A Symbolic Decision Procedure for Symbolic Alternating Finite Automata

Loris D'Antoni[1] Zachary Kincaid[2] Fang Wang[1]

[1] *University of Wisconsin-Madison* [2] *Princeton University*

Abstract

We introduce Symbolic Alternating Finite Automata (s-AFA) as a succinct and decidable model for describing sets of finite sequences over arbitrary alphabets. Boolean operations over s-AFAs have linear complexity, which contrasts 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 exploits the power of SAT 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 formulae over finite traces, and 2) checking equivalence of Boolean combinations of regular expressions. Our experiments show that our technique can be beneficial in both applications.

Keywords: Automata, decision procedures

1 Introduction

Programs that operate over sequences are used for text processing [1], program monitoring [17], and deep packet inspection [23]. Efficiently reasoning about these programs is crucial and these recent applications have renewed interest in automata theory, especially in the fields of security and programming languages [4,8,23]. Despite the recent improvements, existing automata-based decision procedures are not as advanced as other decision procedures such as SMT solvers yet.

We illustrate the limitations of classic automata with the following spam detection example. Spam detection is a notoriously hard task and spam filters are continuously modified to handle new malicious behaviours and to relax overly restrictive conditions. While machine learning is the typical choice for spam detection, many companies prefer using custom filters created using regular expressions. The number of 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 complicated and it is therefore desirable to avoid adding redundant filters to the list of processed ones.

Suppose that a spam filter is given as a set of regular expressions $R = \{r_1, \ldots, r_n\}$ such that a string is recognized by the spam filter R if it belongs to the language

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: `www.elsevier.nl/locate/entcs`

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

Fig. 1. Regexes forming a spam scenario.

of each regular expression $r_i \in R$ (e.g., Fig. 1). When a new spam filter $R' = \{r'_1, \ldots, r'_n\}$ is added to the set of all spam filters S, we 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 in W is classified as spam. We can perform these checks using automata operations, but in doing so, we face the following limitations:

(i) The regular expressions operate over a large alphabet (2^{16} characters), making classic automata operations impractical.

(ii) The subsumption check requires repeated Boolean automata operations, which can cause an exponential blow-up in the number of states.

(iii) Checking equivalence of non-deterministic automata is PSpace-complete.

In this paper, we present *symbolic alternating finite automata* together with a decision procedure for equivalence to address these three problems.

Symbolic alphabets: *Symbolic Finite Automata* (s-FA) [26,8], have been proposed to address the problem of handling large alphabets. s-FAs 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 elements of 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.

Alternation: While s-FAs provide a way to cope with large alphabets, they have the same state complexities as 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 formulae that describe the set of target states. For example, when an s-AFA is in state p and reads a string $w = a_1 \ldots a_n$, the transition $p \xrightarrow{[a-z]} q_1 \vee (q_2 \wedge q_3)$ specifies that w is accepted from state p if a_1 is a lower-case letter and the string $a_2 \ldots a_n$ is accepted from state q_1 or it is accepted from both q_2 and q_3. Using alternation, s-AFAs obtain Boolean operations with linear complexity, which is in sharp contrast with the quadratic intersection and exponential complementation of s-FAs.

Equivalence via bisimulation up to: 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 of two s-AFAs and show that the procedure is effective in practice. The algorithm extends the *bisimulation up to congruence* technique recently proposed by Bonchi and Pous [4] for solving the language equivalence problem for nondeterministic finite automata. 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 classic algorithm for checking equivalence of deterministic finite automata is a member of the family that employs a *bisimulation up to* technique [15,4]. Rather than computing 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) $a_1 \; R \; b_1$ 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 extend 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 set of verification and security benchmarks. First, we used s-AFAs to check satisfiability of more than 10,000 LTL formulae appearing in [12]. Second, we used s-AFAs to check equivalence of Boolean combinations of complex regular expressions. Our experiments show that s-AFAs and our bisimulation technique often outperform existing techniques.

Contributions. In summary our contributions are:

- *Symbolic Alternating Finite Automata,* s-AFA*s*, a model for describing languages of strings operating over large and potentially infinite alphabets and for which Boolean operations have linear time complexity (Section 2).
- An *algorithm for checking equivalence of two* s-AFA*s*, which integrates bisimulation up to congruence with propositional SAT solving (Section 3).
- A *modular, open-source implementation of* s-AFA*s and their algorithms*, and a comprehensive evaluation on more than 40,000 benchmarks (Section 4).

2 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, that there is a function $\mathit{Witness}(\varphi)$ that computes for any satisfiable predicate φ a character $a \in \mathfrak{D}$ such that $a \in [\![\varphi]\!]$, and that for every domain element $a \in \mathfrak{D}$ and predicate φ, it is decidable to check whether $a \in [\![\varphi]\!]$.

