership of formulas to the congruence closure we use the SAT solver JSAT (`https://j-sat.com/tag/jsat/`). Our implementation is open source.

Optimizations We implemented a few simple optimizations for improving the performance of our decision procedure.

First, whenever we construct an S-AFA, we remove all the states that are trivially non reachable from the initial state or that cannot reach a final state. Given a positive Boolean formula $f \in \mathcal{B}^+(Q)$, let $st(f) \subseteq Q$ be the set of states appearing in f. Given a S-AFA $M = \langle \mathcal{A}, Q, p_0, F, \Delta \rangle$, we construct a graph $G_M = (V, E)$ with set of vertices $V = Q$, and transitions $E = \{(s, s') \mid (s, \varphi, q) \in \Delta \wedge s' \in st(q)\}$. We then remove from M all the states that are not reachable from one of the states $s_0 \in st(p_0)$ in G_M and all the states that do not have a path to some state $s \in F$ in G_M. In each positive Boolean formulas of M we replace every removed state with *false*. The resulting S-AFA is equivalent to M'.

Second, note that Algorithm 2 does not specify what data structure to use for the worklist. Natural choices for such a data structure are a stack or a queue, but none of these data structures leverages the fact that smaller formulas are more likely to generate better congruences. Instead, we implement the worklist using a priority queue which always extracts the pair of smallest size, where the size of a pair (p, q) is given by the formula $|p| + |q|$.

6. Evaluation

We evaluated the performance of our algorithms on the following benchmarks.

1. We check satisfiability of the LTL formulas appearing in [17] using the semantics of LTL over finite traces from [15].
2. We check equivalence of Boolean combinations of regular expressions appearing in [reg].

All experiments were run on an Intel Core i7 2.60 GHz CPU with 16 GB of RAM.

6.1 Satisfiability checking for LTL over finite traces

Linear temporal logic (LTL) plays a prominent role in program verification and its properties have been studied extensively. While the semantics of LTL is typically defined over infinite strings, recently there has been a lot of interest in the interpretation of LTL over finite traces [15] because this variant can be used in applications such as program monitoring. We use LTL-F to refer to the interpretation of LTL over finite traces. LTL-F is as expressive as first-order logic over strings, while checking satisfiability of LTL-F formulas is a PSPACE-complete problem, there exists a linear time translation from LTL-F formulas to alternating automata [15]. Similarly to what happens with regular LTL, this translation results in an alphabet of size exponential in the number of atomic proposition appearing in the formula. Due to this reason, S-AFAs are a promising model for designing decision procedures for of LTL-F.

In this section, we evaluate the performance of our algorithm on the tasks of checking satisfiability and equivalence of LTL-F formulas using the linear time translation from LTL-F formulas to alternating automata proposed in [15] We first describe the set of considered formulas and then present one experiment for checking satisfiability and one for checking equivalence.

Benchmark formulas We consider three sets of LTL formulas. The first set (lift and lift_b in the figures) contains 16 parametric formulas describing a lift system of increasing complexity [18]. The second set (counter and counter_l in the figures) contains 30 formulas describing counters for which satisfiability under the infinite string semantics is notoriously hard [28]. Interestingly, these formulas have exactly one model under the infinite string semantics, while they are unsatisfiable under the finite string one. The third set contains more than 10,000 random formulas that were created as part of the experimental evaluation in [12]. The formulas have size varying between 10 and 100, and number of atomic propositions varying between 2 and 4. For the non-random formulas we set the timeout at 60 seconds, while for the random ones, given how many there are, we set the timeout at 5 seconds.

6.1.1 Satisfiability checking

We evaluate the performance of our algorithm for checking satisfiability of the S-AFAs corresponding to the given LTL-F formulas. We use the linear time translation from LTL-F to monadic second order logic (MSO) proposed in [15] to generate MSO formulas equivalent to the LTL-F ones and compare our implementation against Mona [19], a solver for the monadic second order logic of one successor.[4] In LTL-F the alphabet is the set of bitvectors of size n, where n is the number of atomic propositions appearing in the formula and each bit indicates whether one of the atomic propositions is true or false. The constructed S-AFAs will therefore be over the theory of bit-vectors and we use BDDs to describe predicates in such a theory.

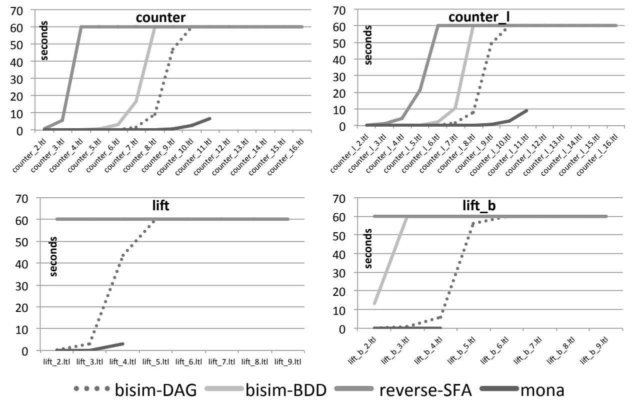

Figure 2. Satisfiability checking for non-random LTL formulas. The missing points are instances for which Mona runs out of memory.

