Succinct Determinisation of Counting Automata via Sphere Construction [⋆]

Lukáš Holík[1], Ondřej Lengál[1], Olli Saarikivi[2],
Lenka Turoňová[1], Margus Veanes[2], Tomáš Vojnar[1]

[1] FIT, Brno University of Technology, IT4Innovations Centre of Excellence, Czech Republic
[2] Microsoft Research, Redmond, USA

Abstract. We propose an efficient algorithm for determinising counting automata (CAs), i.e., finite automata extended with bounded counters. The algorithm avoids unfolding counters into control states, unlike the naïve approach, and thus produces much smaller deterministic automata. We also develop a simplified and faster version of the general algorithm for the sub-class of so-called monadic CAs (MCAs), i.e., CAs with counting loops on character classes, which are common in practice. Our main motivation is (besides applications in verification and decision procedures of logics) the application of deterministic (M)CAs in pattern matching regular expressions with counting, which are very common in e.g. network traffic processing and log analysis. We have evaluated our algorithm against practical benchmarks from these application domains and concluded that compared to the naïve approach, our algorithm is much less prone to explode, produces automata that can be several orders of magnitude smaller, and is overall faster.

1 Introduction

The *counting operator*—also known as the operator of *limited repetition*—is an operator commonly used in extended regular expressions (also called *regexes*). Limited repetitions do not extend expressiveness beyond regularity, but allow one to succinctly express patterns such as `(ab){1,100}` representing all words where `ab` appears 1–100 times. Such expressions are very common (cf. [3]), e.g., in the RegExLib library [20], which collects expressions for recognising URIs, markup code, pieces of Java code, or SQL queries; in the Snort rules [17] used for finding attacks in network traffic; or in real-life XML schemas, with the counter bounds being as large as 10 million [3]. This observation is confirmed by our own experiments with patterns provided by Microsoft for verifying absence of information leakage from network traffic logs. Counting constraints may also naturally arise in other contexts, such as in automata-based verification approaches (e.g. [11]) for describing sets of runs through a loop with some number of repetitions.

Several finite automata counterparts of regular counting constraints have appeared in the literature (e.g. [13, 15, 24, 25]), all essentially boiling down to variations on counter automata with counters limited to a bounded range of values. Such counters do not extend the expressive power beyond regularity, but bring succinctness, exactly

[⋆] This work has been supported by the Czech Science Foundation (project No. 19-24397S), the IT4Innovations Excellence in Science (project No. LQ1602), and the FIT BUT internal project FIT-S-17-4014.

as the counters in extended regular expressions. In this paper, we call these automata *counting automata* (CAs).

The main contribution of this paper is a novel *succinct determinisation* of CAs. Our main motivation is in *pattern matching*, where deterministic automata allow for algorithms running reliably in time linear to the length of the text. However, the naïve determinisation of CAs (and counting constraints in general)—which encodes counter values as parts of control states, leading to classical nondeterministic finite automata (NFAs), which are then determinised using the standard subset construction—can easily lead to state explosion, causing the approach to fail. See, e.g., the CA in Fig. 1, for which the minimal deterministic finite automaton (DFA) has 2^{k+1} states with k being the upper bound of the counter. *Backtracking*-based algorithms, which can be used instead, are slower and unpredictable, may easily require a prohibitively large time, and are even prone to DoS attacks, cf. [19]. A viable alternative is *on-the-fly determinisation*, which determinises only the part of the given NFA through which the input word passes, as proposed already in [27]. However, the overhead during matching might be significant, and the construction can still explode on some words, much like the full determinisation, especially when large bounds on counters are used (which, in our experience, makes some regex matchers to give up already the translation to NFAs).

Our algorithm, which allows one to *succinctly* determinise CAs, is therefore a major step towards alleviating the above problems by making the determinisation-based algorithms applicable more widely. We note that this has been an open problem (whose importance was stressed, e.g., in [25]) that a number of other works, such as [13, 15], have attempted to solve, but they could only cope with very restricted fragments or alleviate the problem only partially, yielding solutions of limited practical applicability only.

Our algorithm is general and often produces small results. Moreover, we also propose a version specialised to counting restricted to repetition of single characters from some *character class*, called *monadic counting* here (e.g., `[ab]{10}` is monadic while `(ab){10}` is not). This class is of particular practical relevance since we discovered that most of the regular expressions with counters used in practice are of this form. Our specialised algorithm can produce deterministic CAs exponentially more succinct than the corresponding DFAs and its worst-case complexity is only polynomial in the maximum values of counters (in contrast to the exponential naïve construction).

We have implemented the monadic CA determinisation and evaluated it on real-life datasets of regular expressions with monadic counting. We found that our resulting CAs can be much smaller than minimal DFAs, are less prone to explode, and that our algorithm, though not optimised, is overall faster than the naïve determinisation that unfolds counters. We also confirmed that monadic regexes present an important subproblem, with over 95 % of regexes in the explored datasets being of this type.

Running example. To illustrate our algorithms, consider the regex `.*a.{`k`}` where $k \in \mathbb{N}$. It says that the $(k+1)$-th letter from the end of the word must be `a`. The minimal DFA accepting the language has 2^{k+1} states since it must remember in its control states the positions of all letters `a` that were seen during the last $k+1$ steps. For this,

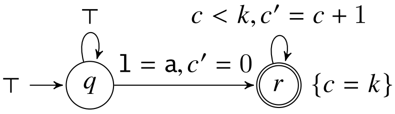

Fig. 1. A CA for the regex `.*a.{`k`}` with $k \in \mathbb{N}$, $I : \mathtt{s} = q$, $F : \mathtt{s} = r \wedge c = k$, and $\Delta : q \xrightarrow{\{\top,\top\}} q \vee q \xrightarrow{\{\mathtt{1}=\mathtt{a}, c'=0\}} r \vee r \xrightarrow{\{c<k, c'=c+1\}} r$.