The following are examples of effective Boolean algebras.

$\mathbf{2^{bv}}^k$ is the powerset algebra whose domain is the finite set $\textsc{bv}k$, 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.

SMT^σ is the decision procedure for a theory (e.g., linear integer arithmetic) over a sort σ (e.g., the integers). Ψ is the set of all formulae $\varphi(x)$ 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 [5,6]. Nested combinations of existential and universal choices can be represented by positive Boolean formulae. Formally, for any set X, we use $\mathcal{B}^+(X)$ to denote the set of *positive Boolean formulae over* X, quotiented by logical equivalence (that is, the set of Boolean formulae built from *true*, *false*, and the members of X using the binary connectives $\wedge$ and $\vee$, and where logically equivalent formulas are identified).

Definition 2.1 A *symbolic alternating finite automaton* (s-AFA) is a tuple $M = \langle \mathcal{A}, Q, p_0, F, \Delta \rangle$ where $\mathcal{A}$ is an effective Boolean algebra, Q is a finite set of states, $p_0 \in \mathcal{B}^+(Q)$ represents M's initial state, $F \subseteq Q$ 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:

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

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

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 formulae over Q. The state $x \wedge y$ 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.

2.1 Boolean operations on s-AFA*s*

One of the critical features of s-AFAs is that Boolean operations have linear complexity in the number of states. For example, the language inclusion problem can be reduced in linear time to checking language equivalence—i.e., $\mathcal{L}(M) \subseteq \mathcal{L}(M')$ iff $\mathcal{L}(M) \cup \mathcal{L}(M') = \mathcal{L}(M')$. The constructions for s-AFA union, intersection, and complementation follow the standard ones for AFAs, with the exception that s-AFA complementation (like s-FA complementation) requires a preprocessing step. For the sake of completeness, we 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:

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

(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 [18] for computing all satisfiable assignments to a propositional formula, and the one in [25] for computing satisfiable Boolean combinations of a set of predicates, and also the representative character enumeration algorithm in Section 3). Normalization may (in the worst case) cause an exponential blow-up 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, its complement is constructed as $\overline{M} \triangleq \langle \mathcal{A}, Q, \overline{p_0}, Q \setminus F, \{\langle x, \varphi, \overline{p}\rangle : \langle x, \varphi, p\rangle \in \Delta\rangle$, where $\bar{\cdot}$ denotes the "De Morganization" transformation that replaces every $\wedge$ with $\vee$ (and vice versa).

Procedure *Normalize*(M)
  Input : s-AFA $M = \langle \mathcal{A}, Q, p_0, F, \Delta \rangle$
  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 Δ'
  return $\langle \mathcal{A}, Q, p_0, F, \Delta' \rangle$

Algorithm 1: Normalization algorithm for s-AFA

2.2 An algebraic view of s-AFA

This section provides an algebraic description of s-AFAs. We use this description in the next section to give a concise presentation of our equivalence checking technique.

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 formulae. Operating (as we do) under the assumption that we do not distinguish between logically equivalent positive Boolean formulae, 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:

$$m(false) \triangleq 0 \qquad m(p \wedge q) \triangleq m(p) \wedge m(q)$$
$$m(true) \triangleq 1 \qquad m(p \vee q) \triangleq m(p) \vee m(q) \ .$$

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 $\mathcal{B}^+(X) \to \mathcal{L}$.[1] 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:

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

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

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

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 $\Delta_w : Q \to \mathcal{B}^+(Q)$ 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 $w \in \mathcal{L}_M(p) \iff F(\Delta_w(p)) = 1$.

3 Equivalence checking for s-AFAs

This section describes an algorithm for checking whether two s-AFAs recognize the same language. Recently, Bonchi and Pous introduced the *bisimulation up to congruence* technique for solving the language equivalence problem for nondeterministic finite automata [4]. 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.

3.1 Bisimulation up to congruence

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

Definition 3.1 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 formulae 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 $a \in \mathfrak{D}_\mathcal{A}$, $\Delta_a(p) \ R \ \Delta_a(q)$.

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

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 (adapted from known results for classical finite automata [19,21]) states the soundness and completeness of using bisimulations to prove language equivalence for s-AFA.

Proposition 3.2 *Let $M = \langle \mathcal{A}, Q, p_0, F, \Delta \rangle$ be an* s-AFA. *For any $p, q \in \mathcal{B}^+(Q)$, we have $\mathcal{L}_M(p) = \mathcal{L}_M(q)$ iff there exists a bisimulation R such that $p\ R\ q$.*

We will now define bisimulation up to congruence for s-AFA*s*.

