Symbolic Register Automata

Loris D'Antoni[1], Tiago Ferreira[2], Matteo Sammartino[2](✉),
and Alexandra Silva[2]

[1] University of Wisconsin–Madison, Madison, WI 53706-1685, USA
loris@cs.wisc.edu

[2] University College London, Gower Street, London WC1E 6BT, UK
me@tiferrei.com, {m.sammartino,a.silva}@ucl.ac.uk

Abstract. Symbolic Finite Automata and Register Automata are two orthogonal extensions of finite automata motivated by real-world problems where data may have unbounded domains. These automata address a demand for a model over large or infinite alphabets, respectively. Both automata models have interesting applications and have been successful in their own right. In this paper, we introduce Symbolic Register Automata, a new model that combines features from both symbolic and register automata, with a view on applications that were previously out of reach. We study their properties and provide algorithms for emptiness, inclusion and equivalence checking, together with experimental results.

1 Introduction

Finite automata are a ubiquitous formalism that is simple enough to model many real-life systems and phenomena. They enjoy a large variety of theoretical properties that in turn play a role in practical applications. For example, finite automata are closed under Boolean operations, and have decidable emptiness and equivalence checking procedures. Unfortunately, finite automata have a fundamental limitation: they can only operate over finite (and typically small) alphabets. Two *orthogonal* families of automata models have been proposed to overcome this: *symbolic automata* and *register automata*. In this paper, we show that these two models can be combined yielding a new powerful model that can cover interesting applications previously out of reach for existing models.

Symbolic finite automata (SFAs) allow transitions to carry predicates over rich first-order alphabet theories, such as linear arithmetic, and therefore extend classic automata to operate over infinite alphabets [12]. For example, an SFA can define the language of all lists of integers in which the first and last elements are positive integer numbers. Despite their increased expressiveness, SFAs enjoy the same closure and decidability properties of finite automata—e.g., closure under Boolean operations and decidable equivalence and emptiness.

This work was partially funded by NSF Grants CCF-1763871, CCF-1750965, a Facebook TAV Research Award, the ERC starting grant Profoundnet (679127) and a Leverhulme Prize (PLP-2016-129). See [10] for the full version of this paper.

I. Dillig and S. Tasiran (Eds.): CAV 2019, LNCS 11561, pp. 3–21, 2019.
https://doi.org/10.1007/978-3-030-25540-4_1

Register automata (RA) support infinite alphabets by allowing input characters to be stored in registers during the computation and to be compared against existing values that are already stored in the registers [17]. For example, an RA can define the language of all lists of integers in which all numbers appearing in even positions are the same. RAs do not have some of the properties of finite automata (e.g., they cannot be determinized), but they still enjoy many useful properties that have made them a popular model in static analysis, software verification, and program monitoring [15].

In this paper, we combine the best features of these two models—first order alphabet theories and registers—into a new model, *symbolic register automata* (SRA). SRAs are strictly more expressive than SFAs and RAs. For example, an SRA can define the language of all lists of integers in which the first and last elements are positive rational numbers and all numbers appearing in even positions are the same. This language is not recognizable by either an SFA nor by an RA.

While other attempts at combining symbolic automata and registers have resulted in undecidable models with limited closure properties [11], we show that SRAs enjoy the same closure and decidability properties of (non-symbolic) register automata. We propose a new application enabled by SRAs and implement our model in an open-source automata library.

In summary, our contributions are:

- Symbolic Register Automata (SRA): a new automaton model that can handle complex alphabet theories while allowing symbols at arbitrary positions in the input string to be compared using equality (Sect. 3).
- A thorough study of the properties of SRAs. We show that SRAs are closed under intersection, union and (deterministic) complementation, and provide algorithms for emptiness and forward (bi)simulation (Sect. 4).
- A study of the effectiveness of our SRA implementation on handling regular expressions with back-references (Sect. 5). We compile a set of benchmarks from existing regular expressions with back-references (e.g., `(\d)[a-z]*\1`) and show that SRAs are an effective model for such expressions and existing models such as SFAs and RAs are not. Moreover, we show that SRAs are more efficient than the `java.util.regex` library for matching regular expressions with back-references.

2 Motivating Example

In this section, we illustrate the capabilities of symbolic register automata using a simple example. Consider the regular expression r_p shown in Fig. 1a. This expression, given a sequence of product descriptions, checks whether the products have the same code and lot number. The reader might not be familiar with some of the unusual syntax of this expression. In particular, r_p uses two back-references `\1` and `\2`. The semantics of this construct is that the string matched by the regular expression for `\1` (resp. `\2`) should be exactly the string that matched the subregular expression r appearing between the first (resp. second)

```
C:(.{3}) L:(.) D:[^\s]+( C:\1 L:\2 D:[^\s]+)+
```

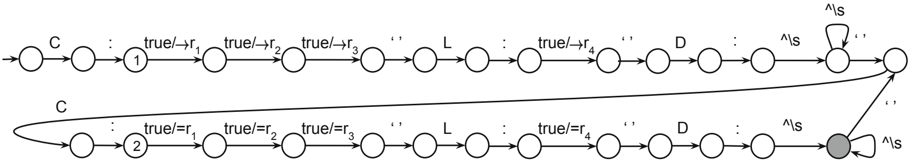
(a) Regular expression r_p (with back-reference).

```
C:X4a L:4 D:bottle C:X4a L:4 D:jar
```

(b) Example text matched by r_p.

```
C:X4a L:4 D:bottle C:X5a L:4 D:jar
```

(c) Example text *not* matched by r_p.



(d) Snippets of a symbolic register automaton A_p corresponding to r_p.

Fig. 1. Regular expression for matching products with same code and lot number—i.e., the characters of C and L are the same in all the products.

two parenthesis, in this case `(.{3})` (resp. `(.)`). Back-references allow regular expressions to check whether the encountered text is the same or is different from a string/character that appeared earlier in the input (see Figs. 1b and c for examples of positive and negative matches).

Representing this complex regular expression using an automaton model requires addressing several challenges. The expression r_p:

1. operates over large input alphabets consisting of upwards of 2^{16} characters;
2. uses complex character classes (e.g., `\s`) to describe different sets of characters in the input;
3. adopts back-references to detect repeated strings in the input.

Existing automata models do not address one or more of these challenges. Finite automata require one transition for each character in the input alphabet and blow-up when representing large alphabets. Symbolic finite automata (SFA) allow transitions to carry predicates over rich structured first-order alphabet theories and can describe, for example, character classes [12]. However, SFAs cannot directly check whether a character or a string is repeated in the input. An SFA for describing the regular expression r_p would have to store the characters after `C:` directly in the states to later check whether they match the ones of the second product. Hence, the smallest SFA for this example would require billions of states! Register automata (RA) and their variants can store characters in registers during the computation and compare characters against values already stored in the registers [17]. Hence, RAs can check whether the two products have the same code. However, RAs only operate over unstructured infinite alphabets and cannot check, for example, that a character belongs to a given class.

The model we propose in this paper, *symbolic register automata* (SRA), combines the best features of SFAs and RAs—first-order alphabet theories and registers—and can address all the three aforementioned challenges. Figure 1d shows a snippet of a symbolic register automaton A_p corresponding to r_p. Each transition in A_p is labeled with a predicate that describes what characters can trigger the transition. For example, `^\s` denotes that the transition can be triggered by any non-space character, `L` denotes that the transition can be triggered by the character `L`, and `true` denotes that the transition can be triggered by any character. Transitions of the form $\varphi/\!\rightarrow r_i$ denote that, if a character x satisfies the predicate φ, the character is then stored in the register r_i. For example, the transition out of state 1 reads any character and stores it in register r_1. Finally, transitions of the form $\varphi/\!= r_i$ are triggered if a character x satisfies the predicate φ and x is the same character as the one stored in r_i. For example, the transition out of state 2 can only be triggered by the same character that was stored in r_1 when reading the transition out state 1—i.e., the first characters in the product codes should be the same.