For each formula we measure: 1) the runtime of Mona on the equivalent MSO formula (mona); 2) the runtime of computing the deterministic automaton accepting the reverse language of the S-AFA and checking its emptiness (reverse-SFA)[5]; 3) the runtime of the bisimulation algorithm using a directed acyclic graph (i.e., hash-consed) representation of positive Boolean expressions (bisim-DAG); 4) the runtime of the bisimulation algorithm using BDDs to represent positive Boolean expressions (bisim-BDD). The results for the non-random LTL formulas are depicted in Figure 2, while Figure 3 illustrates the difference in runtime between our bisimulation algorithm, reverse-S-FA, and Mona. Since the bisimulation algorithm that uses an explicit representation of positive Boolean expressions is consistently faster than the one using BDDs, Figure 3 only shows results for the former. In these graphs a point above 0 means that the bisimulation solver is faster than the other solver. Finally, in the top part of Figure 5 we show the number of times each solver timed out when checking satisfiability of the 10,000 randomly generated formulas.

Results Mona outperforms the bisimulation solver on non-random formulas. However, Mona runs out of memory for relatively small instances for which our solver does not. Mona is slower than our algorithm and times out more often on randomly generated LTL formulas. In particular the bisimulation algorithm outperforms Mona on 87% of the instances. The reverse-SFA algorithm also times out often and is in general slower than the bisimulation algorithm. Explicitly representing positive Boolean expressions is generally

[4] In an early version of this experiment we compared against the tool Alaska [17], which checks for satisfiability of LTL-F formulas using a BDD-based variant of alternating automata. After observing that Mona consistently outperformed Alaska, we decided to only report the comparison against Mona. We do not compare against non-symbolic automata libraries as these would not support large alphabets. Moreover, most libraries only support NFAs [5], which would force us to choose a way to encode the LTL formulas into NFAs.

[5] The S-AFA to S-FA conversion is doubly exponential in the worse case (this bound is tight), but constructing an S-FA recognizing the reverse language of an S-AFA yields an automaton that has only exponential size.

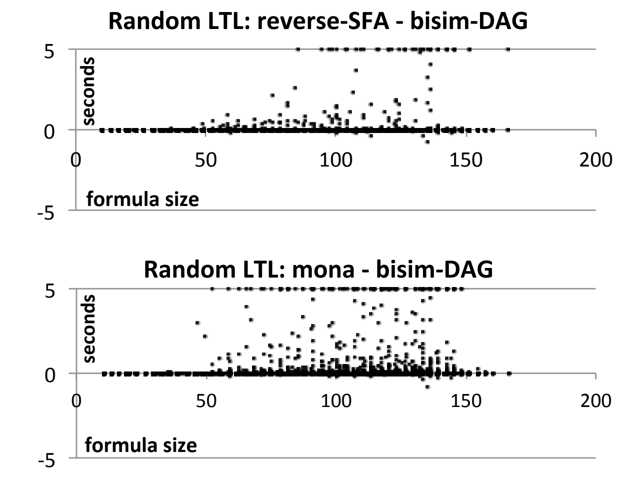

Figure 3. Satisfiability checking for randomly generated LTL formulas.

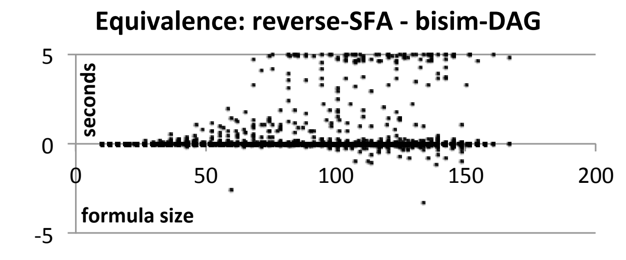

Figure 4. Equivalence checking for randomly generated LTL formulas.

much faster than using BDDs. Even though our algorithm is mostly suited for equivalence and inclusion checking, this experiments illustrates that S-AFAs and our bisimulation technique are a viable solution for checking satisfiability of LTL-F formulas.

6.1.2 Equivalence checking

We evaluate the performance of our algorithm for checking the equivalence of the S-AFAs corresponding to the given LTL-F formulas against slight modifications of such automata. Given an LTL-F formula φ, let A_φ be the corresponding S-AFA. For each formula φ in the benchmark, we compute a variation A'_φ of the automaton A_φ by randomly flipping the acceptance condition of one of the states in A_φ. We then measure the cost of checking the equivalence of A_φ with A'_φ. In this experiment, we do not consider Mona as there is no natural way to generate a formula corresponding to the modified S-AFAs. Given the *random* nature of our experiment we only consider the 10,000 randomly generated LTL formulas from [12]. The results are depicted in Figure 4. The graph shows the time difference between the reverse-SFA algorithm and the bisimulation algorithm. Again we only plot the case in which positive Boolean expressions are represented explicitly rather than with BDDs.The bottom graph in the Figure 5 shows the number times each solver timed out.

Results The bisimulation algorithm is again faster than the reverse-SFA algorithm and times out in only one instance. In this experiment representing positive Boolean expressions using BDDs incurs in more timeouts than those observed using the reverse-SFA algorithm. We believe that the slow performance of BDDs is due to the many substitution operations—a slow operation for BDDs—needed by the equivalence algorithm.