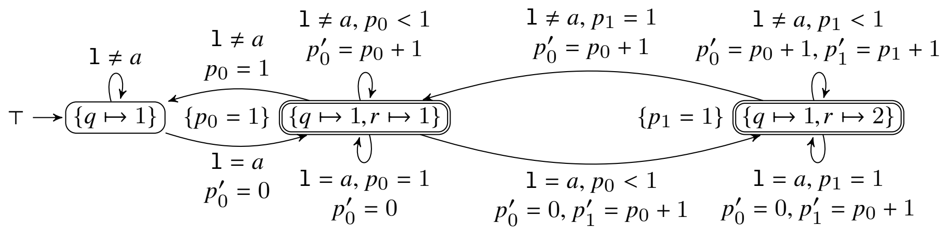
$\top$
$\mathtt{1} \neq a$
$\{q \mapsto 1\}$
$\mathtt{1} \neq a$
$p_0 = 1$
$\{p_0 = 1\}$
$\mathtt{1} = a$
$p'_0 = 0$
$\mathtt{1} \neq a, p_0 < 1$
$p'_0 = p_0 + 1$
$\{q \mapsto 1, r \mapsto 1\}$
$\mathtt{1} = a, p_0 = 1$
$p'_0 = 0$
$\mathtt{1} \neq a, p_1 = 1$
$p'_0 = p_0 + 1$
$\mathtt{1} = a, p_0 < 1$
$p'_0 = 0, p'_1 = p_0 + 1$
$\{p_1 = 1\}$
$\{q \mapsto 1, r \mapsto 2\}$
$\mathtt{1} \neq a, p_1 < 1$
$p'_0 = p_0 + 1, p'_1 = p_1 + 1$
$\mathtt{1} = a, p_1 = 1$
$p'_0 = 0, p'_1 = p_0 + 1$

Fig. 2. The DCA generated from the CA of Fig. 1 for $k = 1$ by our algorithm for determinisation of monadic CA (Section 4.2).

it needs a finite memory of $k + 1$ bits, which has 2^{k+1} reachable configurations. The regex corresponds to the nondeterministic CA of Fig. 1. In the transition labels, the predicates over the variable $\mathtt{1}$ constrain the input symbol, the predicates over c constrain the current value of the counter c, and the primed variant of c, i.e., c', stands for the value of c after taking the transition. The initial value of c is unrestricted, and the automaton accepts in the state r if the value of c equals k. Our monadic determinisation algorithm, presented in Section 4.2, then outputs the deterministic CA (DCA) of Fig. 2 (for $k = 1$). Intuitively, it uses $k + 1$ counters to remember how far back the last $k + 1$ occurrences of a appeared. Depending on k, the resulting DCA has $k + 2$ states, $4(k + 1) + 1$ transitions, and $k + 1$ counters. That is, its size is linear to k in contrast to the factor 2^k in the size of the minimal DFA. □

2 Counting Automata

Preliminaries. We use $\mathbb{N}$ to denote the set of natural numbers $\{0, 1, 2, \ldots\}$. Given a function $f : A \to B$, we refer to the elements of f using $a \mapsto b$ (when $f(a) = b$). For the rest of the paper, we consider a fixed finite *alphabet* Σ of *symbols*. A word over Σ is a finite sequence of symbols $w = a_1 \cdots a_n \in \Sigma^*$. We use ϵ to denote the *empty word*.

Given a set of variables V and a set of constants Q (disjoint with $\mathbb{N}$), we define a *Q-formula over V* to be a quantifier-free formula φ of Presburger arithmetic extended with constants from Q and Σ, i.e., a Boolean combination of (in-)equalities $t_1 = t_2$ or $t_1 \leq t_2$ where t_1 and t_2 are constructed using $+$, $\mathbb{N}$, and V, and predicates of the form $x = a$ or $x = q$ for $x \in V$, $a \in \Sigma$, and $q \in Q$. An assignment M to free variables of φ is a *model* of φ, denoted as $M \models \varphi$, if it makes φ true. We use $\mathtt{sat}(\varphi)$ to denote that φ has a model.

Given a formula φ and a (partial) map $\theta : \mathit{terms}(\varphi) \to S$, where $\mathit{terms}(\varphi)$ denotes the set of terms in φ and S is some set of terms, $\varphi[\theta]$ denotes a *term substitution*, i.e., the formula φ with all occurrences of every term $t \in \mathit{dom}(\theta)$ replaced by $\theta(t)$. As usual, replacing a larger term takes priority over replacing its subterms (we treat primed variables and parameters as atomic terms, hence $(p' = 1)[\{p \mapsto q\}]$ is still $p' = 1$). The *substitution formula* φ_θ of θ is defined as the conjunction of equalities $\varphi_\theta \overset{\text{def}}{=} \bigwedge_{t \in \mathit{dom}(\theta)} (\theta(t) = t)$. Finally, the set of *minterms* of a finite set Φ of predicates is defined as the set of all satisfiable predicates of $\{\bigwedge_{\phi \in \Phi'} \phi \wedge \bigwedge_{\phi \in \Phi \setminus \Phi'} \neg\phi \mid \Phi' \subseteq \Phi\}$.

Labelled transition systems. We will introduce our counting automata, such as that of Fig.1, as a specialisation of the more general model of labelled transition systems. This perspective and related notation allows for a more abstract and concise formulation of our algorithms than the more standard approach, in which one would define counting automata in a more straightforward manner as an extension of the classical finite automata.