SRAs are a natural model for describing regular expressions like r_p, where capture groups are of bounded length, and hence correspond to finitely-many registers. The SRA A_p has fewer than 50 states (vs. more than 100 billion for SFAs) and can, for example, be used to check whether an input string matches the given regular expression (e.g., monitoring). More interestingly, in this paper we study the closure and decidability properties of SRAs and provide an implementation for our model. For example, consider the following regular expression r_{pC} that only checks whether the product codes are the same, but not the lot numbers:

```
C:(.{3}) L:. D:[^\s]+( C:\1 L:. D:[^\s]+)+
```

The set of strings accepted by r_{pC} is a superset of the set of strings accepted by r_p. In this paper, we present simulation and bisimulation algorithms that can check this property. Our implementation can show that r_p subsumes r_{pC} in 25 s and we could not find other tools that can prove the same property.

3 Symbolic Register Automata

In this section we introduce some preliminary notions, we define symbolic register automata and a variant that will be useful in proving decidability properties.

Preliminaries. An *effective Boolean algebra* $\mathcal{A}$ is a tuple $(\mathfrak{D}, \varPsi, [\![\_]\!], \bot, \top, \wedge, \vee, \neg)$, where: $\mathfrak{D}$ is a set of domain elements; $\varPsi$ is a set of predicates closed under the Boolean connectives and $\bot, \top \in \varPsi$. The denotation function $[\![\_]\!] \colon \varPsi \to 2^{\mathfrak{D}}$ is such that $[\![\bot]\!] = \emptyset$ and $[\![\top]\!] = \mathfrak{D}$, for all $\varphi, \psi \in \varPsi$, $[\![\varphi \vee \psi]\!] = [\![\varphi]\!] \cup [\![\psi]\!]$, $[\![\varphi \wedge \psi]\!] = [\![\varphi]\!] \cap [\![\psi]\!]$, and $[\![\neg\varphi]\!] = \mathfrak{D} \setminus [\![\varphi]\!]$. For $\varphi \in \varPsi$, we write $\mathsf{isSat}(\varphi)$ whenever $[\![\varphi]\!] \neq \emptyset$ and say that φ is *satisfiable*. $\mathcal{A}$ is *decidable* if isSat is decidable. For each $a \in \mathfrak{D}$, we assume predicates $\mathsf{atom}(a)$ such that $[\![\mathsf{atom}(a)]\!] = \{a\}$.

Example 1. The theory of linear integer arithmetic forms an effective BA, where $\mathfrak{D} = \mathbb{Z}$ and $\varPsi$ contains formulas $\varphi(x)$ in the theory with one fixed integer variable. For example, $\mathsf{div_k} := (x \;\mathsf{mod}\; k) = 0$ denotes the set of all integers divisible by k.

Notation. Given a set S, we write $\mathcal{P}(S)$ for its powerset. Given a function $f\colon A \to B$, we write $f[a \mapsto b]$ for the function such that $f[a \mapsto b](a) = b$ and $f[a \mapsto b](x) = f(x)$, for $x \neq a$. Analogously, we write $f[S \mapsto b]$, with $S \subseteq A$, to map multiple values to the same b. The *pre-image* of f is the function $f^{-1}\colon \mathcal{P}(B) \to \mathcal{P}(A)$ given by $f^{-1}(S) = \{a \mid \exists b \in S\colon b = f(a)\}$; for readability, we will write $f^{-1}(x)$ when $S = \{x\}$. Given a relation $\mathcal{R} \subseteq A \times B$, we write $a\mathcal{R}b$ for $(a, b) \in \mathcal{R}$.

Model Definition. Symbolic register automata have transitions of the form:

$$p \xrightarrow{\varphi/E,I,U} q$$

where p and q are states, φ is a predicate from a fixed effective Boolean algebra, and E, I, U are subsets of a fixed finite set of registers R. The intended interpretation of the above transition is: an input character a can be read in state q if (i) $a \in [\![\varphi]\!]$, (ii) the content of all the registers in E is *equal* to a, and (iii) the content of all the registers in I is *different* from a. If the transition succeeds then a is stored into all the registers U and the automaton moves to q.

Example 2. The transition labels in Fig. 1d have been conveniently simplified to ease intuition. These labels correspond to full SRA labels as follows:

$$\varphi/{\to}r \implies \varphi/\emptyset, \emptyset, \{r\} \qquad \varphi/{=}r \implies \varphi/\{r\}, \emptyset, \emptyset \qquad \varphi \implies \varphi/\emptyset, \emptyset, \emptyset\ .$$

Given a set of registers R, the transitions of an SRA have labels over the following set: $L_R = \Psi \times \{(E, I, U) \in \mathcal{P}(R) \times \mathcal{P}(R) \times \mathcal{P}(R) \mid E \cap I = \emptyset\}$. The condition $E \cap I = \emptyset$ guarantees that register constraints are always satisfiable.

Definition 1 (Symbolic Register Automaton). *A* symbolic register automaton *(SRA) is a 6-tuple* $(R, Q, q_0, v_0, F, \Delta)$*, where* R *is a finite set of* registers*,* Q *is a finite set of* states*,* $q_0 \in Q$ *is the* initial *state,* $v_0\colon R \to \mathcal{D} \cup \{\sharp\}$ *is the* initial register assignment *(if* $v_0(r) = \sharp$*, the register* r *is considered* empty*),* $F \subseteq Q$ *is a finite set of* final *states, and* $\Delta \subseteq Q \times L_R \times Q$ *is the* transition relation*. Transitions* $(p, (\varphi, \ell), q) \in \Delta$ *will be written as* $p \xrightarrow{\varphi/\ell} q$*.*

An SRA can be seen as a finite description of a (possibly infinite) labeled transition system (LTS), where states have been assigned concrete register values, and transitions read a single symbol from the potentially infinite alphabet. This so-called *configuration LTS* will be used in defining the semantics of SRAs.

Definition 2 (Configuration LTS). *Given an SRA* $\mathcal{S}$*, the* configuration *LTS* $\mathsf{CLTS}(\mathcal{S})$ *is defined as follows. A* configuration *is a pair* (p, v) *where* $p \in Q$ *is a state in* $\mathcal{S}$ *and a* $v\colon R \to \mathcal{D} \cup \{\sharp\}$ *is* register assignment*;* (q_0, v_0) *is called the* initial configuration*; every* (q, v) *such that* $q \in F$ *is a* final *configuration. The set of transitions between configurations is defined as follows:*

$$\frac{p \xrightarrow{\varphi/E,I,U} q \in \Delta \qquad E \subseteq v^{-1}(a) \quad I \cap v^{-1}(a) = \emptyset}{(p, v) \xrightarrow{a} (q, v[U \mapsto a]) \in \mathsf{CLTS}(\mathcal{S})}$$

Intuitively, the rule says that a SRA transition from p can be instantiated to one from (p, v) that reads a when the registers containing the value a, namely $v^{-1}(a)$, satisfy the constraint described by E, I (a is contained in registers E but not in I). If the constraint is satisfied, all registers in U are assigned a.

A *run* of the SRA $\mathcal{S}$ is a sequence of transitions in $\mathsf{CLTS}(\mathcal{S})$ starting from the initial configuration. A configuration is *reachable* whenever there is a run ending up in that configuration. The *language* of an SRA $\mathcal{S}$ is defined as

$$\mathscr{L}(\mathcal{S}) := \{a_1 \dots a_n \in \mathcal{D}^n \mid \exists (q_0, v_0) \xrightarrow{a_1} \dots \xrightarrow{a_n} (q_n, v_n) \in \mathsf{CLTS}(\mathcal{S}), q_n \in F\}$$

An SRA $\mathcal{S}$ is *deterministic* if its configuration LTS is; namely, for every word $w \in \mathcal{D}^\star$ there is at most one run in $\mathsf{CLTS}(\mathcal{S})$ spelling w. Determinism is important for some application contexts, e.g., for runtime monitoring. Since SRAs subsume RAs, nondeterministic SRAs are strictly more expressive than deterministic ones, and language equivalence is undecidable for nondeterministic SRAs [27].