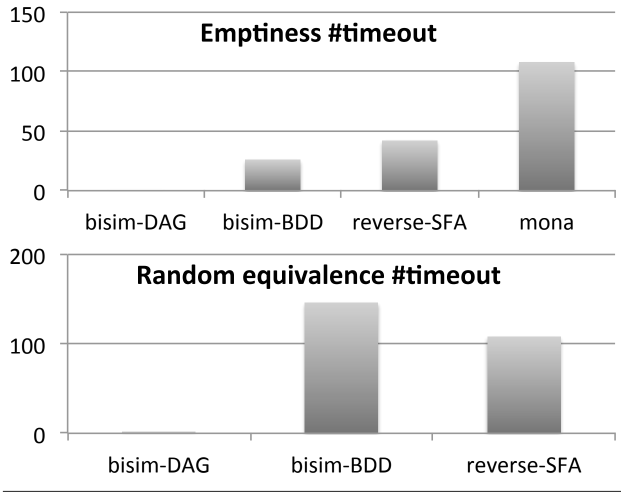

Figure 5. Number of timeouts for different solver for checking emptiness and equivalence of all the random LTL formulas.

6.2 Boolean combinations of regular expressions

Regular expressions are ubiquitous and their analysis is fundamental in many domains, from deep-packet inspection in networking [29] to static analysis of string-manipulating programs [2]. Classic automata techniques for analyzing regular expressions are often limited by two factors. First, regular expressions operate over large alphabets, making most existing automata representations impractical. Second, Boolean combinations of regular expressions produce automata with large number of states. Since our model promises to attenuate both these problems, in this experiment we ask the following question. *Is our technique more efficient than classic automata techniques when analyzing properties involving Boolean combinations of regular expressions?* In this experiment we use *unions of intervals* to represent predicates in the alphabet theory. This representation naturally models character classes which often appear in regular expressions—e.g., [a-z0-9]. We stress how the ability to easily change the representation of the underlying alphabet illustrates the versatility of S-AFAs.

We first describe the set of considered regular expression and then describe what experiments we evaluate our techniques on.

Benchmark formulas We consider regular expressions from [reg]. This site contains more than 3,000 crowd-sourced regular expressions for tasks such as email filtering, phone number detection, and URL detection. From these expressions, we isolate 75 regular expressions for email filtering and consider Boolean combinations of them. For each experiment we set the timeout at 20 seconds.

6.2.1 Equivalence checking

We evaluate the performance of our algorithm on the task of checking equivalence of intersected regular expressions. This experiment is inspired by the application described in Section 2, where we were interested in detecting whether a newly added spam filter is already present in a set of existing spam filters. We identify sets of regular expressions $\{r_1, \ldots, r_n\}$ such that $L(r_1) \cap \ldots \cap L(r_n) \neq \emptyset$ and $n \in \{3, 4, 5\}$. Here $L(r)$ denotes the set of strings accepted by r. For each set of expressions we then measure the time required to check whether $L(r_1) \cap \ldots \cap L(r_{n-1}) = L(r_1) \cap \ldots \cap L(r_n)$. We only illustrate instances on which at least one solver doesn't timeout and, for each value of n, we stop the generation at 6,000 sets. In this experiment we compare our algorithm against the classic decision procedure based on finite automata intersection, determinization, and equivalence. Concretely, for each set of regu-

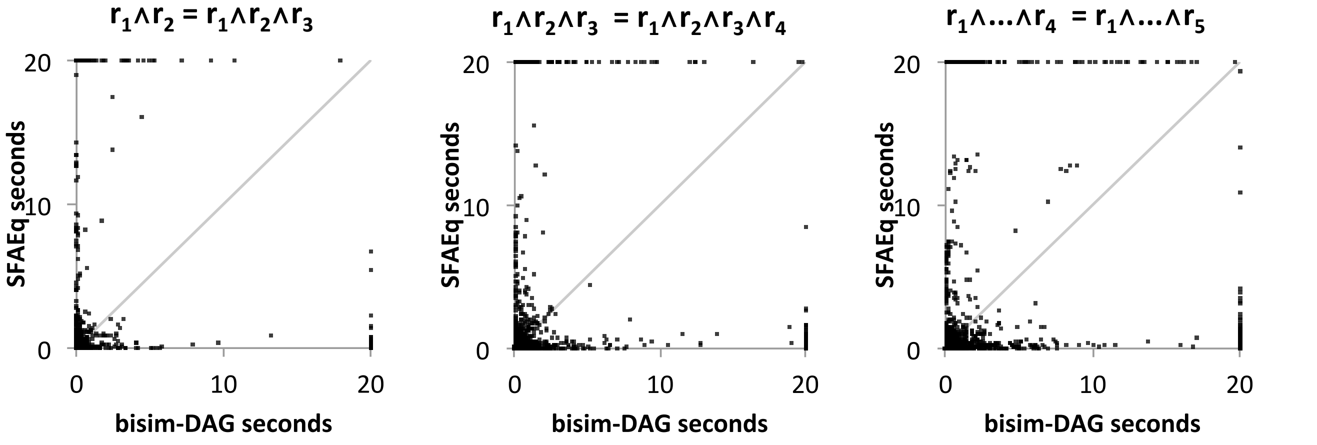

Figure 6. Running times for checking $L(r_1) \cap \ldots \cap L(r_n) = L(r_1) \cap \ldots \cap L(r_n) \cap L(r_{n+1})$ for $n \in \{2, 3, 4\}$.

lar expressions we build the corresponding (non-alternating) symbolic finite automata (S-FAS), then perform automata intersection, determinize the two automata corresponding to the left-hand and right-hand sides of the equality, and finally use Hopcroft-Karp algorithm [21] to check the equivalence of the resulting automata. [6] When measuring the running time of this procedure we consider the cumulative cost of all operations.