A *labelled transition system* (LTS) over Σ is a tuple $T = (Q, V, I, F, \Delta)$ where Q is a finite set of *control states*, V is a finite set of *configuration variables*, I is the *initial Q-formula* over V, F is the *final Q-formula* over V, and Δ is the *transition Q-formula* over $V \cup V' \cup \{\mathtt{l}\}$ with $V' = \{x' \mid x \in V\}$, $V \cap V' = \emptyset$, and $\mathtt{l} \notin V$. We call $\mathtt{l}$ the *letter/symbol variable* and allow it as the only term that can occur within a predicate $\mathtt{l} = a$ for $a \in \Sigma$, called an *atomic symbol guard*.[3] Moreover, $\mathtt{l}$ is also not allowed to occur in any other predicates in Δ. A *configuration* is an assignment $\alpha : V \to \mathbb{N} \cup Q$ that maps every configuration variable to a number from $\mathbb{N}$ or a state from Q. Let C be the set of all configurations. The transition formula Δ encodes the transition relation $[\![\Delta]\!] \subseteq C \times \Sigma \times C$ such that $(\alpha, a, \alpha') \in [\![\Delta]\!]$ iff $\alpha \cup \{x' \mapsto k \mid \alpha'(x) = k\} \cup \{\mathtt{l} \mapsto a\} \models \Delta$. We use $|\Delta|$ to denote the size of $[\![\Delta]\!]$. For a word $w \in \Sigma^*$, we define inductively that a configuration α' is a *w-successor* of α, written $\alpha \xrightarrow{w} \alpha'$, such that $\alpha \xrightarrow{\epsilon} \alpha$ for all $\alpha \in C$, and $\alpha \xrightarrow{av} \alpha'$ iff $\alpha \xrightarrow{a} \bar{\alpha} \xrightarrow{v} \alpha'$ for some $\bar{\alpha} \in C$, $a \in \Sigma$, and $v \in \Sigma^*$. A configuration α is *initial* or *final* if $\alpha \models I$ or $\alpha \models F$, respectively. The *outcome* of T on a word w is the set $out_T(w)$ of all w-successors of the initial configurations, and w is accepted by T if $out_T(w)$ contains a final configuration. The *language* $\mathcal{L}(T)$ of T is the set of all words that T accepts.

Counting automata. A *counting variable (counter)* is a configuration variable c whose value ranges over $\mathbb{N}$ and which can appear (within Δ, I, and F) only in *atomic counter guards* of the form $c \leq k, c \geq k$, (using $<, =, >$ as syntactic sugar) or *term equality tests* $t_1 = t_2$, and in *atomic counter assignments* $c' = t$ with t, t_1, t_2 being *arithmetic terms* of the form $d + k$ or k with $k \in \mathbb{N}$ and d being a counter. A *control state variable* is a variable $\mathtt{s}$ whose value ranges over states Q and appears only in *atomic state guards* $\mathtt{s} = q$ and *atomic state assignments* $\mathtt{s}' = q$ for $q \in Q$. A Boolean combination of atomic guards (counter, state, or symbol) is a *guard formula* and a Boolean combination of atomic assignments is an *assignment formula*.

A *(nondeterministic) counting automaton* (CA) is a tuple $A = (Q, C, I, F, \Delta)$ such that (Q, V, I, F, Δ) is an LTS with the following properties: (1) The set of configuration variables $V = C \cup \{\mathtt{s}\}$ consists of a set of counters C and a single control state variable $\mathtt{s}$ s.t. $\mathtt{s} \notin C$. (2) The transition formula Δ is a disjunction of *transitions*, which are conjunctions of the form $\mathtt{s} = q \wedge g \wedge f \wedge \mathtt{s}' = r$, denoted by $q\text{-}\{g,f\}\!\mapsto r$, where $q, r \in Q$, g is the transition's *guard formula* over $V \cup \{\mathtt{l}\}$, and f is the transition's *counter assignment formula*, a conjunction of atomic assignments to counters, in which every counter is assigned at most once. (3) There is a constant $\boldsymbol{max}_A \in \mathbb{N}$ such that no counter can ever grow above that value, i.e., $\forall c \in C\ \forall w \in \Sigma^*\ \forall \alpha \in out_T(w) : \alpha \models c \leq \boldsymbol{max}_A$.

The last condition in the definition of CAs is semantic and can be achieved in different ways in practice. For instance, regular expressions can be compiled to CAs where assignment terms are of the form $c + 1$, 0, or c only, and every appearance of $c + 1$ is paired with a guard containing a constraint $c \leq k$ for some $k \in \mathbb{N}$. In this case, $\boldsymbol{max}_A = K + 1$ where K is the maximum constant used in the guards of the form $c \leq k$.

[3] To handle large or infinite sets of symbols symbolically, the predicates $\mathtt{l} = a$ may be generalised to predicates from an arbitrary effective Boolean algebra, as in [6].

We will often consider the initial and final formulae of CAs given as a disjunction $\bigvee_{q\in Q}(\mathtt{s} = q \wedge \varphi_q)$ where φ_q is a formula over counter guards, in which case we write $I(q)$ or $F(q)$ to denote the disjunct φ_q of the initial or final formula, respectively. An example of a CA is given in Fig. 1.

A *deterministic counting automaton* (DCA) is a CA A where I has at most one model and, for every symbol $a \in \Sigma$, every reachable configuration α has at most one a-successor (equivalently, the outcome of every word in A is either a singleton or the empty set). Finally, in the special case when $C = \emptyset$, the CA is a (classical) nondeterministic *finite automaton* (NFA), or a deterministic finite automaton (DFA) if it is deterministic.

3 Determinisation of Counting Automata

In this section, we discuss an algorithm for determinising CAs. A naïve determinisation converts a given CA A into an NFA by hard-wiring counter configurations as a part of control states, followed by the classical subset construction to determinise the obtained NFA (the NFA is finite due to the bounds on the maximum values of counters). The state space of the obtained DFA then consists of all reachable outcomes of A. By determinising A in this way, the succinctness of using counters is lost, and the size of the DFA can explode exponentially not only in the number of control states of A but also in the number of reachable counter valuations, which makes the construction impractical. Instead, our construction will retain counters (though their number may grow) and represent possible word outcomes as configurations of the resulting DCA.

Spheres. In particular, the outcome of a word $w \in \Sigma^*$ in a CA $A = (Q, C, I, F, \Delta)$ can be represented as a formula φ over equalities of the form $c = k$ and $\mathtt{s} = q$ where $q \in Q$, $c \in C$, $k \in \mathbb{N}$. Intuitively, disjunctions can be used to obtain a single formula for the possibly many configurations reachable in A over w. For example, the outcome of the word $\mathtt{aab}$ in Fig. 1 is $\varphi : \mathtt{s} = q \vee (\mathtt{s} = r \wedge (c = 1 \vee c = 2))$. Generally, the outcome of $\mathtt{aab}^i$, for $0 \leq i < k$, assuming $k > 2$, is $\varphi_i : \mathtt{s} = q \vee (\mathtt{s} = r \wedge (c = i \vee c = i + 1))$.