We now introduce the notions of *simulation* and *bisimulation* for SRAs, which capture whether one SRA behaves "at least as" or "exactly as" another one.

Definition 3 ((Bi)simulation for SRAs). *A simulation $\mathcal{R}$ on SRAs $\mathcal{S}_1$ and $\mathcal{S}_2$ is a binary relation $\mathcal{R}$ on configurations such that $(p_1, v_1)\mathcal{R}(p_2, v_2)$ implies:*

- *if $p_1 \in F_1$ then $p_2 \in F_2$;*
- *for each transition $(p_1, v_1) \xrightarrow{a} (q_1, w_1)$ in $\mathsf{CLTS}(\mathcal{S}_1)$, there exists a transition $(p_2, v_2) \xrightarrow{a} (q_2, w_2)$ in $\mathsf{CLTS}(\mathcal{S}_2)$ such that $(q_1, w_1)\mathcal{R}(q_2, w_2)$.*

A simulation $\mathcal{R}$ is a bisimulation *if $\mathcal{R}^{-1}$ is a also a simulation. We write $\mathcal{S}_1 \prec \mathcal{S}_2$ (resp. $\mathcal{S}_1 \sim \mathcal{S}_2$) whenever there is a simulation (resp. bisimulation) $\mathcal{R}$ such that $(q_{01}, v_{01})\mathcal{R}(q_{02}, v_{02})$, where (q_{0i}, v_{0i}) is the initial configuration of $\mathcal{S}_i$, for $i = 1, 2$.*

We say that an SRA is *complete* whenever for every configuration (p, v) and $a \in \mathcal{D}$ there is a transition $(p, v) \xrightarrow{a} (q, w)$ in $\mathsf{CLTS}(\mathcal{S})$. The following results connect similarity and language inclusion.

Proposition 1. *If $\mathcal{S}_1 \prec \mathcal{S}_2$ then $\mathscr{L}(\mathcal{S}_1) \subseteq \mathscr{L}(\mathcal{S}_2)$. If $\mathcal{S}_1$ and $\mathcal{S}_2$ are deterministic and complete, then the other direction also holds.*

It is worth noting that given a deterministic SRA we can define its *completion* by adding transitions so that every value $a \in \mathcal{D}$ can be read from any state.

Remark 1. RAs and SFAs can be encoded as SRAs on the same state-space:

- An RA is encoded as an SRA with all transition guards $\top$;
- an SFA can be encoded as an SRA with $R = \emptyset$, with each SFA transition $p \xrightarrow{\varphi} q$ encoded as $p \xrightarrow{\varphi/\emptyset,\emptyset,\emptyset} q$. Note that the absence of registers implies that the CLTS always has finitely many configurations.

SRAs are *strictly more expressive* than both RAs and SFAs. For instance, the language $\{n_0 n_1 \dots n_k \mid n_0 = n_k, \mathsf{even}(n_i), n_i \in \mathbb{Z}, i = 1, \dots, k\}$ of finite sequences of even integers where the first and last one coincide, can be recognized by an SRA, but not by an RA or by an SFA.

Boolean Closure Properties. SRAs are closed under intersection and union. Intersection is given by a standard product construction whereas union is obtained by adding a new initial state that mimics the initial states of both automata.

Proposition 2 (Closure under intersection and union). *Given SRAs $\mathcal{S}_1$ and $\mathcal{S}_2$, there are SRAs $\mathcal{S}_1 \cap \mathcal{S}_2$ and $\mathcal{S}_1 \cup \mathcal{S}_2$ such that $\mathscr{L}(\mathcal{S}_1 \cap \mathcal{S}_2) = \mathscr{L}(\mathcal{S}_1) \cap \mathscr{L}(\mathcal{S}_2)$ and $\mathscr{L}(\mathcal{S}_1 \cup \mathcal{S}_2) = \mathscr{L}(\mathcal{S}_1) \cup \mathscr{L}(\mathcal{S}_2)$.*

SRAs in general are not closed under complementation, because RAs are not. However, we still have closure under complementation for a subclass of SRAs.

Proposition 3. *Let $\mathcal{S}$ be a complete and deterministic SRA, and let $\overline{\mathcal{S}}$ be the SRA defined as $\mathcal{S}$, except that its final states are $Q \setminus F$. Then $\mathscr{L}(\overline{\mathcal{S}}) = \mathcal{D}^\star \setminus \mathscr{L}(\mathcal{S})$.*

4 Decidability Properties

In this section we will provide algorithms for checking determinism and emptiness for an SRA, and (bi)similarity of two SRAs. Our algorithms leverage *symbolic* techniques that use the finite syntax of SRAs to indirectly operate over the underlying configuration LTS, which can be infinite.

Single-Valued Variant. To study decidability, it is convenient to restrict register assignments to *injective* ones on non-empty registers, that is functions $v \colon R \to \mathcal{D} \cup \{\sharp\}$ such that $v(r) = v(s)$ and $v(r) \neq \sharp$ implies $r = s$. This is also the approach taken for RAs in the seminal papers [17,27]. Both for RAs and SRAs, this restriction does not affect expressivity. We say that an SRA is *single-valued* if its initial assignment v_0 is injective on non-empty registers. For single-valued SRAs, we only allow two kinds of transitions:

Read transition: $p \xrightarrow{\varphi/r^{=}} q$ triggers when $a \in [\![\varphi]\!]$ and a is already stored in r.

Fresh transition: $p \xrightarrow{\varphi/r^{\bullet}} q$ triggers when the input $a \in [\![\varphi]\!]$ and a is *fresh*, i.e., is not stored in any register. After the transition, a is stored into r.

SRAs and their single-valued variants have the same expressive power. Translating single-valued SRAs to ordinary ones is straightforward:

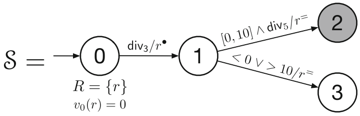
$$p \xrightarrow{\varphi/r^{=}} q \implies p \xrightarrow{\varphi/\{r\},\emptyset,\emptyset} q \qquad\qquad p \xrightarrow{\varphi/r^{\bullet}} q \implies p \xrightarrow{\varphi/\emptyset,R,\{r\}} q$$

The opposite translation requires a state-space blow up, because we need to encode register equalities in the states.

Theorem 1. *Given an SRA $\mathcal{S}$ with n states and r registers, there is a single-valued SRA $\mathcal{S}'$ with $\mathcal{O}(nr^r)$ states and $r+1$ registers such that $\mathcal{S} \sim \mathcal{S}'$. Moreover, the translation preserves determinism.*

Normalization. While our techniques are inspired by analogous ones for non-symbolic RAs, SRAs present an additional challenge: they can have arbitrary predicates on transitions. Hence, the values that each transition can read, and thus which configurations it can reach, depend on the history of past transitions and their predicates. This problem emerges when checking reachability and similarity, because a transition may be *disabled* by particular register values, and so lead to unsound conclusions, a problem that does not exist in register automata.

Example 3. Consider the SRA below, defined over the BA of integers.



All predicates on transitions are satisfiable, yet $\mathscr{L}(\mathcal{S}) = \emptyset$. To go from 0 to 1, $\mathcal{S}$ must read a value n such that $\mathsf{div}_3(n)$ and $n \neq 0$ and then n is stored into r. The transition from 1 to 2 can only happen if the content of r also satisfies $\mathsf{div}_5(n)$ and $n \in [0, 10]$. However, there is no n satisfying $\mathsf{div}_3(n) \wedge n \neq 0 \wedge \mathsf{div}_5(n) \wedge n \in [0, 10]$, hence the transition from 1 to 2 never happens.

To handle the complexity caused by predicates, we introduce a way of *normalizing* an SRA to an equivalent one that *stores additional information about input predicates*. We first introduce some notation and terminology.

A register abstraction θ for $\mathcal{S}$, used to "keep track" of the domain of registers, is a family of predicates indexed by the registers R of $\mathcal{S}$. Given a register assignment v, we write $v \models \theta$ whenever $v(r) \in [\![\theta_r]\!]$ for $v(r) \neq \sharp$, and $\theta_r = \bot$ otherwise. Hereafter we shall only consider "meaningful" register abstractions, for which there is at least one assignment v such that $v \models \theta$.