For the bisimulation experiment, we take advantage of the fact that our algorithm can also check the equivalence of two configurations of the same S-AFA. In particular, instead of building two S-AFAs and check whether they accept the same language, we only build one S-AFA and check the equivalence of the two state configurations corresponding to $L(r_1) \cap \ldots \cap L(r_n)$ and $L(r_1) \cap \ldots \cap L(r_n) \cap L(r_{n+1})$. Given a set of regular expressions $\{r_1, \ldots, r_n\}$, let $\{A_1, \ldots, A_n\}$ be the corresponding nondeterministic S-FAS (notice that an S-FA is also an S-AFA) with corresponding initial states $\{q_0^1, \ldots, q_0^n\}$. After building the intersected S-AFA $A = A_1 \cap \ldots \cap A_n$ with initial state $q_0^1 \wedge \ldots \wedge q_0^n$, we check whether the state $q_0^1 \wedge \ldots \wedge q_0^n$ is equivalent to the state $q_0^1 \wedge \ldots \wedge q_0^{n-1}$.

Figure 6 illustrates the runtime of the classic algorithm based on S-FA equivalence (SFAEq) and our bisimulation based equivalence procedure (bisim-DAG) for sets of regular expressions of different sizes. Given the slowdown observed in Section 6.1, in this experiment we do not measure the performance of the bisimulation equivalence that uses BDD to represent positive Boolean expressions. A point above the diagonal indicates an instance in which the bisimulation algorithm is *faster* than performing S-FA equivalence. The first three entries of Figure 7 show the number of instances on which each algorithm timed out.

Results Remarkably, the two algorithms have orthogonal performances and none of the two strictly outperforms the other one. The bisimulation algorithm is faster than the S-FA algorithm on approximately 45% of the instances, but it times out 803 fewer times. The results are truly encouraging and show that each algorithm has its benefits.

6.2.2 Forced equivalence checking

In the previous experiment almost all tuples return inequivalent as the result. To better appreciate the cost of checking equivalence

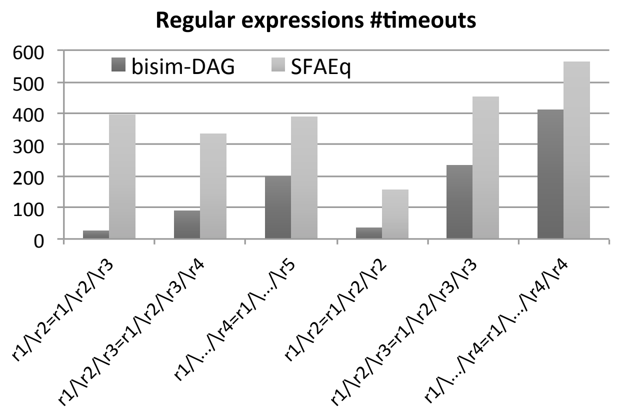

Figure 7. Timeout distribution for regular expression equivalence checks.

in the case in which both sides of the equality describe the same language we perform the following experiment. For every tuple $\{r_1, \ldots, r_n\}$ ($n \in \{2, 3, 4\}$) such that $L(r_1) \cap \ldots \cap L(r_n) \neq \emptyset$, we measure the time required to check whether $L(r_1) \cap \ldots \cap L(r_n) = L(r_1) \cap \ldots \cap L(r_n) \cap L(r_n)$, where one of the regular expression is added twice.[7] We only illustrate instances on which at least one solver doesn't timeout and, for each value of n, we stop the generation at 6,000 sets. The results are showed in Figure 8. Figure 8 illustrates the running times of this experiment, while the last three entries of Figure 7 show the number of instances on which each algorithm timed out.

To better asses the effectiveness of the congruence in pruning the space of explored states we also measured how many states each algorithm explored during the equivalence check. The results are shown in Figure 9.

Results Again, the two algorithms have orthogonal performances and none of the two strictly outperforms the other one. However,

[6] For a fair comparison we consider symbolic finite automata instead of classical automata, as the latter would suffer from the large alphabet size.

[7] In our implementation, to make the comparison fair and make the computation of the bisimulation non-trivial, we create an isomorphic copy of the automaton for the last formula rather than re-using it. Thus, the bisimulation formula corresponding to the equivalence of the initial states has the shape $q_1 \wedge \ldots \wedge q_n \iff q_1 \wedge \ldots \wedge q_n \wedge q'_n$, where q'_n is the start state of an automaton disjoint from the one for q_n.