A crucial notion for our construction is then the notion of *sphere*. A sphere ψ arises from an outcome φ by replacing the constants from $\mathbb{N}$ by parameters drawn from a countable set $\mathcal{P}$ (disjoint from $\mathbb{N}$, V, Q, and $\{\mathtt{l}, \mathtt{s}\}$). In the example above, the sphere obtained from the φ is $\psi : \mathtt{s} = q \vee (\mathtt{s} = r \wedge (c = p_0 \vee c = p_1))$, and the same sphere arises from all outcomes φ_i with $0 \leq i < k$.

Spheres will play the role of the control states of the resulting DCA. The idea of the construction is that the outcome of every word w in a DCA A^d will contain a single configuration (A^d is deterministic) consisting of a sphere ψ as the control state and a valuation of its parameters $\eta : \mathcal{P} \to \mathbb{N}$. The construction will ensure that $\psi[\eta]$ models the outcome $out_A(w)$ of w in A. In our example, the outcome of $\mathtt{aab}$ in A^d would contain the single configuration $\{\mathtt{s} \mapsto \psi, p_0 \mapsto 1, p_1 \mapsto 2\}$, and the outcome of each φ_i, for $0 \leq i < k$, would contain the single configuration $\{\mathtt{s} \mapsto \psi, p_0 \mapsto i, p_1 \mapsto i + 1\}$. The example shows the advantage of our construction. Every outcome φ_i would be a control state of the naïvely determinised automaton, with a b-transition from each φ_j to φ_{j+1}, for $0 \leq j < k - 1$. In contrast to that, all these states and transitions will be in A^d replaced by a single control state ψ with a single b-labelled self-loop that increments both p_0 and p_1. This structure can be seen in Fig. 2 (states are spheres, labelled by their multiset representation introduced in Section 4.2).

3.1 Determinisation by Sphere Construction

We now provide a basic version of our sphere-based determinisation, which can also be viewed as an algorithm that constructs parametric versions of the subsets used in subset-based determinisation. For this basic algorithm, termination is not guaranteed, but it serves as a basis on which we will subsequently build a terminating algorithm. Let us first introduce some needed additional notation.

Given a formula φ, we denote by $at(\varphi)$ and by $num(\varphi)$ the sets of assignment terms and numerical constants, respectively, appearing in φ. We will use the set $\mathcal{P}' = \{p' \mid p \in \mathcal{P}\}$ and the substitution $\theta_{unprime} = \{p' \mapsto p \mid p \in \mathcal{P}\}$. We say that a formula over variables $V \cup V' \cup \{1\} \cup \mathcal{P}$ is *factorised wrt guards* if it is a disjunction $\bigvee_{i=1}^{n}(g_i) \wedge (u_i)$ of factors, each consisting of a guard g_i over $V \cup \{1\} \cup \mathcal{P}$ and an update formula u_i over atomic assignments such that the guards of any two different factors are mutually exclusive, i.e., $g_i \wedge g_j$ is unsatisfiable for any $1 \leq i \neq j \leq n$.[4] For a set of variables U, we denote by $\exists\!\!\exists U : \varphi$ a formula obtained by eliminating all variables in U from φ (i.e., a quantifier-free formula equivalent to $\exists U : \varphi$).[5]

The algorithm. The core of our determinisation algorithm is the sphere construction described in Algorithm 1. It builds a DCA $A^d = (Q^d, P, I^d, F^d, \Delta^d)$ whose control states Q^d are spheres. Its counters are parameters from the set P that is built during the run of the algorithm. The initial formula I^d defined on line 3 assigns to s the initial control state ψ_I (obtained on line 2), which is a parametric version of I with integer constants replaced by parameters according to the renaming θ_{const}. Moreover, I^d also equates the parameters in ψ_I with the constants they are replacing in I. Hence, the formula $\psi_I[\theta_{const}^{-1}]$ models exactly the initial configurations of A.

Example 1. In the running example (Fig. 1), whenever referring to some variable that is assigned multiple times during the run of the algorithm, we use superscripts to distinguish the different assignments during the run. On lines 1–4, the initial sphere ψ_I is assigned the formula $\mathsf{s} = q$, and the initial formula I^d is set to $\mathsf{s} = \psi_I$, which specifies that ψ_I is indeed the initial control state only (I does not constrain counters, hence I^d does not talk about parameters). □

The remaining states of Q^d and transitions of Δ^d are computed by a worklist algorithm on line 5 with the worklist initialised with ψ_I. Every iteration computes the outgoing transitions of a control state $\psi \in \mathit{Worklist}$ as follows: On line 7, after

[4] A Boolean combination of atomic guards and updates can be factorised through (1) a transformation to DNF, yielding a set of clauses X; (2) writing each clause $\varphi \in X$ as a conjunction of a guard formula g_φ and an assignment formula f_φ; (3) computing minterms of the set $\{g_\varphi \mid \varphi \in X\}$; (4) creating one factor $(g) \wedge (f)$ from every minterm g where f is the disjunction of all the assignment formulae f_φ with $\varphi \in X$ compatible with g (i.e., such that $g \wedge f_\varphi$ is satisfiable).

[5] We note that we only need to use a specialised, simple, and cheap quantifier elimination. In particular, we only need to eliminate counter variables c from formulae such that, in clauses of their DNF, c always appears together with a predicate $c = p$ where p is a parameter. Eliminating c from such a DNF clause is then done by simply substituting occurrences of c by p. We do not need complex algorithms such as the general quantifier elimination for Presburger arithmetic.