With the contextual information about register domains given by θ, we say that a transition $p \xrightarrow{\varphi/\ell} q \in \Delta$ is *enabled by* θ whenever it has at least an instance $(p, v) \xrightarrow{a} (q, w)$ in $\mathsf{CLTS}(\mathcal{S})$, for all $v \models \theta$. Enabled transitions are important when reasoning about reachability and similarity.

Checking whether a transition has at least one realizable instance in the CLTS is difficult in practice, especially when $\ell = r^\bullet$, because it amounts to checking whether $[\![\varphi]\!] \setminus \mathsf{img}(v) \neq \emptyset$, for all injective $v \models \theta$.

To make the check for enabledness practical we will use minterms. For a set of predicates Φ, a *minterm* is a minimal satisfiable Boolean combination of all predicates that occur in Φ. Minterms are the analogue of atoms in a complete atomic Boolean algebra. E.g. the set of predicates $\Phi = \{x > 2, x < 5\}$ over the theory of linear integer arithmetic has minterms $\mathsf{mint}(\Phi) = \{x > 2 \wedge x < 5,\ \neg x > 2 \wedge x < 5,\ x > 2 \wedge \neg x < 5\}$. Given $\psi \in \mathsf{mint}(\Phi)$ and $\varphi \in \Phi$, we will write $\varphi \sqsubset \psi$ whenever φ appears non-negated in ψ, for instance $(x > 2) \sqsubset (x > 2 \wedge \neg x < 5)$. A crucial property of minterms is that they do not overlap, i.e., $\mathsf{isSat}(\psi_1 \wedge \psi_2)$ if and only if $\psi_1 = \psi_2$, for ψ_1 and ψ_2 minterms.

Lemma 1 (Enabledness). *Let θ be a register abstraction such that θ_r is a minterm, for all $r \in R$. If φ is a minterm, then $p \xrightarrow{\varphi/\ell} q$ is enabled by θ iff:*

(1) if $\ell = r^{=}$, then $\varphi = \theta_r$; *(2) if $\ell = r^{\bullet}$, then $|[\![\varphi]\!]| > \mathscr{E}(\theta, \varphi)$, where $\mathscr{E}(\theta, \varphi) = |\{r \in R \mid \theta_r = \varphi\}|$ is the # of registers with values from $[\![\varphi]\!]$.*

Intuitively, (1) says that if the transition reads a symbol stored in r satisfying φ, the symbol must also satisfy θ_r, the range of r. Because φ and θ_r are minterms, this only happens when $\varphi = \theta_r$. (2) says that the enabling condition $[\![\varphi]\!] \setminus \mathsf{img}(v) \neq \emptyset$, for all injective $v \models \theta$, holds if and only if there are fewer registers storing values from φ than the cardinality of φ. That implies we can always find a fresh element in $[\![\varphi]\!]$ to enable the transition. Registers holding values from φ are exactly those $r \in R$ such that $\theta_r = \varphi$. Both conditions can be effectively checked: the first one is a simple predicate-equivalence check, while the second one amounts to checking whether φ holds for at least a certain number k of distinct elements. This can be achieved by checking satisfiability of $\varphi \wedge \neg\mathsf{atom}(a_1) \wedge \cdots \wedge \neg\mathsf{atom}(a_{k-1})$, for $a_1, \ldots, a_{k-1}$ distinct elements of $[\![\varphi]\!]$.

Remark 2. Using single-valued SRAs to check enabledness might seem like a restriction. However, if one would start from a generic SRA, the process to check enabledness would contain an extra step: for each state p, we would have to keep track of all possible equations among registers. In fact, register equalities determine whether (i) register constraints of an outgoing transition are satisfiable; (ii) how many elements of the guard we need for the transition to happen, analogously to condition 2 of Lemma 1. Generating such equations is the key idea behind Theorem 1, and corresponds precisely to turning the SRA into a single-valued one.

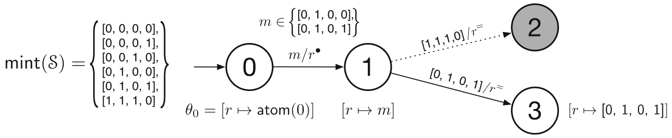
Given any SRA, we can use the notion of register abstraction to build an equivalent *normalized* SRA, where (i) states keep track of how the domains of registers change along transitions, (ii) transitions are obtained by breaking the one of the original SRA into minterms and discarding the ones that are disabled according to Lemma 1. In the following we write $\mathsf{mint}(\mathcal{S})$ for the minterms for the set of predicates $\{\varphi \mid p \xrightarrow{\varphi/\ell} q \in \Delta\} \cup \{\mathsf{atom}(v_0(r)) \mid v_0(r) \in \mathcal{D}, r \in R\}$. Observe that an atomic predicate always has an equivalent minterm, hence we will use atomic predicates to define the initial register abstraction.

Definition 4 (Normalized SRA). *Given an SRA $\mathcal{S}$, its normalization $\mathsf{N}(\mathcal{S})$ is the SRA $(R, \mathsf{N}(Q), \mathsf{N}(q_0), v_0, \mathsf{N}(F), \mathsf{N}(\Delta))$ where:*

- $\mathsf{N}(Q) = \{\theta \mid \theta$ *is a register abstraction over* $\mathsf{mint}(\mathcal{S}) \cup \{\bot\}\ \} \times Q$*; we will write $\theta \rhd q$ for $(\theta, q) \in \mathsf{N}(Q)$.*
- $\mathsf{N}(q_0) = \theta_0 \rhd q_0$*, where $(\theta_0)_r = \mathsf{atom}(v_0(r))$ if $v_0(r) \in \mathcal{D}$, and $(\theta_0)_r = \bot$ if $v_0(r) = \sharp$;*
- $\mathsf{N}(F) = \{\theta \rhd p \in \mathsf{N}(Q) \mid p \in F\}$
- $$\mathsf{N}(\Delta) = \{\theta \rhd p \xrightarrow{\theta_r/r^{=}} \theta \rhd q \mid p \xrightarrow{\varphi/r^{=}} q \in \Delta, \varphi \sqsubset \theta_r\} \cup$$ $$\{\theta \rhd p \xrightarrow{\psi/r^{\bullet}} \theta[r \mapsto \psi] \rhd q \mid p \xrightarrow{\varphi/r^{\bullet}} q \in \Delta, \varphi \sqsubset \psi, |[\![\psi]\!]| > \mathscr{E}(\theta, \psi)\}$$

The automaton $\mathsf{N}(\mathcal{S})$ enjoys the desired property: each transition from $\theta \rhd p$ is enabled by θ, by construction. $\mathsf{N}(\mathcal{S})$ is always *finite*. In fact, suppose $\mathcal{S}$ has n states, m transitions and r registers. Then $\mathsf{N}(\mathcal{S})$ has at most m predicates, and $|\mathsf{mint}(\mathcal{S})|$ is $\mathcal{O}(2^m)$. Since the possible register abstractions are $\mathcal{O}(r2^m)$, $\mathsf{N}(\mathcal{S})$ has $\mathcal{O}(nr2^m)$ states and $\mathcal{O}(mr^2 2^{3m})$ transitions.