Definition 3.3 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 3.4 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 3.2 and the following:

Proposition 3.5 *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 formulae 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.

3.2 Congruence checking

We now 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 formulae $p, q \in \mathcal{B}^+(Q)$, 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 3.6 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 $p \in \mathcal{B}^+(Q)$, $m(p) = 1 \iff m \models p$). It is easy to see that m is a model of $\Phi(R)$ if and only if m is consistent with R. Thus, we may read the definition of $cl(R)$ as the set of all pairs $\langle p, q\rangle$ such that for every bounded lattice homomorphism $m : \mathcal{B}^+(Q) \to \mathbf{2}$, if m that is consistent with R (i.e., $m \models \Phi$), then we necessarily have $m(p) = m(q)$ (i.e., $m \models p \iff q$). That is, $cl(R)$ is the *largest* relation such that every bounded lattice homomorphism that is consistent with R is also consistent with $cl(R)$.

The question now is what is the relationship between the congruence closure and the logical closure. We now prove that the two closures are identical. First, a lemma:

Lemma 3.7 *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 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 $c \sqsubseteq a$. Since L is finite, there exists a minimal (w.r.t. $\sqsubseteq$) 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

$$f(e) \triangleq \begin{cases} 1 & \text{if } c \sqsubseteq e \\ 0 & \text{otherwise} \end{cases}$$

One may check that f is a bounded lattice homomorphism with $f(a) = 1$ and $f(b) = 0$. □

Proposition 3.8 *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 3.7, 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 3.8 yields a simple candidate algorithm for the CONGRUENCE problem: simply check whether the pair of positive Boolean formulae 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 3.9 CONGRUENCE *is NP-complete.*

Proof. Membership in NP follows immediately from Proposition 3.8. We prove NP-hardness of CONGRUENCE by giving a polytime reduction from SAT. The key insight is that the relation R can be used to axiomatize negation, so that arbitrary Boolean formulae can be encoded into positive Boolean formulae.

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:

$$R \triangleq \{\langle p \wedge \overline{p}, \mathit{false}\rangle : p \in P\} \cup \{\langle p \vee \overline{p}, \mathit{true}\rangle : p \in P\}$$

Let $\widehat{\varphi}$ be the formula obtained by replacing every negative literal $\neg p$ with $\overline{p}$. We have that φ is satisfiable if and only if $\widehat{\varphi} \not\equiv_R \mathit{false}$. □

Finally, using the technical machinery we have developed in this section, we will re-state and prove Proposition 3.5.

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 exist positive Boolean formulae $p, q \in \mathcal{B}^+(Q)$ 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 $m(\Delta_a(r)) = m(\Delta_a(s))$. □

3.3 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 bisimulation conditions.

The equivalence decision procedure is pictured in Algorithm 2. The idea is simple: given an s-AFA $M = \langle \mathcal{A}, Q, r_0, F, \Delta \rangle$ and two positive Boolean formulae p_0 and q_0, the algorithm attempts to synthesize a bisimulation up to congruence R such that $p_0 \; R \; q_0$ or shows 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: $s.add(\varphi)$ 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 3.6 and Proposition 3.8). We may add a pair $\langle p, q \rangle$ to the relation R by calling $s.add(p \iff q)$ and we may check whether a given pair $\langle p, q \rangle$ belongs to $\equiv_R$ by issuing the query $s.isSat(\neg(p \iff q))$.

The relation R is initialized to the singleton set containing the pair of $\langle p_0, q_0 \rangle$ of formulae 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 3.4. 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 compatibility 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 naïve 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. We may define a set of representative characters as a set of equivalence class representatives of a suitably chosen equivalence relation. Towards this end, for any set of states $S \subseteq Q$, we define an equivalence relation $\simeq_S$ on the set of characters $\mathfrak{D}_\mathcal{A}$ as follows:

$$a \simeq_S b \iff \left(\begin{array}{c} \forall \langle x, \varphi, q \rangle \in \Delta \text{ with } x \in S. \\ a \in [\![\varphi]\!] \iff b \in [\![\varphi]\!] \end{array} \right).$$

Intuitively, we have $a \simeq_S b$ if a and b are indistinguishable by transitions emanating from states in S. Observe that if S is the set of states that appear in p or q, then any set B containing one member of each equivalence class of $\simeq_S$ is a valid choice for a representative set of characters:

(i) B is finite: follows from the fact that Δ is finite.