Algorithm 1: Sphere-based CA determinisation (non-terminating)

```
   Input: A CA $A = (Q, C, I, F, \Delta)$.
   Output: A DCA $A^d = (Q^d, P, I^d, F^d, \Delta^d)$ s.t. $\mathcal{L}(A) = \mathcal{L}(A^d)$.
1  $Q^d \leftarrow Worklist \leftarrow \emptyset$; $\Delta^d \leftarrow \bot$;
2  $\psi_I \leftarrow I[\theta_{const}]$ for some total injection $\theta_{const} : num(I) \rightarrow \mathcal{P}$;
3  $I^d \leftarrow \mathtt{s} = \psi_I \wedge \varphi_{\theta_{const}}$;
4  add $\psi_I$ to $Q^d$ and to $Worklist$;
5  while $Worklist \neq \emptyset$ do
6      $\psi \leftarrow pop(Worklist)$;
7      Let $\bigvee_{i=1}^{n} (g_i) \wedge (u_i)$ be the formula $\exists C, \mathtt{s} : \psi \wedge \Delta$ factorised wrt guards;
8      foreach $1 \leq i \leq n$ do
9          $\psi_i \leftarrow u_i[\theta_{at}][\theta_{unprime}]$ for a total injection $\theta_{at} : at(u_i) \rightarrow \mathcal{P}'$;
10         add $\psi \mathrel{-\{g_i, \varphi_{\theta_{at}}\}\!\!\rightarrow} \psi_i$ to $\Delta^d$;
11         if $\psi_i \notin Q^d$ then add $\psi_i$ to $Q^d$ and to $Worklist$ ;
12 $P \leftarrow$ all parameters found in $Q^d$;
13 $F^d \leftarrow \bigvee_{\psi \in Q^d} \mathtt{s} = \psi \wedge \exists C, \mathtt{s} : \psi \wedge F$;
14 $I^d \leftarrow ground(I^d)$; $\Delta^d \leftarrow ground(\Delta^d)$;
15 return $A^d = (Q^d, P, I^d, F^d, \Delta^d)$;
```

eliminating $C \cup \{\mathtt{s}\}$ from the formula $\psi \wedge \Delta$, which describes how the next state and counter values depend on the input symbol and the current values of parameters, it is transformed into a guard-factorised form.

Example 2. When ψ_I is taken from $Worklist$ as ψ^1 on line 6, its processing starts by factorising $\exists\{c, \mathtt{s}\} : \psi^1 \wedge \Delta$ on line 7. Here, $\psi^1 \wedge \Delta$ is the formula $\mathtt{s} = q \wedge (q \mathrel{-\{\top,\top\}\!\!\rightarrow} q \vee q \mathrel{-\{\mathtt{l}=\mathtt{a}, c'=0\}\!\!\rightarrow} r \vee r \mathrel{-\{c<k, c'=c+1\}\!\!\rightarrow} r)$, which can be also written as

$$\mathtt{s} = q \wedge (\mathtt{s}' = q \vee (\mathtt{l} = \mathtt{a} \wedge c' = 0 \wedge \mathtt{s}' = r))\,.$$

The elimination of $\{c, \mathtt{s}\}$ gives the formula $\mathtt{s}' = q \vee (\mathtt{l} = \mathtt{a} \wedge c' = 0 \wedge \mathtt{s}' = r)$. This formula is factorised into the following two factors:

(F_1) $(\mathtt{l} = \mathtt{a}) \wedge (\mathtt{s}' = q \vee (c' = 0 \wedge \mathtt{s}' = r))$,
(F_2) $(\mathtt{l} \neq \mathtt{a}) \wedge (\mathtt{s}' = q)$. □

In the for-loop on line 8, every factor $(g_i) \wedge (u_i)$ is turned into a transition with the guard g_i; the mutual incompatibility of the guards guarantees determinism. The formula u_i describes the target sphere in terms of the parameters of the source sphere ψ, updated according to the transition relation. That is, it is a Boolean combination of assignments of the form $c' = p + k$ or $c' = k$ for $c \in C, p \in \mathcal{P}$, and $k \in \mathbb{N}$. Line 9 creates a sphere by substituting each of the assignment terms (of the form $p + k$ or k) with a parameter and replacing primed variables by their unprimed versions.[6] The corresponding assignment term substitution θ_{at} records how the values of the new parameters are obtained from the original values of the parameters occurring in ψ. It is used to define the assignment

[6] The choice of the parameters in the image of $\theta_{at} : at(u_i) \rightarrow \mathcal{P}'$ on line 9 is arbitrary, although, in practice, it would be sensible to define some systematic parameter naming policy and reuse existing parameters whenever possible.

formula of the new transition that is added to Δ^d on line 10. The argument justifying that the construction preserves the language is the following: if reading $w \in \Sigma^*$ takes A^d to ψ with a parameter valuation η such that $\psi[\eta]$ is equivalent to $out_A(w)$, then reading a next symbol a using a transition newly created on line 10 takes A^d to ψ' with the parameter valuation η' such that $\psi'[\eta']$ models $out_A(wa)$.

Example 3. Factor F_1 of Example 2 above is processed as follows. A possible choice for θ^1_{at} on line 9 is the assignment $\{0 \mapsto p_0\}$. Its application followed by $\theta_{unprime}$ creates

$$\psi^1_1 : \mathtt{s} = q \vee (c = p_0 \wedge \mathtt{s} = r).$$

From θ^1_{at}, we get the substitution formula $\varphi_{\theta^1_{at}} : (p'_0 = 0)$ on line 10, and so the transition added to Δ^d is $(\mathtt{s} = q) \xrightarrow{\{\mathtt{l}=\mathtt{a}, p'_0=0\}} (\mathtt{s} = q \vee (c = p_0 \wedge \mathtt{s} = r))$. The target ψ^1_1 of the transition is added to Q^d and to *Worklist* on line 11. Next, Factor F_2 generates the self-loop $(\mathtt{s} = q) \xrightarrow{\{\mathtt{l}\neq\mathtt{a}, \top\}} (\mathtt{s} = q)$, which ends the first iteration of the while-loop.