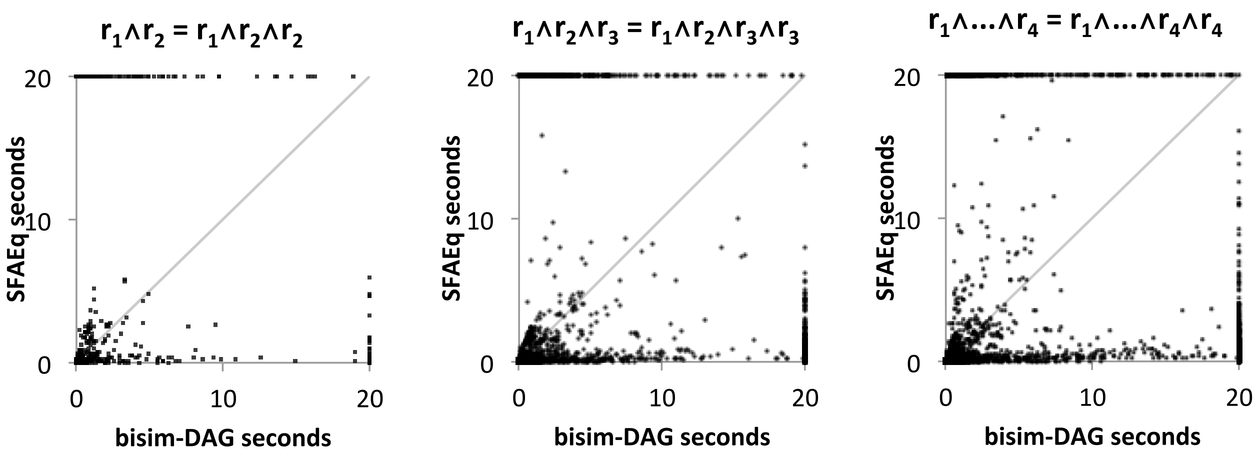

Figure 8. Running times for checking $L(r_1) \cap \ldots \cap L(r_n) = L(r_1) \cap \ldots \cap L(r_n) \cap L(r_n)$ for $n \in \{2, 3, 4\}$.

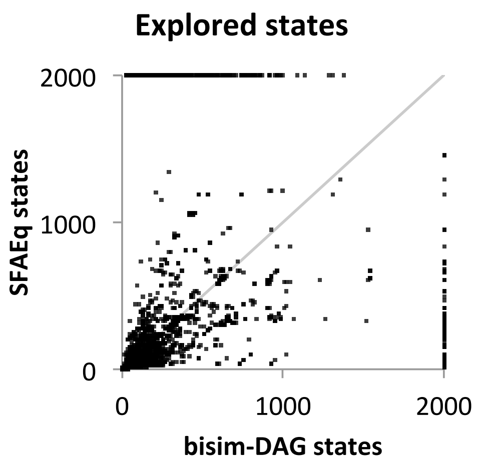

Figure 9. Number of explored states of each algorithm when performing checks of the form $L(r_1) \cap \ldots \cap L(r_n) = L(r_1) \cap \ldots \cap L(r_n) \cap L(r_n)$. The graph shows the cumulative result for $n \in \{2, 3, 4\}$. The points at coordinate 2,000 indicate timeouts.

this time, the bisimulation algorithm is faster than the S-FA algorithm on approximately 20% of the instances, but it times out 492 fewer times. We find quite remarkable how the two algorithms have complementary performances. When we combine more than 30,000 equivalence checks performed in this section, we have 2,292 instances in which the bisimulation algorithm terminated while the S-FA equivalence timed out and 997 instances in which the S-FA equivalence terminated but the bisimulation algorithm timed out. This experiment shows that our algorithm is useful for analyzing regular expressions and it will be a great addition to practical regular expression engines. Moreover, we could also appreciate how, in the forced equivalence experiment, the bisimulation procedure explored fewer states than the algorithm for checking S-FA equivalence approximately 46% of the times. These are typically the instances for which the bisimulation algorithm is the faster one.

7. Applications and future directions

The development of the theory of S-AFAs is motivated by several concrete practical problems. Here we discuss six such applications. In each case we illustrate what kind of alphabet theory the automaton operates with, the role of alternation, and how our decision procedure could be adopted.

7.1 Regular expression analysis

We discussed how regular expressions are used in many different contexts. In particular, the problems of checking equivalence and inclusion arise in many practical applications, such as text processing and analysis of string-manipulating programs [2, 33]. In Section 6.2 we empirically showed that S-AFAs are an effective model for checking equivalence of complex combinations of regular expressions. Moreover, unlike classic models like DFAs and NFAs, S-AFAs can use interval arithmetic or BDDs to succinctly represent the complex alphabet structure of real-world regular expressions.

7.2 Model-checking LTL over finite traces

In Section 6.1 we showed how S-AFAs can effectively check satisfiability of LTL formulas interpreted over finite sequences. Part of this success was enabled by the BDD representation of the input alphabet. While we mostly focused on checking emptiness of the S-AFAs generated from the LTL-F formulas, another promising option is that of using our algorithm to model-check transitions systems against LTL-F formulas [10]. Intuitively, one could model a transition system as an S-AFA T and the LTL-F property as another S-AFA P. The model-checking problem then amounts to whether the language of the transitions system T is included in the language of the property P. We can reduce this check to an equivalence query of the form $T \vee P = P$. Exploring whether our algorithm could improve on the state of the art model checkers is an intriguing research direction.