Example 4. We now show the normalized version of Example 3. The first step is computing the set $\mathsf{mint}(\mathcal{S})$ of minterms for $\mathcal{S}$, i.e., the satisfiable Boolean combinations of $\{\mathsf{atom}(0), \mathsf{div}_3, [0,10] \wedge \mathsf{div}_5, < 0 \vee > 10\}$. For simplicity, we represent minterms as bitvectors where a 0 component means that the corresponding predicate is negated, e.g., $[1,1,1,0]$ stands for the minterm $\mathsf{atom}(0) \wedge ([0,10] \wedge \mathsf{div}_3) \wedge \mathsf{div}_5 \wedge \neg(< 0 \vee > 10)$. Minterms and the resulting SRA $\mathsf{N}(\mathcal{S})$ are shown below.


mint(S) = {[0, 0, 0, 0], [0, 0, 0, 1], [0, 0, 1, 0], [0, 1, 0, 0], [0, 1, 0, 1], [1, 1, 1, 0]}
m ∈ {[0, 1, 0, 0], [0, 1, 0, 1]}
m/r•
[1,1,1,0]/r=
[0, 1, 0, 1]/r=
0
1
2
3
θ0 = [r ↦ atom(0)]
[r ↦ m]
[r ↦ [0, 1, 0, 1]]

On each transition we show how it is broken down to minterms, and for each state we show the register abstraction (note that state 1 becomes two states in $\mathsf{N}(\mathcal{S})$). The transition from 1 to 2 is *not* part of $\mathsf{N}(\mathcal{S})$ – this is why it is dotted. In fact, in every register abstraction $[r \mapsto m]$ reachable at state 1, the component for the transition guard $[0,10] \wedge \mathsf{div}_5$ in the minterm m (3rd component) is 0, i.e., $([0,10] \wedge \mathsf{div}_5) \not\subseteq m$. Intuitively, this means that r will never be assigned a value that satisfies $[0,10] \wedge \mathsf{div}_5$. As a consequence, the construction of Definition 4 will not add a transition from 1 to 2.

Finally, we show that the normalized SRA behaves exactly as the original one.

Proposition 4. *$(p, v) \sim (\theta \rhd p, v)$, for all $p \in Q$ and $v \models \theta$. Hence, $\mathcal{S} \sim \mathsf{N}(\mathcal{S})$.*

Emptiness and Determinism. The transitions of $\mathsf{N}(\mathcal{S})$ are always enabled by construction, therefore every path in $\mathsf{N}(\mathcal{S})$ always corresponds to a run in $\mathsf{CLTS}(\mathsf{N}(\mathcal{S}))$.

Lemma 2. *The state $\theta \rhd p$ is reachable in $\mathsf{N}(\mathcal{S})$ if and only if there is a reachable configuration $(\theta \rhd p, v)$ in $\mathsf{CLTS}(\mathsf{N}(\mathcal{S}))$ such that $v \models \theta$. Moreover, if $(\theta \rhd p, v)$ is reachable, then all configurations $(\theta \rhd p, w)$ such that $w \models \theta$ are reachable.*

Therefore, using Proposition 4, we can reduce the reachability and emptiness problems of $\mathcal{S}$ to that of $\mathsf{N}(\mathcal{S})$.

Theorem 2 (Emptiness). *There is an algorithm to decide reachability of any configuration of $\mathcal{S}$, hence whether $\mathscr{L}(\mathcal{S}) = \emptyset$.*

Proof. Let (p, v) be a configuration of $\mathcal{S}$. To decide whether it is reachable in $\mathsf{CLTS}(\mathcal{S})$, we can perform a visit of $\mathsf{N}(\mathcal{S})$ from its initial state, stopping when a state $\theta \rhd p$ such that $v \models \theta$ is reached. If we are just looking for a final state, we can stop at any state such that $p \in F$. In fact, by Proposition 4, there is a run in $\mathsf{CLTS}(\mathcal{S})$ ending in (p, v) if and only if there is a run in $\mathsf{CLTS}(\mathsf{N}(\mathcal{S}))$ ending in $(\theta \rhd p, v)$ such that $v \models \theta$. By Lemma 2, the latter holds if and only if there is a path in $\mathsf{N}(\mathcal{S})$ ending in $\theta \rhd p$. This algorithm has the complexity of a standard visit of $\mathsf{N}(\mathcal{S})$, namely $\mathcal{O}(nr2^m + mr^2 2^{3m})$. □

Now that we characterized which transitions are reachable, we define what it means for a normalized SRA to be deterministic and we show that determinism is preserved by the translation from SRA.

Proposition 5 (Determinism). $\mathsf{N}(\mathcal{S})$ *is* deterministic *if and only if for all reachable transitions* $p \xrightarrow{\varphi_1/\ell_1} q_1, p \xrightarrow{\varphi_2/\ell_2} q_2 \in \mathsf{N}(\Delta)$ *the following holds:* $\varphi_1 \neq \varphi_2$ *whenever either (1)* $\ell_1 = \ell_2$ *and* $q_1 \neq q_2$*, or; (2)* $\ell_1 = r^\bullet$*,* $\ell_2 = s^\bullet$*, and* $r \neq s$*;*

One can check determinism of an SRA by looking at its normalized version.

Proposition 6. $\mathcal{S}$ *is deterministic if and only if* $\mathsf{N}(\mathcal{S})$ *is deterministic.*

Similarity and Bisimilarity. We now introduce a symbolic technique to decide similarity and bisimilarity of SRAs. The basic idea is similar to *symbolic (bi)simulation* [20,27] for RAs. Recall that RAs are SRAs whose transition guards are all $\top$. Given two RAs $\mathcal{S}_1$ and $\mathcal{S}_2$ a symbolic simulation between them is defined over their state spaces Q_1 and Q_2, not on their configurations. For this to work, one needs to add an extra piece of information about how registers of the two states are related. More precisely, a symbolic simulation is a relation on triples (p_1, p_2, σ), where $p_1 \in Q_1, p_2 \in Q_2$ and $\sigma \subseteq R_1 \times R_2$ is a *partial injective* function. This function encodes constraints between registers: $(r, s) \in \sigma$ is an equality constraint between $r \in R_1$ and $s \in R_2$, and $(r, s) \notin \sigma$ is an inequality constraint. Intuitively, (p_1, p_2, σ) says that all configurations (p_1, v_1) and (p_2, v_2) such that v_1 and v_2 satisfy σ – e.g., $v_1(r) = v_2(s)$ whenever $(r, s) \in \sigma$ – are in the simulation relation $(p_1, v_1) \prec (p_2, v_2)$. In the following we will use $v_1 \bowtie v_2$ to denote the function encoding constraints among v_1 and v_2, explicitly: $\sigma(r) = s$ if and only if $v_1(r) = v_2(s)$ and $v_1(r) \neq \sharp$.

Definition 5 (Symbolic (bi)similarity [27]). *A symbolic simulation is a relation* $\mathcal{R} \subseteq Q_1 \times Q_1 \times \mathcal{P}(R_1 \times R_2)$ *such that if* $(p_1, p_2, \sigma) \in \mathcal{R}$*, then* $p_1 \in F_1$ *implies* $p_2 \in F_2$*, and if* $p_1 \xrightarrow{\ell} q_1 \in \Delta_1$[1] *then:*

1. *if* $\ell = r^=$*:*
   - *(a) if* $r \in \mathsf{dom}(\sigma)$*, then there is* $p_2 \xrightarrow{\sigma(r)^=} q_2 \in \Delta_2$ *such that* $(q_1, q_2, \sigma) \in \mathcal{R}$*.*
   - *(b) if* $r \notin \mathsf{dom}(\sigma)$ *then there is* $p_2 \xrightarrow{s^\bullet} q_2 \in \Delta_2$ *s.t.* $(q_1, q_2, \sigma[r \mapsto s]) \in \mathcal{R}$*.*

[1] We will keep the $\top$ guard implicit for succinctness.

2 if $\ell = r^\bullet$:

(a) for all $s \in R_2 \setminus \mathsf{img}(\sigma)$, there is $p_2 \xrightarrow{s^=} q_2 \in \Delta_2$ such that $(q_1, q_2, \sigma[r \mapsto s]) \in \mathcal{R}$, and;

(b) there is $p_2 \xrightarrow{s^\bullet} q_2 \in \Delta_2$ such that $(q_1, q_2, \sigma[r \mapsto s]) \in \mathcal{R}$.

Here $\sigma[r \mapsto s]$ stands for $\sigma \setminus (\sigma^{-1}(s), s) \cup (r, s)$, which ensures that σ stays injective when updated.