Let us also walk through a part of the second iteration of the while-loop, in which ψ^1_1 is taken from *Worklist* as ψ^2 on line 6. The formula $\psi^2 \wedge \Delta$ from line 7 is $((\mathtt{s} = r \wedge c = p_0) \vee \mathtt{s} = q) \wedge (q \xrightarrow{\{\top,\top\}} q \vee q \xrightarrow{\{\mathtt{l}=\mathtt{a}, c'=0\}} r \vee r \xrightarrow{\{c<k, c'=c+1\}} r)$, which is equivalent to $(\mathtt{s} = q \wedge (\mathtt{s}' = q \vee (\mathtt{l} = \mathtt{a} \wedge c' = 0 \wedge \mathtt{s}' = r))) \vee \big(\mathtt{s} = r \wedge c = p_0 \wedge c < k \wedge c' = c + 1 \wedge \mathtt{s}' = r\big)$. The elimination of $\{c, \mathtt{s}\}$ on line 7 then gives the formula $\big(\mathtt{s}' = q \vee (\mathtt{l} = \mathtt{a} \wedge c' = 0 \wedge \mathtt{s}' = r)\big) \vee (p_0 < k \wedge c' = p_0 + 1 \wedge \mathtt{s}' = r)$, which is factorised into the following four factors:

(F_3) $(\mathtt{l} = \mathtt{a} \wedge p_0 < k) \wedge (\mathtt{s}' = q \vee (c' = 0 \wedge \mathtt{s}' = r) \vee (c' = p_0 + 1 \wedge \mathtt{s}' = r))$,
(F_4) $(\mathtt{l} \neq \mathtt{a} \wedge p_0 < k) \wedge (\mathtt{s}' = q \vee (c' = p_0 + 1 \wedge \mathtt{s}' = r))$,
(F_5) $(\mathtt{l} = \mathtt{a} \wedge p_0 \geq k) \wedge (\mathtt{s}' = q \vee (c' = 0 \wedge \mathtt{s}' = r))$, and
(F_6) $(\mathtt{l} \neq \mathtt{a} \wedge p_0 \geq k) \wedge (\mathtt{s}' = q)$.

In the for-loop on line 8, Factor F_3 is processed as follows. Let the chosen substitution θ^2_{at} on line 9 be $\{p_0 + 1 \mapsto p_1, 0 \mapsto p_0\}$. Its application followed by $\theta_{unprime}$ generates

$$\psi^2_1 : \mathtt{s} = q \vee (c = p_0 \wedge \mathtt{s} = r) \vee (c = p_1 \wedge \mathtt{s} = r).$$

The substitution formula $\varphi_{\theta^2_{at}}$ on line 10 is $p'_1 = p_0 + 1 \wedge p'_0 = 0$, and so Δ^d gets the new transition $\psi^1_1 \xrightarrow{\{\mathtt{l}=\mathtt{a} \wedge p_0 < k, p'_1 = p_0 + 1 \wedge p'_0 = 0\}} \psi^2_1$. The evaluation of the while-loop would continue analogously. □

In the final stage of the algorithm, when (and if) the while-loop terminates, line 12 collects the set P of all parameters used in the constructed parametric spheres of Q^d as new counters of A^d. Further, line 13 derives the new final formula by considering all computed spheres, restricting them to valuations where the original final formula is satisfied, and quantifying out the original counters. This way, final constraints on the original counters get translated to constraints over parameters in P.

Example 4. In our running example, for the spheres discussed above, we would have $F(\psi^1) : \bot$, $F(\psi^1_1) : p_0 = 1$, and $F(\psi^2_1) : p_0 = 1 \vee p_1 = 1$. □

Finally, line 14 applies the function *ground* on the initial formula and the transition formula of the constructed automaton before returning it. This step is needed in order to avoid nondeterminism on unused and unconstrained counters. The function *ground* conjuncts constraints of the form $p = 0$ with the initial formula and with the guard of every transition for every parameter $p \in P$ that is so far unconstrained in the concerned formula. Moreover, it will introduce a reset $p' = 0$ to the assignment formula of every transition for every counter $p \in P$ that is so far not assigned on the concerned transition. The while-loop of Algorithm 1 needs, however, not terminate, as witnessed also by our example.[7]

Example 5. Continuing in Example 4, the DCA in Fig. 2 would be a part of the DCA constructed by Algorithm 1, its states being the spheres ψ^1, ψ_1^1, ψ_1^2 from the left, but the while-loop would not terminate, with ψ_1^2. Instead, it would eventually generate a successor of ψ_1^2, the sphere

$$\psi_1^3 : \mathsf{s} = q \vee (c = p_0 \wedge \mathsf{s} = r) \vee (c = p_1 \wedge \mathsf{s} = r) \vee (c = p_2 \wedge \mathsf{s} = r),$$

i.e., a sphere similar to ψ_1^2 but extended by a new disjunct with a new parameter p_2. Repeating this, the algorithm would keep generating larger and larger spheres with more and more parameters. □

3.2 Ensuring Termination of the Sphere Construction

In this section, we will discus reasons for possible non-termination of Algorithm 1 and a way to tackle them. The main reason is that the algorithm may generate unboundedly many parameters that correspond to different histories of a counter c when processing the input word (including also impossible ones in which the counter exceeds the maximum value). The algorithm indeed "splits" a parameter appearing in a sphere into two parameters in the successor sphere when the transitions of A update the counter in two different ways.

In our terminating version of Algorithm 1, we build on the following: (1) distinguishing between histories that converge in the same counter value is not necessary, they can be "merged", and (2) the number of different reachable counter values is bounded (by the definition of CAs). We thus enforce the invariant of every reachable configuration of A^d that all parameters in the configuration have distinct values. The invariant is enforced by testing equalities of parameters and merging parameters with equal values on transitions of A^d. All transitions of A^d entering spheres with more than $\boldsymbol{max}_A + 1$ parameters can then be discarded because the invariant implies that they cannot be taken at any configuration of A^d. Furthermore, we will also ensure that the algorithm does not diverge because of generating semantically equivalent but syntactically different spheres (because of different names of parameters or different formulae structure).

A terminating determinisation of CAs is obtained from Algorithm 1 by replacing lines 9–11 by the code in Algorithm 2. In order to ensure that parameters have pairwise distinct values, the transitions of A^d test equalities of the values assigned to parameters and ensure that two parameters are never used to represent the same value. Different histories of counters are thus merged if they converge into the same value. To achieve

[7] For this step to preserve the language of the automaton, we need to assume that the input CA does not assign nondeterministic values to live counters. We are refering to the standard notion: a counter is live at a state if the value it holds at that state may influence satisfaction of some guard in the future. Any CA can be transformed into this form, and CAs we compile from regular expressions satisfy this condition by construction.