7.3 SMT solving with sequences

SMT solvers such as [16] have drastically changed the world of programming languages and turned previously unsolvable problems into feasible ones. The recent interest in verifying programs operating over sequences has created a need for extending existing SMT solving techniques to handle sequences over complex theories [32]. Solvers that are able to handle strings, typically do so by building automata and then performing complex operations over such automata [24]. As we advocated in this paper, this approach is doomed to incur into a state blow-up as the SMT formulas typically contain many Boolean operations. Moreover, existing solvers only handle strings over finite and small alphabets [24]. S-AFAs and the techniques presented in this paper have the potential to impact the way in which such solvers for SMT are built as they can support sequences over arbitrary alphabet theory and do not incur into state explosion. Investigating ways to integrate S-AFAs into SMT solvers for sequences is an exciting and challenging research direction we plan to pursue in the future.

7.4 Automata learning

The first algorithm for efficiently learning deterministic finite automata was introduced by Dana Angluin [3]. Since then, automata learning has found many applications in program verification and program synthesis [1, 36]. Recently, algorithms have been proposed to learn alternating finite automata [4]. Similarly to Angluin's original algorithm, the one proposed in [4] uses a learning model in which the learner, who is trying to learn an automaton for describing an unknown target language L, is allowed to query the teachers and ask two types of queries.

Equivalence queries Given an automaton A, does A accepts exactly the language L.

Membership queries Given a string s, is the string s in the language L.

In a model in which the language L is specified as an automaton itself (this is for example the case in [6]) being able to efficiently perform such queries is crucial for efficient learning. The equivalence algorithm proposed in this paper has the potential to make the problem of learning alternating automata more efficient.

7.5 S-AFA minimization

Automata minimization is a crucial operation for mitigating state space explosion. For example, one of the "secrets" that made the monadic second-order logic solver Mona [19] practical was to eagerly perform automata minimization when building intermediate automata corresponding to sub-formulas of the input formula. Recently, minimization and state reduction algorithms have been proposed for S-FAS [13] and nondeterministic automata [26]. However, the problem of reducing the state space of alternating automata has not been explored yet. Interestingly, when performing the LTL-F to S-AFA reduction during our evaluation in Section 6.1 we observed that many states of the produced S-AFAs were redundant because the same sub-formulas (e.g., Xa) appeared multiple times in the input LTL-F formula. While we solved this problem by hashing sub-formulas, it would have been better to automatically detect such redundancies using a state reduction algorithm for S-AFAs. Automata minimization and bisimulations are tightly related concepts and we are confident that the notions presented in this paper can be of aid in designing efficient algorithms for reducing the number of states in S-AFAs and in general in alternating automata.

7.6 List manipulating programs

Since S-AFAs support efficient Boolean operations (Section 3.1), they offer an intruiging possibility of acting as the predicates of an effective Boolean algebra. That is, we can consider S-AFAs over an effective Boolean algebra where characters are strings and the predicates are themselves S-AFAs. Naturally, this process can be iterated indefinitely (strings of strings of strings of strings ...).

One compelling use-case for such technology is in expressing and verifying correctness properties of list-manipulating programs. For example, a sparse representation of a 10×10 matrix of integers might be represented by type of the form:

type pos = $\{p : \text{int} \mid 0 \leq p < 10\}$
type row = $\{r : (\text{pos} \times \text{int}) \text{ list} \mid \text{length}(r) \leq 10\}$
type matrix = $\{m : (\text{pos} \times \text{row}) \text{ list} \mid \text{length}(m) \leq 10\}$

One could imagine that the decision procedure presented in Section 4 might be useful for type checking in a language with such a rich type system (e.g., consider the problem of checking that matrix addition preserves the sparse matrix invariants).

8. Related work

Automata with predicates The concept of automata with predicates instead of concrete symbols was first mentioned in [35] and was first discussed in [30] in the context of natural language processing. S-FAS were then formally introduced in [33] and then adopted in [34] with a focus on security analysis of sanitizers. D'Antoni et al. studied alternation for symbolic *tree* automata, but all their techniques are based on classic algorithms for eliminating alternation and reductions to non-alternating automata [14]. Our approach is different from the one in [14] as our equivalence procedure does not need to build the S-FA corresponding to a S-AFA. 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 [22]. 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 the alternating automata generated by LTL formulas.

Alternating automata Alternation is an old concept in computer science and and the notion of alternating automata dates back to the 80s [7, 9]. Vardi recognized the potential of such a model in program verification, in spite of their high theoretical complexities [31]. Alaska was the one of the first practical implementation of alternating automata [17]. In Alaska, the alphabet and the set of states are both represented using bit-vectors and this allows to model the search space using BDD. While this representation is somewhat similar to ours, S-AFAs are much more modular because they support arbitrary alphabets and alphabet representation (not just bit-vectors and BDDs) and arbitrary state representations (again not just BDDs). Alaska performs state-space reduction using antichains while checking AFA emptiness. As observed by Bonchi and Pous [5], bisimulation up to congruence strictly subsumes antichain reduction.