(ii) B is representative: suppose $\Delta_b(p) \equiv_R \Delta_b(q)$ for all $b \in B$, and let $a \in \mathfrak{D}_{\mathcal{A}}$ – we must prove that $\Delta_a(p) \equiv_R \Delta_a(q)$. Let $c \in B$ such that $a \simeq_S c$. Observe that (by the choice of S and definitions of $\Delta_\cdot$ and $\simeq_S$), we have $\Delta_a(p) = \Delta_c(p)$ and $\Delta_a(q) = \Delta_c(q)$. We have $\Delta_a(p) = \Delta_c(p) \equiv_R \Delta_c(q) = \Delta_a(q)$ and thus $\Delta_a(p) \equiv_R \Delta_a(q)$.

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 [18]. 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 $\simeq_S$:

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

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, $\neg\mathit{class}$ 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:

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

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

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:

(i) ($R = \{\langle v, w \rangle\}$, *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 $\mathit{class} = c \leq 0$ and set $\mathit{chars} \leftarrow \mathit{chars} \wedge \neg(c \leq 0)$. We add $\langle \Delta_0(v), \Delta_0(w) \rangle = \langle x \vee y, z \rangle$ to R and the worklist.

```
1  Procedure IsEquivalent(M, p_0, q_0)
       Input  : s-AFA M = ⟨A, Q, p, F, Δ⟩
                positive Boolean formulae p_0, q_0 ∈ B^+(Q)
       Output: true if L_M(p_0) = L_M(q_0), false otherwise
2      s ← new solver
3      worklist ← [(p_0, q_0)]
4      s.add(p_0 ⟺ q_0)
5      while worklist is not empty do
6          Pick (p, q) off worklist
7          S ← set of states in p or q
           /* ⟦chars⟧ is the set of characters that remain to be
              processed                                          */
8          chars ← ⊤
9          while IsSat(chars) do
10             a ← Witness(chars)
               /* Compute the transition function Δ_a and a predicate
                  representing the equivalence class of a in ≃_S.  */
11             class ← true
12             Δ_a ← λx.false
13             for x ∈ S, ⟨x, φ, q⟩ ∈ Δ do
14                 if a ∈ ⟦φ⟧ then
15                     class ← class ∧ φ
16                     Δ_a(x) ← Δ_a(x) ∨ q
17                 else
18                     class ← class ∧ ¬φ
19             p' ← Δ_a(p)
20             q' ← Δ_a(q)
               /* Remove a's equivalence class from chars        */
21             chars ← chars ∧ ¬class
22             if F(p') ≠ F(q') then
23                 return false
               /* If p' ≢_R q', add ⟨p', q'⟩ to R                 */
24             if s.isSat(¬(p' ⟺ q')) then
25                 Add (p', q') to worklist
26                 s.add(p ⟺ q)
27     return true
```

Algorithm 2: Equivalence algorithm for s-AFAs

(b) ($R = \{\langle v, w\rangle, \langle x \vee y, z\rangle\}$, $\mathit{worklist} = [\langle x \vee y, z\rangle]$, $\mathit{chars} = \neg(c \leq 0)$) We generate 5 as a witness to satisfiability of *chars*. We compute the equivalence class of 5 to be $\mathit{class} = c > 0$ and set $\mathit{chars} \leftarrow \mathit{chars} \wedge \neg(c > 0)$. 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) ($R = \{\langle v, w\rangle, \langle x \vee y, z\rangle\}$, $\mathit{worklist} = [\langle x \vee y, z\rangle]$, $\mathit{chars} = \neg(c \leq 0) \wedge \neg(c > 0)$). We find that *chars* is unsatisfiable and exit the inner loop.

(ii) ($R = \{\langle v, w\rangle, \langle x \vee y, z\rangle\}$, *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.

(iii) ($R = \{\langle v, w\rangle, \langle x \vee y, z\rangle\}$, *worklist* $= [\,]$) Since the worklist is empty, the algorithm terminates: R is a bisimulation up to congruence containing $\langle v, w\rangle$, so $\mathcal{L}_M(v) = \mathcal{L}_M(w)$.

4 Implementation and evaluation

We implemented s-AFAs and their decision procedure in the Java symbolicautomata library (open source and available at https://github.com/lorisdanto/symbolicautomata). The implementation provides an interface for specifying custom Boolean algebras for both the alphabet theory and the positive Boolean formulae over the automaton states and it can be easily integrated with externally specified alphabet theories. To represent the positive Boolean formulae over the automaton states we implemented two algebras: one which simply maintains the explicit Boolean representations of formulae (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 formulae to the congruence closure we use the SAT solver Sat4j [2].

We implemented two 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 an s-AFA $M = \langle \mathcal{A}, Q, p_0, F, \Delta\rangle$, we construct a graph $G_M = \langle V, E\rangle$ with set of vertices $V = Q$, and edges $E = \{\langle x, y\rangle : (x, \varphi, q) \in \Delta \wedge y \in st(q)\}$. 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 formula of M we replace every removed state with *false*. The resulting s-AFA is equivalent to M'.