Algorithm 2: Ensuring termination of sphere-based CA determinisation

```
16 foreach equivalence $\sim$ on $at(u_i)$ s.t. $\mathtt{sat}(\varphi_\sim)$ and $|at(u_i)/_\sim| \leq \boldsymbol{max}_A + 1$ do
17     let $\theta_{at} : at(u_i) \rightarrow \mathcal{P}'$ be an injection;
18     $\psi_i \leftarrow u_i[\theta_{at}][\theta_{unprime}]$;
19     if $\exists \theta_{rename} : \mathcal{P} \leftrightarrow \mathcal{P}\ \exists \sigma \in Q^d : \psi_i[\theta_{rename}] \Leftrightarrow \sigma$ then
20         add $\psi \text{-}\{g_i \wedge \varphi_\sim[\theta_{at}], \varphi_{\theta_{at}}[\theta'_{rename}]\}\!\!\rightarrow \sigma$ to $\Delta^d$;
21     else
22         add $\psi \text{-}\{g_i \wedge \varphi_\sim[\theta_{at}], \varphi_{\theta_{at}}\}\!\!\rightarrow \psi_i$ to $\Delta^d$;
23         add $\psi_i$ to $Q^d$ and to $Worklist$;
```

this, Algorithm 2 enumerates all feasible equivalences of the assignment terms of u_i on line 16 and generates successor transitions for each of them separately. When deciding whether an equivalence $\sim$ on the assignment terms is feasible, the algorithm performs two tests: (1) The formula $\varphi_\sim \overset{\text{def}}{=} \bigwedge_{t_1 \sim t_2, t_1, t_2 \in at(u_i)} (t_1 = t_2) \wedge \bigwedge_{t_1 \nsim t_2, t_1, t_2 \in at(u_i)} (t_1 \neq t_2)$ is tested for satisfiability, meaning that the equivalence is not trying to merge terms that can never be equal (such as, e.g., p and $p + 1$). (2) The number of equivalence classes should be at most $\boldsymbol{max}_A + 1$ since this is the maximum number of different values that the counters can reach due to the requirement that the values must be between 0 and $\boldsymbol{max}_A$.

Line 17 builds a term assignment replacement θ_{at} that maps all $\sim$-equivalent terms to the same (future) parameter, and line 18 computes the target sphere, reflecting the given merge. The test on line 19 checks whether the target sphere is equal to some already generated sphere up to a parameter renaming (represented by a bijection $\theta_{rename} : \mathcal{P} \leftrightarrow \mathcal{P}$). If so, the created sphere is discarded, and a new transition going to the old sphere is generated on line 20; we need to rename the primed parameters used in the transition's assignment appropriately according to $\theta'_{rename} = \{p'_0 \mapsto p'_1 \mid p_0 \mapsto p_1 \in \theta_{rename}\}$. Otherwise, a transition into the new sphere is added on line 22, and the new sphere is added to Q^d and $Worklist$. In both cases, the guard of the generated transition is extended by the formula $\varphi_\sim[\theta_{at}]$, which encodes the equivalence $\sim$, and hence explicitly enforces that $\sim$ holds when the transition is taken.

Note that the test on the maximum number of equivalence classes can be optimised if finer information about the maximum reachable values of the individual counters is available. Such information can be obtained, e.g., by looking at the constants used in the guards of the transitions where the different counters are increased. For any counter, one should then not generate more parameters representing its possible values than what the upper bound on that counter is (plus one).

Theorem 1. *Algorithm 1 with the modification presented in Algorithm 2 terminates and produces a DCA with $\mathcal{L}(A) = \mathcal{L}(A^d)$ and $|Q^d| \leq 2^{|Q| \cdot (\boldsymbol{max}_A + 1)^{|C|}}$.*

Proof (idea). The fact that the algorithm indeed constructs a DCA is because line 7 of Algorithm 1 generates pairwise incompatible guards on transitions only. It is also easy to show by induction on the length of the words that the language is preserved. The termination then follows from the facts that (1) the algorithm has a bound on the maximum number of parameters in spheres (ensured by the condition over $\sim$ on line 16 of Algorithm 2) and (2) no spheres equal up to renaming are generated (ensured by the check on line 19). The bound on the size follows from the structure of spheres. □

The number of equivalences generated on line 16 of Algorithm 2 (and therefore also the number of transitions leaving ψ) may be large. Many of them are, however, infeasible (cannot be taken in any reachable configuration of A^d), and could be removed. In most cases, the majority of such infeasible transitions may be identified locally, taking advantage of the invariant of all reachable configurations of A^d enforced by Algorithm 2: namely, values of distinct parameters are always pairwise distinct. Therefore, before building a transition for an equivalence $\sim$, we ask whether the $\sim$-equivalent assignment terms may indeed be made equivalent assuming that the constructed transition guard g_i and—importantly—also the distinctness invariant hold right before the transition is taken. Technically, we create new transitions only from those equivalences $\sim$ such that $\mathtt{sat}(\bigwedge_{p_1,p_2 \in P_\psi, dist(p_1,p_2)}(p_1 \neq p_2) \land g_i \land \varphi_\sim)$ where P_ψ is the set of parameters of ψ and $dist(p_1, p_2)$ holds iff p_1 and p_2 are distinct parameters.

3.3 Reachability-Restricted CA Determinisation

Above, we have described a terminating algorithm for CA determinisation. While it is witnessed by our experiments that the algorithm often generates much smaller automata than what could be obtained by transforming the automata into NFAs and determinising them, a natural question is whether the generated DCA is *always* smaller or equal in size to the DFA built by getting rid of the counters and using classical determinisation. Unfortunately, the answer to this question is no. The reason is that the transformation to a DCA needs not recognise that some generated transitions can never be executed and that some spheres are not reachable. To see this, it is enough to imagine a transition setting some counter c to zero and the only successor transition testing whether c is positive. The latter transition would not be executed when generating the DFA due to working with concrete values of counters, but it would be considered when constructing the DCA (since the construction does not know the values of the counters).