Equivalence using bisimulation up to Notions of bisimulations similar to the one presented in this paper have been studied in the past [8, 20, 25] in different context that did not related to language equivalence. The paper that most relates to ours is by Bonchi and Pous [5], where the idea of bisimulation up to congruence is used to check equivalence and inclusion of non-deterministic finite state automata (NFAs). There are two main differences with the ideas we presented here. First, generalizing bisimulation up to congruence to alternating finite automata is absolutely non-trivial. While in the case of NFAs, the congruence closure can be computed in polynomial time with a simple saturation algorithm, this is not the case for AFAs. In fact, for AFAs the problem becomes NP-complete. Second, the techniques in [5] are described for NFAs operating over finite alphabets and most of the presented examples operate over alphabets of size two. We demonstrated a technique for extending the bisimulation up to congruence technique to symbolic and potentially infinite alphabets using a representative enumeration algorithm reminiscent of AllSat solving. In fact, our technique can also be used to extend the techniques proposed in [5] to arbitrary domains.

9. Conclusion

We presented *symbolic alternating finite automata*, S-AFAs, a succinct and efficient automaton model for describing sets of finite sequences over large and potentially infinite alphabets. We also introduced an algorithm for checking the equivalence of two S-AFAs based on bisimulation up to congruence and showed how this algorithm often outperforms other automaton models in two concrete applications. First, our algorithm can be used to efficiently check satisfiability of linear temporal logic formulas over finite traces. Second, our algorithm can efficiently checking whether different Boolean combinations of regular expressions describe the same language.

References

[reg] Regular expression library. `http://www.regexlib.com/`.

[1] R. Alur, P. Černý, P. Madhusudan, and W. Nam. Synthesis of interface specifications for java classes. *SIGPLAN Not.*, 40(1):98–109, Jan. 2005. ISSN 0362-1340. doi: 10.1145/1047659.1040314. URL `http://doi.acm.org/10.1145/1047659.1040314`.

[2] R. Alur, L. D'Antoni, and M. Raghothaman. Drex: A declarative language for efficiently evaluating regular string transformations. *SIGPLAN Not.*, 50(1):125–137, Jan. 2015. ISSN 0362-1340. doi: 10.1145/2775051.2676981. URL `http://doi.acm.org/10.1145/2775051.2676981`.

[3] D. Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, Nov. 1987. ISSN 0890-5401. doi: 10.1016/0890-5401(87)90052-6. URL `http://dx.doi.org/10.1016/0890-5401(87)90052-6`.

[4] D. Angluin, S. Eisenstat, and D. Fisman. Learning regular languages via alternating automata. In *Proceedings of the 24th International Conference on Artificial Intelligence*, IJCAI'15, pages 3308–3314. AAAI Press, 2015. ISBN 978-1-57735-738-4. URL `http://dl.acm.org/citation.cfm?id=2832581.2832710`.

[5] F. Bonchi and D. Pous. Checking nfa equivalence with bisimulations up to congruence. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, pages 457–468, New York, NY, USA, 2013. ACM.

[6] M. Botinčan and D. Babić. Sigma*: Symbolic Learning of Input-Output Specifications. In *POPL'13: Proceedings of the 40th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 443–456, New York, NY, USA, 2013. ACM.

[7] J. A. Brzozowski and E. L. Leiss. On equations for regular languages, finite automata, and sequential networks. *Theor. Comput. Sci.*, 10: 19–35, 1980. doi: 10.1016/0304-3975(80)90069-9. URL `http://dx.doi.org/10.1016/0304-3975(80)90069-9`.

[8] D. Caucal. Graphes canoniques de graphes algébriques. *ITA*, 24:339–352, 1990.

[9] A. K. Chandra, D. C. Kozen, and L. J. Stockmeyer. Alternation. *J. ACM*, 28(1):114–133, Jan. 1981. ISSN 0004-5411. doi: 10.1145/322234.322243. URL `http://doi.acm.org/10.1145/322234.322243`.

[10] A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri. Nusmv: A new symbolic model verifier. In *Proceedings of the 11th International Conference on Computer Aided Verification*, CAV '99, pages 495–499, London, UK, UK, 1999. Springer-Verlag. ISBN 3-540-66202-2. URL `http://dl.acm.org/citation.cfm?id=647768.733923`.

[11] M. Dalla Preda, R. Giacobazzi, A. Lakhotia, and I. Mastroeni. Abstract symbolic automata: Mixed syntactic/semantic similarity analysis of executables. *SIGPLAN Not.*, 50(1):329–341, Jan. 2015. ISSN 0362-1340. doi: 10.1145/2775051.2676986. URL `http://doi.acm.org/10.1145/2775051.2676986`.

[12] M. Daniele, F. Giunchiglia, and M. Y. Vardi. Improved automata generation for linear temporal logic. In *Proceedings of the 11th International Conference on Computer Aided Verification*, CAV '99, pages 249–260, London, UK, UK, 1999. Springer-Verlag. ISBN 3-540-66202-2. URL `http://dl.acm.org/citation.cfm?id=647768.733938`.

[13] L. D'Antoni and M. Veanes. Minimization of symbolic automata. *SIGPLAN Not.*, 49(1):541–553, Jan. 2014. ISSN 0362-1340. doi: 10.1145/2578855.2535849. URL `http://doi.acm.org/10.1145/2578855.2535849`.

[14] L. D'antoni, M. Veanes, B. Livshits, and D. Molnar. Fast: A transducer-based language for tree manipulation. *ACM Trans. Program. Lang. Syst.*, 38(1):1:1–1:32, Oct. 2015. ISSN 0164-0925. doi: 10.1145/2791292. URL `http://doi.acm.org/10.1145/2791292`.