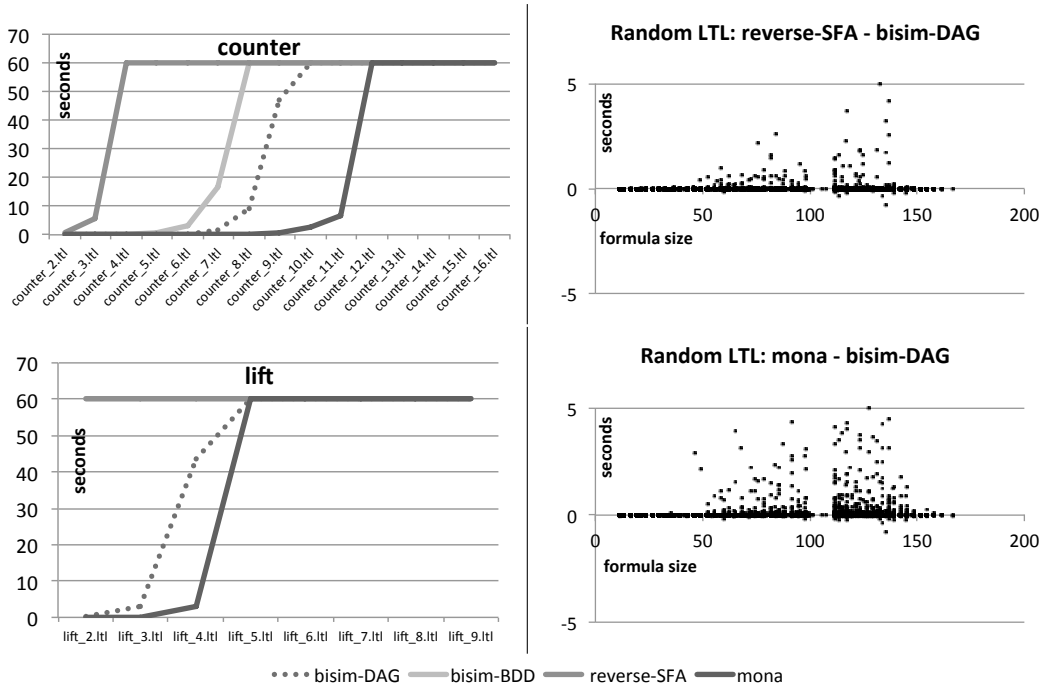


Fig. 2. Satisfiability checking for LTL formulae. Non-random formulae on the left and random formulae from [7] on the right. On the right, each point is the difference (in seconds) between one of the two compared algorithms and bisim-DAG.

Second, for the worklist data structure employed by Algorithm 2 we employ a priority queue, where the priority of a pair $\langle p, q\rangle$ is the sum of the sizes of p and q. This heuristic leverages the intuition that smaller formulae have fewer successors and are also likely to generate better congruences (e.g., the pair $\langle x, y\rangle$ generates a larger congruence than $\langle x \wedge z, y \wedge z\rangle$). [2]

We evaluate our algorithm through two experiments. First, we use s-AFAs to check satisfiability of the LTL formulae appearing in [12] using the semantics of LTL over finite traces from [11] (§ 4.1). Second, we use s-AFAs to check equivalence of Boolean combinations of regular expressions appearing in `http://www.regexlib.com/` (§ 4.2). All experiments were run on an Intel Core i7 2.60GHz, 16GB RAM.

4.1 Satisfiability checking for LTL *over finite traces*

Linear temporal logic (LTL) plays a prominent role in program verification. While the semantics of LTL is typically defined over infinite strings, recently there has been interest in interpreting LTL over finite traces [11]. We use LTL-f to refer to the interpretation of LTL over finite traces. Checking satisfiability of LTL-f formulae is a PSpace-complete problem, but there exists a linear time translation from LTL-f formulae to alternating automata [11]. Since this translation results in an alphabet of size exponential in the number of atomic proposition appearing in the formula, s-AFAs are a promising model for designing decision procedures for

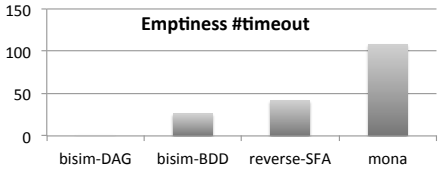
[2] We have separately evaluated the use of a priority queue for our worklist and our tool also has implementations for FIFO and LIFO. The use of a priority queue consistently outperforms the other heuristics. Concretely, on our benchmarks FIFO and LIFO time out approximately twice as often as the priority queue implementation. We will report data only about the priority queue implementation because it is consistently faster than the other two.