Given a symbolic simulation $\mathcal{R}$, its inverse is defined as $\mathcal{R}^{-1} = \{t^{-1} \mid t \in \mathcal{R}\}$, where $(p_1, p_2, \sigma)^{-1} = (p_2, p_1, \sigma^{-1})$. A symbolic bisimulation *$\mathcal{R}$ is a relation such that both $\mathcal{R}$ and $\mathcal{R}^{-1}$ are symbolic simulations.*

Case 1 deals with cases when p_1 can perform a transition that reads the register r. If $r \in \mathsf{dom}(\sigma)$, meaning that r and $\sigma(r) \in R_2$ contain the same value, then p_2 must be able to read $\sigma(r)$ as well. If $r \notin \mathsf{dom}(\sigma)$, then the content of r is fresh w.r.t. p_2, so p_2 must be able to read any fresh value—in particular the content of r. Case 2 deals with the cases when p_1 reads a fresh value. It ensures that p_2 is able to read all possible values that are fresh for p_1, be them already in some register s – i.e., $s \in R_2 \setminus \mathsf{img}(\sigma)$, case 2(a) – or fresh for p_2 as well – case 2(b). In all these cases, σ must be updated to reflect the new equalities among registers.

Keeping track of equalities among registers is enough for RAs, because the actual content of registers does not determine the capability of a transition to fire (RA transitions have implicit $\top$ guards). As seen in Example 3, this is no longer the case for SRAs: a transition may or may not happen depending on the register assignment being compatible with the transition guard.

As in the case of reachability, normalized SRAs provide the solution to this problem. We will reduce the problem of checking (bi)similarity of $\mathcal{S}_1$ and $\mathcal{S}_2$ to that of checking symbolic (bi)similarity on $\mathsf{N}(\mathcal{S}_1)$ and $\mathsf{N}(\mathcal{S}_2)$, with minor modifications to the definition. To do this, we need to assume that minterms for both $\mathsf{N}(\mathcal{S}_1)$ and $\mathsf{N}(\mathcal{S}_2)$ are computed over the union of predicates of $\mathcal{S}_1$ and $\mathcal{S}_2$.

Definition 6 (N-simulation). *A N-simulation on $\mathcal{S}_1$ and $\mathcal{S}_2$ is a relation $\mathcal{R} \subseteq \mathsf{N}(Q_1) \times \mathsf{N}(Q_2) \times \mathcal{P}(R_1 \times R_2)$, defined as in Definition 5, with the following modifications:*

(i) we require that $\theta_1 \rhd p_1 \xrightarrow{\varphi_1/\ell_1} \theta_1' \rhd q_1 \in \mathsf{N}(\Delta_1)$ must be matched by transitions $\theta_2 \rhd p_2 \xrightarrow{\varphi_2/\ell_2} \theta_2' \rhd q_2 \in \mathsf{N}(\Delta_2)$ such that $\varphi_2 = \varphi_1$.

(ii) we modify case 2 as follows (changes are underlined):

2(a)' for all $s \in R_2 \setminus \mathsf{img}(\sigma)$ $\underline{\text{such that } \varphi_1 = (\theta_2)_s,}$ there is $\theta_2 \rhd p_2 \xrightarrow{\varphi_1/s^=} \theta_2' \rhd q_2 \in \mathsf{N}(\Delta_2)$ such that $(\theta_1' \rhd q_1, \theta_2' \rhd q_2, \sigma[r \mapsto s]) \in \mathcal{R}$, and;

2(b)' $\underline{\text{if } \mathscr{E}(\theta_1, \varphi_1) + \mathscr{E}(\theta_2, \varphi_1) < |[\![\varphi_1]\!]|, \text{ then}}$ there is $\theta_2 \rhd p_2 \xrightarrow{\varphi_1/s^\bullet} \theta_2' \rhd q_2 \in \mathsf{N}(\Delta_2)$ such that $(\theta_1' \rhd q_1, \theta_2' \rhd q_2, \sigma[r \mapsto s]) \in \mathcal{R}$.

A N-bisimulation $\mathcal{R}$ is a relation such that both $\mathcal{R}$ and $\mathcal{R}^{-1}$ are N-simulations. We write $\mathcal{S}_1 \overset{\mathtt{N}}{\prec} \mathcal{S}_2$ (resp. $\mathcal{S}_1 \overset{\mathtt{N}}{\sim} \mathcal{S}_2$) if there is a N-simulation (resp. bisimulation) $\mathcal{R}$ such that $(\mathsf{N}(q_{01}), \mathsf{N}(q_{02}), v_{01} \bowtie v_{02}) \in \mathcal{R}$.

The intuition behind this definition is as follows. Recall that, in a normalized SRA, transitions are defined over minterms, which cannot be further broken down, and are mutually disjoint. Therefore two transitions can read the same values if and only if they have the same minterm guard. Thus condition (i) makes sure that matching transitions can read exactly the same set of values. Analogously, condition (ii) restricts how a fresh transition of $\mathsf{N}(\mathcal{S}_1)$ must be matched by one of $\mathsf{N}(\mathcal{S}_2)$: 2(a)' only considers transitions of $\mathsf{N}(\mathcal{S}_2)$ reading registers $s \in R_2$ such that $\varphi_1 = (\theta_2)_s$ because, by definition of normalized SRA, $\theta_2 \rhd p_2$ has no such transition if this condition is not met. Condition 2(b)' amounts to requiring a fresh transition of $\mathsf{N}(\mathcal{S}_2)$ that is enabled by both θ_1 and θ_2 (see Lemma 1), i.e., that can read a symbol that is fresh w.r.t. both $\mathsf{N}(\mathcal{S}_1)$ and $\mathsf{N}(\mathcal{S}_2)$.

N-simulation is sound and complete for standard simulation.

Theorem 3. *$\mathcal{S}_1 \prec \mathcal{S}_2$ if and only if $\mathcal{S}_1 \overset{\mathtt{N}}{\prec} \mathcal{S}_2$.*

As a consequence, we can decide similarity of SRAs via their normalized versions. N-simulation is a relation over a finite set, namely $\mathsf{N}(Q_1) \times \mathsf{N}(Q_2) \times \mathcal{P}(R_1 \times R_2)$, therefore N-similarity can always be decided in finite time. We can leverage this result to provide algorithms for checking language inclusion/equivalence for deterministic SRAs (recall that they are undecidable for non-deterministic ones).

Theorem 4. *Given two deterministic SRAs $\mathcal{S}_1$ and $\mathcal{S}_2$, there are algorithms to decide $\mathscr{L}(\mathcal{S}_1) \subseteq \mathscr{L}(\mathcal{S}_2)$ and $\mathscr{L}(\mathcal{S}_1) = \mathscr{L}(\mathcal{S}_2)$.*

Proof. By Proposition 1 and Theorem 3, we can decide $\mathscr{L}(\mathcal{S}_1) \subseteq \mathscr{L}(\mathcal{S}_2)$ by checking $\mathcal{S}_1 \overset{\mathtt{N}}{\prec} \mathcal{S}_2$. This can be done algorithmically by iteratively building a relation $\mathcal{R}$ on triples that is an N-simulation on $\mathsf{N}(\mathcal{S}_1)$ and $\mathsf{N}(\mathcal{S}_2)$. The algorithm initializes $\mathcal{R}$ with $(\mathsf{N}(q_{01}), \mathsf{N}(q_{02}), v_{01} \bowtie v_{02})$, as this is required to be in $\mathcal{R}$ by Definition 6. Each iteration considers a candidate triple t and checks the conditions for N-simulation. If satisfied, it adds t to $\mathcal{R}$ and computes the next set of candidate triples, i.e., those which are required to belong to the simulation relation, and adds them to the list of triples still to be processed. If not, the algorithm returns $\mathscr{L}(\mathcal{S}_1) \not\subseteq \mathscr{L}(\mathcal{S}_2)$. The algorithm terminates returning $\mathscr{L}(\mathcal{S}_1) \subseteq \mathscr{L}(\mathcal{S}_2)$ when no triples are left to process. Determinism of $\mathcal{S}_1$ and $\mathcal{S}_2$, and hence of $\mathsf{N}(\mathcal{S}_1)$ and $\mathsf{N}(\mathcal{S}_2)$ (by Proposition 6), ensures that computing candidate triples is deterministic. To decide $\mathscr{L}(\mathcal{S}_1) = \mathscr{L}(\mathcal{S}_2)$, at each iteration we need to check that both t and t^{-1} satisfy the conditions for N-simulation.