[15] G. De Giacomo and M. Y. Vardi. Linear temporal logic and linear dynamic logic on finite traces. In *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence*, IJCAI '13, pages 854–860. AAAI Press, 2013. ISBN 978-1-57735-633-2. URL `http://dl.acm.org/citation.cfm?id=2540128.2540252`.

[16] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 3-540-78799-2, 978-3-540-78799-0. URL `http://dl.acm.org/citation.cfm?id=1792734.1792766`.

[17] M. De Wulf, L. Doyen, N. Maquet, and J. F. Raskin. *Antichains: Alternative Algorithms for LTL Satisfiability and Model-Checking*, pages 63–77. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. ISBN 978-3-540-78800-3. doi: 10.1007/978-3-540-78800-3_6. URL `http://dx.doi.org/10.1007/978-3-540-78800-3_6`.

[18] A. Harding. Symbolic strategy synthesis for games with LTL winning conditions. Technical report, 2005.

[19] 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.

[20] Y. Hirshfeld, M. Jerrum, and F. Moller. A polynomial algorithm for deciding bisimilarity of normed context-free processes. *Theor. Comput. Sci.*, 158(1-2):143–159, May 1996. ISSN 0304-3975. doi: 10.1016/0304-3975(95)00064-X. URL `http://dx.doi.org/10.1016/0304-3975(95)00064-X`.

[21] J. E. Hopcroft and R. M. Karp. A linear algorithm for testing equivalence of finite automata. Technical report, Cornell University, 1971.

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

[23] G. Le Guernic, A. Banerjee, T. Jensen, and D. A. Schmidt. *Automata-Based Confidentiality Monitoring*, pages 75–89. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007. ISBN 978-3-540-77505-8. doi: 10.1007/978-3-540-77505-8_7. URL `http://dx.doi.org/10.1007/978-3-540-77505-8_7`.

[24] T. Liang, A. Reynolds, C. Tinelli, C. Barrett, and M. Deters. *A DPLL(T) Theory Solver for a Theory of Strings and Regular Expressions*, pages 646–662. Springer International Publishing, Cham, 2014. ISBN 978-3-319-08867-9. doi: 10.1007/978-3-319-08867-9_43. URL `http://dx.doi.org/10.1007/978-3-319-08867-9_43`.

[25] D. Lucanu and G. Roşu. *Circular Coinduction with Special Contexts*, pages 639–659. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. ISBN 978-3-642-10373-5. doi: 10.1007/978-3-642-10373-5_33. URL `http://dx.doi.org/10.1007/978-3-642-10373-5_33`.

[26] R. Mayr and L. Clemente. Advanced automata minimization. *SIGPLAN Not.*, 48(1):63–74, Jan. 2013. ISSN 0362-1340. doi: 10.1145/2480359.2429079. URL `http://doi.acm.org/10.1145/2480359.2429079`.

[27] K. L. McMillan. Applying sat methods in unbounded symbolic model checking. In *Proceedings of the 14th International Conference on Computer Aided Verification*, CAV '02, pages 250–264, London, UK, UK, 2002. Springer-Verlag. ISBN 3-540-43997-8. URL `http://dl.acm.org/citation.cfm?id=647771.734421`.

[28] K. Y. Rozier and M. Y. Vardi. *LTL Satisfiability Checking*, pages 149–167. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007. ISBN 978-3-540-73370-6. doi: 10.1007/978-3-540-73370-6_11. URL `http://dx.doi.org/10.1007/978-3-540-73370-6_11`.

[29] R. Smith, C. Estan, S. Jha, and S. Kong. Deflating the big bang: Fast and scalable deep packet inspection with extended finite automata. In *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication*, SIGCOMM '08, pages 207–218, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-175-0. doi: 10.1145/1402958.1402983. URL `http://doi.acm.org/10.1145/1402958.1402983`.

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

[31] M. Y. Vardi. Alternating automata and program verification. In *Computer Science Today: Recent Trends and Developments*, pages 471–485. 1995. doi: 10.1007/BFb0015261. URL `http://dx.doi.org/10.1007/BFb0015261`.

[32] M. Veanes, N. Bjørner, and L. De Moura. Symbolic automata constraint solving. In *Proceedings of the 17th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, LPAR'10, pages 640–654, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-16241-X, 978-3-642-16241-1. URL `http://dl.acm.org/citation.cfm?id=1928380.1928425`.

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

[34] M. Veanes, P. Hooimeijer, B. Livshits, D. Molnar, and N. Bjorner. Symbolic finite state transducers: Algorithms and applications. *SIGPLAN Not.*, 47(1):137–150, Jan. 2012. ISSN 0362-1340. doi: 10.1145/2103621.2103674. URL `http://doi.acm.org/10.1145/2103621.2103674`.

[35] 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.

[36] Y. Yuan, R. Alur, and B. T. Loo. Netegg: Programming network policies by examples. In *Proceedings of the 13th ACM Workshop on Hot Topics in Networks*, HotNets-XIII, pages 20:1–20:7, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3256-9. doi: 10.1145/2670518.2673879. URL `http://doi.acm.org/10.1145/2670518.2673879`.