LTL-f. Using the translation proposed in [11] we apply our algorithm to checking satisfiability of LTL-f formulae—i.e., by checking equivalence to an empty automaton.

We consider three sets of LTL formulae: 1) counter contains 15 formulae describing counters for which satisfiability is notoriously hard [20]. 2) lift contains 8 parametric formulae describing a lift system of increasing complexity [13]. 3) More than 10,000 random formulae appearing in [7]. These formulae have size varying between 10 and 100, and number of atomic propositions varying between 2 and 4. We set the timeout at 60 and 5 seconds for the non-random and random formulae respectively.

We use the translation from LTL-f to monadic second order logic (MSO) from [11] to compare against Mona [14], a solver for MSO. [3] 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.

We measure the runtimes 1) of Mona (mona); 2) of computing the deterministic automaton accepting the reverse language of the s-AFA and checking its emptiness (reverse-SFA) [4]; 3) of the bisimulation algorithm using a directed acyclic graph (i.e., hash-consed) representation of positive Boolean expressions (bisim-DAG); 4) of the bisimulation algorithm using BDDs to represent positive Boolean expressions (bisim-BDD). The results are depicted in Figure 2. In the plots on the right, a point above 0 means that the bisimulation solver is faster than the other solver. Finally, the number of timeouts for the random formulae are shown on the right.



Results: Mona outperforms the bisimulation on non-random formulae, but it times out for relatively small instances for which our solver does not. Mona is slower than our algorithm on 87% of the random instances and times out more often. The reverse-SFA algorithm also times out often and is in general slower than the bisimulation algorithm. Representing positive Boolean expressions using BDDs is slower and it incurs in more timeouts than those observed using the reverse-SFA algorithm. The slow performance of BDDs is due to the many substitution operations—a slow operation for BDDs—needed by the equivalence algorithm. This experiment illustrates that s-AFAs and our bisimulation technique are a viable solution for checking satisfiability of LTL-f formulae.

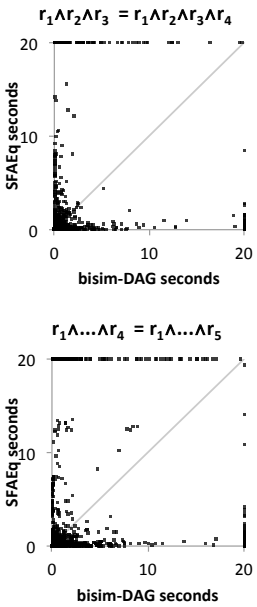
[3] In an early version of this experiment we compared against the tool Alaska [12], which checks for satisfiability of LTL-f formulae using a BDD-based variant of alternating automata. After observing that Mona consistently outperformed Alaska, we decided to only report the comparison against Mona. We do not compare against non-symbolic automata libraries as these would not support large alphabets. Moreover, most libraries only support NFAs [4], which would force us to choose a way to encode the LTL formulae into NFAs. We also do not compare against model checkers such as NuSMV and Spin, since they only support LTL over infinite traces.

[4] The s-AFA to s-FA conversion is doubly exponential in the worst 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.

4.2 Boolean combinations of regular expressions

Regular expressions are ubiquitous and their analysis is fundamental in many domains, from deep-packet inspection in networking [23] to static analysis of string-manipulating programs [1]. In these domains, classic automata techniques fall short because large alphabets and Boolean operations produce automata with large number of states and transitions. In this experiment, we ask *is our technique more efficient than existing automata techniques when analyzing Boolean combinations of regular expressions?* We use *unions of intervals* to represent predicates in the alphabet theory; this representation naturally models character classes—e.g., `[a-z0-9]`. The ability to easily change the representation of the underlying alphabet illustrates the versatility of s-AFAs.

We consider regular expressions from `http://www.regexlib.com/`, a site containing more than 3,000 crowd-sourced regular expressions for tasks such as email filtering, phone number detection, and URL detection. We isolate the first 75 regular expressions for email filtering and consider Boolean combinations of them. For each experiment we set the timeout at 20 seconds.