If $\mathcal{S}_1$ and $\mathcal{S}_2$ have, respectively, n_1, n_2 states, m_1, m_2 transitions, and r_1, r_2 registers, the normalized versions have $\mathcal{O}(n_1 r_1 2^{m_1})$ and $\mathcal{O}(n_2 r_2 2^{m_2})$ states. Each triple, taken from the finite set $\mathsf{N}(Q_1) \times \mathsf{N}(Q_2) \times \mathcal{P}(R_1 \times R_2)$, is processed exactly once, so the algorithm iterates $\mathcal{O}(n_1 n_2 r_1 r_2 2^{m_1+m_2+r_1 r_2})$ times. □

5 Evaluation

We have implemented SRAs in the open-source Java library SVPALib [26]. In our implementation, constructions are computed lazily when possible (e.g., the normalized SRA for emptiness and (bi)similarity checks). All experiments were performed on a machine with 3.5 GHz Intel Core i7 CPU with 16 GB of RAM (JVM 8 GB), with a timeout value of 300 s. The goal of our evaluation is to answer the following research questions:

Q1: Are SRAs more succinct than existing models when processing strings over large but finite alphabets? (Sect. 5.1)
Q2: What is the performance of membership for deterministic SRAs and how does it compare to the matching algorithm in `java.util.regex`? (Sect. 5.2)
Q3: Are SRA decision procedures practical? (Sect. 5.3)

Benchmarks. We focus on regular expressions with back-references, therefore all our benchmarks operate over the Boolean algebra of Unicode characters with interval—i.e., the set of characters is the set of all 2^{16} UTF-16 characters and the predicates are union of intervals (e.g., `[a-zA-Z]`).[2] Our benchmark set contains 19 SRAs that represent variants of regular expressions with back-references obtained from the regular-expression crowd-sourcing website RegExLib [23]. The expressions check whether inputs have, for example, matching first/last name initials or both (Name-F, Name-L and Name), correct Product Codes/Lot number of total length n (Pr-Cn, Pr-CLn), matching XML tags (XML), and IP addresses that match for n positions (IPn). We also create variants of the product benchmark presented in Sect. 2 where we vary the numbers of characters in the code and lot number. All the SRAs are deterministic.

5.1 Succinctness of SRAs vs SFAs

In this experiment, we relate the size of SRAs over finite alphabets to the size of the smallest equivalent SFAs. For each SRA, we construct the equivalent SFA by equipping the state space with the values stored in the registers at each step (this construction effectively builds the configuration LTS). Figure 2a shows the results. As expected, SFAs tend to blow up in size when the SRA contains multiple registers or complex register values. In cases where the register values range over small sets (e.g., `[0-9]`) it is often feasible to build an SFA equivalent to the SRA, but the construction always yields very large automata. In cases where the registers can assume many values (e.g., 2^{16}) SFAs become prohibitively large and do not fit in memory. To answer **Q1**, even for finite alphabets, **it is not feasible to compile SRAs to SFAs**. Hence, SRAs are a succinct model.

5.2 Performance of Membership Checking

In this experiment, we measure the performance of SRA membership, and we compare it with the performance of the `java.util.regex` matching algorithm.

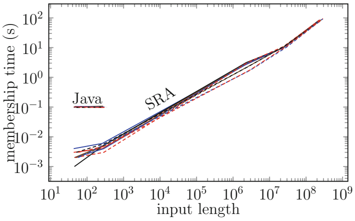
[2] Our experiments are over finite alphabets, but the Boolean algebra can be infinite by taking the alphabet to be positive integers and allowing intervals to contain ∞ as upper bound. This modification does not affect the running time of our procedures, therefore we do not report it.

	SRA				SFA	
	states	tr	reg	|reg|	states	tr
IP2	44	46	3	10	4,013	4,312
IP3	44	46	4	10	39,113	42,112
IP4	44	46	5	10	372,113	402,112
IP6	44	46	7	10	—	—
IP9	44	46	10	10	—	—
Name-F	7	10	2	26	201	300
Name-L	7	10	2	26	129	180
Name	7	10	3	26	3,201	4,500
XML	12	16	4	52	—	—
Pr-C2	26	28	3	2^{16}	—	—
Pr-C3	28	30	4	2^{16}	—	—
Pr-C4	30	32	5	2^{16}	—	—
Pr-C6	34	36	7	2^{16}	—	—
Pr-C9	40	42	10	2^{16}	—	—
Pr-CL2	26	28	3	2^{16}	—	—
Pr-CL3	28	30	4	2^{16}	—	—
Pr-CL4	30	32	5	2^{16}	—	—
Pr-CL6	34	36	7	2^{16}	—	—
Pr-CL9	40	42	10	2^{16}	—	—

(a) Size of SRAs vs SFAs. (—) denotes the SFA didn't fit in memory. |reg| denotes how many different characters a register stored.

SRA $\mathcal{S}_1$	SRA $\mathcal{S}_2$	$\mathscr{L}_1=\emptyset$	$\mathscr{L}_1=\mathscr{L}_1$	$\mathscr{L}_2 \subseteq \mathscr{L}_1$
Pr-C2	Pr-CL2	0.125s	0.905s	3.426s
Pr-C3	Pr-CL3	1.294s	5.558s	24.688s
Pr-C4	Pr-CL4	13.577s	55.595s	—
Pr-C6	Pr-CL6	—	—	—
Pr-CL2	Pr-C2	1.067s	0.952s	0.889s
Pr-CL3	Pr-C3	10.998s	11.104s	11.811s
Pr-CL4	Pr-C4	—	—	—
Pr-CL6	Pr-C6	—	—	—
IP-2	IP-3	0.125s	0.408s	1.845s
IP-3	IP-4	1.288s	2.953s	21.627s
IP-4	IP-6	18.440s	42.727s	—
IP-6	IP-9	—	—	—

(b) Performance of decision procedures. In the table $\mathscr{L}_i = \mathscr{L}(\mathcal{S}_i)$, for $i = 1, 2$.



(c) SRA membership and Java `regex` matching performance. Missing data points for Java are stack overflows.

Fig. 2. Experimental results.

For each benchmark, we generate inputs of length varying between approximately 100 and 10^8 characters and measure the time taken to check membership. Figure 2c shows the results. The performance of SRA (resp. Java) is not particularly affected by the size of the expression. Hence, the lines for different expressions mostly overlap. As expected, for SRAs the time taken to check membership grows linearly in the size of the input (axes are log scale). Remarkably, even though our implementation does not employ particular input processing optimizations, it can still check membership for strings with tens of millions of characters in less than 10 s. We have found that our implementation is more efficient than the Java `regex` library, matching the same input an average of 50 times faster than `java.util.regex.Matcher`. `java.util.regex.Matcher` seems to make use of a recursive algorithm to match back-references, which means it does not scale well. Even when given the maximum stack size, the JVM will return a Stack Overflow for inputs as small as 20,000 characters. Our implementation can match such strings in less than 2 s. To answer **Q2**, **deterministic SRAs can be efficiently executed on large inputs and perform better than the `java.util.regex` matching algorithm**.

5.3 Performance of Decision Procedures

In this experiment, we measure the performance of SRAs simulation and bisimulation algorithms. Since all our SRAs are deterministic, these two checks correspond to language equivalence and inclusion. We select pairs of benchmarks for which the above tests are meaningful (e.g., variants of the problem discussed at the end of Sect. 2). The results are shown in Fig. 2b. As expected, due to the translation to single-valued SRAs, our decision procedures do not scale well in the number of registers. This is already the case for classic register automata and it is not a surprising result. However, our technique can still check equivalence and inclusion for regular expressions that no existing tool can handle. To answer **Q3**, **bisimulation and simulation algorithms for SRAs only scale to small numbers of registers**.

6 Conclusions

In this paper we have presented *Symbolic Register Automata*, a novel class of automata that can handle complex alphabet theories while allowing symbol comparisons for equality. SRAs encompass – and are strictly more powerful – than both Register and Symbolic Automata. We have shown that they enjoy the same closure and decidability properties of the former, despite the presence of arbitrary guards on transitions, which are not allowed by RAs. Via a comprehensive set of experiments, we have concluded that SRAs are vastly more succinct than SFAs and membership is efficient on large inputs. Decision procedures do not scale well in the number of registers, which is already the case for basic RAs.

Related Work. RAs were first introduced in [17]. There is an extensive literature on register automata, their formal languages and decidability properties [7,13,21,22,25], including variants with *global freshness* [20,27] and totally ordered data [4,14]. SRAs are based on the original model of [17], but are much more expressive, due to the presence of guards from an arbitrary decidable theory.

In recent work, variants over richer theories have appeared. In [9] RA over rationals were introduced. They allow for a restricted form of linear arithmetic among registers (RAs with arbitrary linear arithmetic subsume two-counter automata, hence are undecidable). SRAs do not allow for operations on registers, but encompass a wider range of theories without any loss in decidability. Moreover, [9] does not study Boolean closure properties. In [8,16], RAs allowing guards over a range of theories – including (in)equality, total orders and increments/sums – are studied. Their focus is different than ours as they are interested primarily in *active learning* techniques, and several restrictions are placed on models for the purpose of the learning process. We can also relate SRAs with *Quantified Event Automata* [2], which allow for guards and assignments to registers on transitions. However, in QEA guards can be arbitrary, which could lead to several problems, e.g. undecidable equivalence.

Symbolic automata were first introduced in [28] and many variants of them have been proposed [12]. The one that is closer to SRAs is Symbolic Extended Finite Automata (SEFA) [11]. SEFAs are SFAs in which transitions can read more than one character at a time. A transition of arity k reads k symbols which are consumed if they satisfy the predicate $\varphi(x_1, \ldots, x_k)$. SEFAs allow arbitrary k-ary predicates over the input theory, which results in most problems being undecidable (e.g., equivalence and intersection emptiness) and in the model not being closed under Boolean operations. Even when deterministic, SEFAs are not closed under union and intersection. In terms of expressiveness, SRAs and SEFAs are incomparable. SRAs can only use equality, but can compare symbols at arbitrary points in the input while SEFAs can only compare symbols within a constant window, but using arbitrary predicates.

Several works study matching techniques for extended regular expressions [3,5,18,24]. These works introduce automata models with ad-hoc features for extended regular constructs – including back-references – but focus on efficient matching, without studying closure and decidability properties. It is also worth noting that SRAs are not limited to alphanumeric or finite alphabets. On the negative side, SRAs cannot express capturing groups of an unbounded length, due to the finitely many registers. This limitation is essential for decidability.

Future Work. In [21] a polynomial algorithm for checking language equivalence of deterministic RAs is presented. This crucially relies on closure properties of symbolic bisimilarity, some of which are lost for SRAs. We plan to investigate whether this algorithm can be adapted to our setting. Extending SRAs with more complex comparison operators other than equality (e.g., a total order $<$) is an interesting research question, but most extensions of the model quickly lead to undecidability. We also plan to study active automata learning for SRAs, building on techniques for SFAs [1], RAs [6,8,16] and nominal automata [19].

References

1. Argyros, G., D'Antoni, L.: The learnability of symbolic automata. In: CAV, pp. 427–445 (2018)
2. Barringer, H., Falcone, Y., Havelund, K., Reger, G., Rydeheard, D.E.: Quantified event automata: towards expressive and efficient runtime monitors. In: FM, pp. 68–84 (2012)
3. Becchi, M., Crowley, P.: Extending finite automata to efficiently match perl-compatible regular expressions. In: CoNEXT, pp. 25 (2008)
4. Benedikt, M., Ley, C., Puppis, G.: What you must remember when processing data words. In: AMW (2010)
5. Bispo, J., Sourdis, I., Cardoso, J.M.P., Vassiliadis, S.: Regular expression matching for reconfigurable packet inspection. In: FPT, pp. 119–126 (2006)
6. Bollig, B., Habermehl, P., Leucker, M., Monmege, B.: A fresh approach to learning register automata. In: DLT, pp. 118–130 (2013)
7. Cassel, S., Howar, F., Jonsson, B., Merten, M., Steffen, B.: A succinct canonical register automaton model. J. Log. Algebr. Meth. Program. **84**(1), 54–66 (2015)

8. Cassel, S., Howar, F., Jonsson, B., Steffen, B.: Active learning for extended finite state machines. Formal Asp. Comput. **28**(2), 233–263 (2016)
9. Chen, Y., Lengál, O., Tan, T., Wu, Z.: Register automata with linear arithmetic. In: LICS, pp. 1–12 (2017)
10. D'Antoni, L., Ferreira, T., Sammartino, M., Silva, A.: Symbolic register automata. CoRR, abs/1811.06968 (2019). http://arxiv.org/abs/1811.06968
11. D'Antoni, L., Veanes, M.: Extended symbolic finite automata and transducers. Formal Meth. Syst. Des. **47**(1), 93–119 (2015)
12. D'Antoni, L., Veanes, M.: The power of symbolic automata and transducers. In: CAV, pp. 47–67 (2017)
13. Demri, S., Lazic, R.: LTL with the freeze quantifier and register automata. ACM Trans. Comput. Log. **10**(3), 16:1–16:30 (2009)
14. Figueira, D., Hofman, P., Lasota, S.: Relating timed and register automata. Math. Struct. Comput. Sci. **26**(6), 993–1021 (2016)
15. Grigore, R., Distefano, D., Petersen, R.L., Tzevelekos, N.: Runtime verification based on register automata. In: TACAS, pp. 260–276 (2013)
16. Isberner, M., Howar, F., Steffen, B.: Learning register automata: from languages to program structures. Mach. Learn. **96**(1–2), 65–98 (2014)
17. Kaminski, M., Francez, N.: Finite-memory automata. Theor. Comput. Sci. **134**(2), 329–363 (1994)
18. Komendantsky, V.: Matching problem for regular expressions with variables. In: Loidl, H.-W., Peña, R. (eds.) TFP 2012. LNCS, vol. 7829, pp. 149–166. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40447-4_10
19. Moerman, J., Sammartino, M., Silva, A., Klin, B., Szynwelski, M.: Learning nominal automata. In: POPL, pp. 613–625 (2017)
20. Murawski, A.S., Ramsay, S.J., Tzevelekos, N.: Bisimilarity in fresh-register automata. In: LICS, pp. 156–167 (2015)
21. Murawski, A.S., Ramsay, S.J., Tzevelekos, N.: Polynomial-time equivalence testing for deterministic fresh-register automata. In: MFCS, pp. 72:1–72:14 (2018)
22. Neven, F., Schwentick, T., Vianu, V.: Finite state machines for strings over infinite alphabets. ACM Trans. Comput. Log. **5**(3), 403–435 (2004)
23. RegExLib. Regular expression library (2017). http://regexlib.com/
24. Reidenbach, D., Schmid, M.L.: A polynomial time match test for large classes of extended regular expressions. In: Domaratzki, M., Salomaa, K. (eds.) CIAA 2010. LNCS, vol. 6482, pp. 241–250. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-18098-9_26
25. Sakamoto, H., Ikeda, D.: Intractability of decision problems for finite-memory automata. Theor. Comput. Sci. **231**(2), 297–308 (2000)
26. SVPAlib: Symbolic automata library (2018). https://github.com/lorisdanto/symbolicautomata
27. Tzevelekos, N.: Fresh-register automata. In: POPL, pp. 295–306 (2011)
28. Veanes, M., Halleux, P.D., Tillmann, N.: Rex: symbolic regular expression explorer. In: ICST, pp. 498–507 (2010)