Equivalence checking: We evaluate the performance of our algorithm on checking equivalence of intersected regular expressions (figure on the right). This experiment is inspired by the spam-filtering application described in the introduction. 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. We measure the time required to check whether $L(r_1) \cap \ldots \cap L(r_n) = L(r_1) \cap \ldots \cap L(r_{n+1})$. We only illustrate instances on which at least one solver does not timeout and, for each value of n, we generate the first 6,000 combinations, in lexicographic order. We compare our algorithm (bisim-DAG) against the classic decision procedure based on finite automata intersection and equivalence (SFAEq). Concretely, for each set of regular expressions we build the corresponding (non-alternating) symbolic finite automata (s-FAs), perform automata intersection, and then use a lazy version of Hopcroft-Karp algorithm [15] to check the equivalence of the resulting automata while lazily determinizing them on the fly. For a fair comparison we consider symbolic finite automata instead of classic automata, as the latter would suffer from the large alphabet size. In fact, the explicit alphabet implementation of NFA equivalence proposed by Bonchi and Pous [4], throws a stack overflow exception when handling the large alphabets required by regular expressions.

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 checking 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 $L(r_1) \cap \ldots \cap$

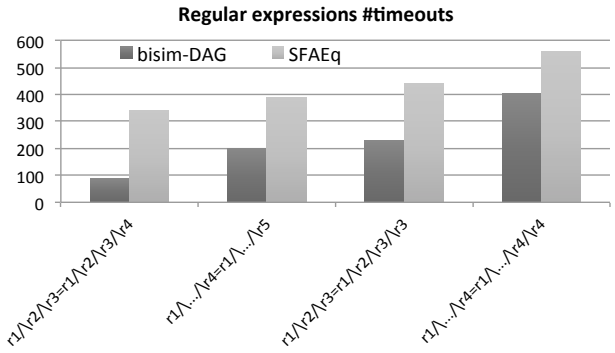
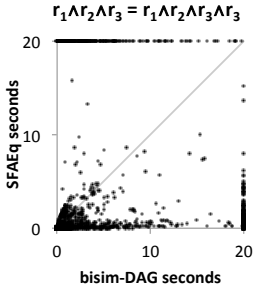
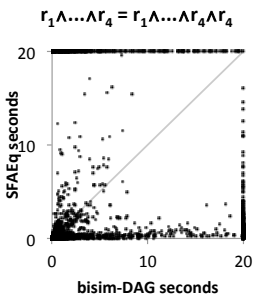


Fig. 6. Timeout distribution for regular expression equivalence checks.

$L(r_n) \cap L(r_{n+1})$. Given a set of regular expressions $\{r_1, \ldots, r_{n+1}\}$, let $\{A_1, \ldots, A_n\}$ be the corresponding non-deterministic 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 $A = A_1 \cap \ldots \cap A_{n+1}$ with initial state $q_0^1 \wedge \ldots \wedge q_0^{n+1}$, we check whether the state $q_0^1 \wedge \ldots \wedge q_0^n$ is equivalent to the state $q_0^1 \wedge \ldots \wedge q_0^{n+1}$.

The results are shown on the right. A point above the diagonal is an instance where bisim-DAG is *faster*. The first two entries of Figure 6 show the number of instances on which each algorithm timed out.





Forced equivalence checking: In the previous experiment almost all tuples return inequivalent as the result. To appreciate the cost of verifying equivalence rather than refuting it, we perform the following experiment. For every tuple $\{r_1, \ldots, r_n\}$ $(n \in \{3, 4\})$ 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 expressions is added twice. [5] We only illustrate instances on which at least one solver does not timeout (figure on the right). The last two entries in Figure 6 show the number of timeouts for each algorithm.

Results: The two algorithms have orthogonal performances and neither of them strictly outperforms the other one. The bisimulation is faster than the s-FA algorithm on approximately 40% of the instances, but it times out on 809 (11%) fewer instances. This is mostly due to the prohibitive cost of computing the intersected s-FA. This experiment shows that our algorithm is useful for analyzing regular expressions and it will be a great addition to regular expression engines. Finally, we also measured how many states each algorithm explored in the forced equivalence experiment. The bisimulation procedure explored fewer states than the other algorithm 46% of the times. These are typically instances for which the bisimulation algorithm is the faster one.

[5] 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.

5 Related work

Automata with predicates: The concept of automata with predicates was first mentioned in [27]. s-FAs were then formally introduced in [26]. 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 [10]. Our approach is different from the one in [10] as our equivalence procedure does not need to build the s-FA corresponding to an s-AFA. D'Antoni and Veanes also designed algorithms for computing forward bisimulations of non-deterministic s-FAs [9]. However, their algorithm does not use congruences and cannot handle alternation. The Mona implementation [16] 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 [16]. 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 formulae.

Alternating automata: Alternation is a classic concept in computer science and and the notion of alternating automata dates back to the 80s [5,6]. Vardi recognized the potential of such a model in program verification, in spite of their high theoretical complexities [24]. Alaska was one of the first practical implementations of alternating automata [12]. 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 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 shown by Bonchi and Pous [4], bisimulation up to congruence strictly subsumes antichain reduction.

Equivalence using bisimulation up to: The use of bisimulations to prove equivalence between languages of finite and infinite words has long been established [19,21]. Bisimulation up-to was introduced as a technique for simplifying bisimulation-based equivalence proofs for concurrent processes [22], and has since been studied extensively in the setting of concurrency theory, coalgebra, and formal language theory — many references can be found in [3]. The paper that most relates to ours is by Bonchi and Pous [4], 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, our work extends bisimulation up to congruence to alternating finite automata. The extension is non-trivial: while Bonchi and Pous showed that for NFAs, membership within the congruence closure can be computed in polynomial time, we showed that for AFAs the problem is NP-complete, and gave an pratical algorithm that leverages propositional satisfiability solvers. Second, the techniques in [4] apply to NFAs operating over finite alphabets (and most of the presented examples operate over alphabets of size two). We showed how to extend the technique to symbolic alphabets using representative character enumeration. Our method can also be used to extend the NFA equivalence algorithm [4] to symbolic alphabets.

References

[1] R. Alur, L. D'Antoni, and M. Raghothaman. Drex: A declarative language for efficiently evaluating regular string transformations. In *POPL*, pages 125–137, 2015.

[2] D. L. Berre and A. Parrain. The sat4j library, release 2.2. *JSAT*, 7(2-3):59–6, 2010.

[3] F. Bonchi, D. Petrişan, D. Pous, and J. Rot. A general account of coinduction up-to. *Acta Informatica*, 54(2):127–190, 2017.

[4] F. Bonchi and D. Pous. Checking NFA equivalence with bisimulations up to congruence. In *POPL*, pages 457–468, 2013.

[5] J. A. Brzozowski and E. L. Leiss. On equations for regular languages, finite automata, and sequential networks. *Theor. Comput. Sci.*, 10:19–35, 1980.

[6] A. K. Chandra, D. C. Kozen, and L. J. Stockmeyer. Alternation. *J. ACM*, 28(1):114–133, Jan. 1981.

[7] M. Daniele, F. Giunchiglia, and M. Y. Vardi. Improved automata generation for linear temporal logic. In *CAV*, pages 249–260, 1999.

[8] L. D'Antoni and M. Veanes. Minimization of symbolic automata. In *POPL*, pages 541–553, 2014.

[9] L. D'Antoni and M. Veanes. Forward bisimulations for nondeterministic symbolic finite automata. In *TACAS*, 2017.

[10] 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.

[11] G. De Giacomo and M. Y. Vardi. Linear temporal logic and linear dynamic logic on finite traces. In *IJCAI*, pages 854–860, 2013.

[12] M. De Wulf, L. Doyen, N. Maquet, and J. F. Raskin. Antichains: Alternative algorithms for LTL satisfiability and model-checking. In *TACAS*, pages 63–77, 2008.

[13] A. Harding. Symbolic strategy synthesis for games with LTL winning conditions, Tech. Report 2005.

[14] J. Henriksen, J. Jensen, M. Jørgensen, N. Klarlund, B. Paige, T. Rauhe, and A. Sandholm. Mona: Monadic second-order logic in practice. In *TACAS*, 1995.

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

[16] N. Klarlund, A. Møller, and M. I. Schwartzbach. MONA implementation secrets. *IJFCS*, 13(4):571–586, 2002.

[17] G. Le Guernic, A. Banerjee, T. Jensen, and D. A. Schmidt. Automata-based confidentiality monitoring. In *Advances in Computer Science - ASIAN*, pages 75–89, 2007.

[18] K. L. McMillan. Applying SAT methods in unbounded symbolic model checking. In *CAV*, pages 250–264, 2002.

[19] D. Park. Concurrency and automata on infinite sequences. In *TCS*, pages 167–183, 1981.

[20] K. Y. Rozier and M. Y. Vardi. LTL satisfiability checking. In *Model Checking Software: 14th International SPIN Workshop*, pages 149–167, 2007.

[21] J. J. M. M. Rutten. Automata and coinduction (an exercise in coalgebra). In *CONCUR*, pages 194–218, 1998.

[22] D. Sangiorgi. On the bisimulation proof method. *Mathematical Structures in Computer Science*, 8(05):447–479, 1998.

[23] R. Smith, C. Estan, S. Jha, and S. Kong. Deflating the big bang: Fast and scalable deep packet inspection with extended finite automata. In *SIGCOMM*, pages 207–218, 2008.

[24] M. Y. Vardi. Alternating automata and program verification. In *Computer Science Today: Recent Trends and Developments*, pages 471–485. 1995.

[25] M. Veanes, N. Bjørner, and L. De Moura. Symbolic automata constraint solving. In *LPAR*, pages 640–654, 2010.

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

[27] 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.