In our experiments with CAs obtained from real-life regexes, the above was not a problem, but we note that, for the price of an increased cost of the construction, one could further improve the algorithm by taking into account some reachability information. In an extreme case, one could first generate the DFA corresponding to the given CA and then use it when generating the DCA (as a hopefully more compact representation of the DFA). In particular, whenever adding some new sphere into the DCA being built, the algorithm can check whether there is a subset of states in the original CA represented as a state of the DFA that is an instance of the sphere. If not, the sphere is not added. The resulting DCA can then never be bigger than the DFA since each control state of the DFA (i.e., a subset of states of the original CA) is represented by a single sphere only, likewise each transition of the DFA is represented by a single transition of the DCA, and there are not any unreachable spheres or transitions that cannot be executed.

Notice that the reachability pruning is an alternative to Algorithm 2. Algorithm 1 equipped with the reachability analysis is guaranteed to terminate. For example, when run on the CA in Fig. 1, it would generate a DCA isomorphic to that from Fig. 2.

4 Monadic Counting

We now provide a simplified and more efficient version of the determinisation algorithm. The simplified version targets CAs that naturally arise from *monadic regexes*, i.e., regular expressions extended with counting limited to *character classes*. Their abstract syntax is

$$R ::= \emptyset \mid \varepsilon \mid \sigma \mid R_1R_2 \mid R_1 + R_2 \mid R{*} \mid \sigma\{n,m\}$$

where σ is a predicate denoting a set of alphabet symbols, i.e., a *character class* (σ will be used to denote character classes from now on), and $n, m \geq 0$ are integers. The semantics is defined as usual, with $\sigma\{n,m\}$ denoting a string w with $n \leq |w| \leq m$ symbols satisfying σ.

The specialised determinisation algorithm is of a high practical relevance since the monadic class is very common, as witnessed by our experiments, where it covers over 95 % of the regexes with counting that we found (cf. Section 5).

4.1 Monadic Counting Automata

Monadic regexes can be easily compiled to nondeterministic monadic CAs satisfying certain structural properties summarised below.[8] In particular, a (nondeterministic) *monadic counting automaton (MCA)* is a CA $A = (Q, C, I, F, \Delta)$ where the following holds:

1. The set Q of control states is partitioned into a set of *simple states* Q_s and a set of *counting states* Q_c, i.e., $Q = Q_s \uplus Q_c$.

2. The set of counters $C = \{c_q \mid q \in Q_c\}$ consists of a unique counter c_q for every counting state $q \in Q_c$.

3. All transitions containing counter guards or updates must be incident with a counting state in the following manner. Every counting state $q \in Q_c$ has a single *increment transition*, a self-loop $q\text{-}\{\sigma \wedge c_q < \boldsymbol{max}_q, c'_q = c_q + 1\}\!\!\rightarrow q$ with the value of c_q limited by the *bound* $\boldsymbol{max}_q$ of q, and possibly several *entry transitions* of the form $r\text{-}\{\sigma \wedge c'_q = 0\}\!\!\rightarrow q$, which set c_q to 0. As for *exit transitions*, every counting state is either *exact* or *range*, where exact counting states have exit transitions of the form $q\text{-}\{\sigma \wedge c_q = \boldsymbol{max}_q\}\!\!\rightarrow s$, and *range* counting states have exit transitions of the form $q\text{-}\{\sigma, \top\}\!\!\rightarrow s$ with $s \in Q$ s.t. $s \neq q$. That is, an exact counting state may be left only after exactly $\boldsymbol{max}_q$ repetitions of the incrementing transition (it corresponds to a regular expression σ`{k}`), while a range counting state may be left sooner (it corresponds to a regular expression σ`{0,k}`). We denote the set of range counting states Q_r and the set of exact counting states Q_e, with $Q_c = Q_r \uplus Q_e$.

4. The initial condition I is of the form $I : \bigvee_{q \in Q_s^I} \mathsf{s} = q \vee \bigvee_{q \in Q_c^I} (\mathsf{s} = q \wedge c_q = 0)$ for some sets of initial simple and counting states $Q_s^I \subseteq Q_s$ and $Q_c^I \subseteq Q_c$, respectively, with the counters of initial counting states initialised to 0.

5. The final condition F is of the form $F : \bigvee_{q \in Q_s^F \cup Q_r^F} \mathsf{s} = q \vee \bigvee_{q \in Q_e^F} (\mathsf{s} = q \wedge c_q = \boldsymbol{max}_q)$ where $Q_s^F \subseteq Q_s$ is a set of simple final states, $Q_r^F \subseteq Q_r$ is a set of final range counting states, and $Q_e^F \subseteq Q_e$ is a set of final exact counting states. That is, final conditions on final states are the same as counter conditions on exit transitions.[9]

[8] We note that we restrict ourselves to range sub-expressions of the form $\sigma\{n,n\}$ or $\sigma\{0,n\}$ only. This is without loss of generality since a general range expression $\sigma\{m,n\}$ can be rewritten as $\sigma\{m,m\}.\sigma\{0,n-m\}$.

[9] Notice that the guards $c_q < \boldsymbol{max}_q$ on the incrementing self-loops of exact counting states could be removed without affecting the language since when c_q exceeds $\boldsymbol{max}_q$, then the run can never

4.2 Determinisation of MCAs

Algorithm 2 can be simplified when specialised to monadic CAs. The simplification is based on the following observations. *Observation 1. Counters are dead outside their states.* To simplify the representation of spheres, we use the fact that every counter c_q of an MCA is "active" in the state q only, while c_q is "dead" in other states (i.e., its current value has no influence on runs of the MCA that are not in q). To represent different variants of c_q, we use parameters of the form $c_q[i]$ obtained by indexing c_q by an index i, for $0 \leq i \leq \boldsymbol{max}_q$, while enforcing the invariant that, for distinct indices i and j, $c_q[i]$ and $c_q[j]$ always have different values. Since the value of c_q ranges from 0 to $\boldsymbol{max}_q$, at most $\boldsymbol{max}_q + 1$ variants of c_q are needed.[10] Since spheres only need parameters to remember values of live counters, every sphere can be equivalently written in the *normal form*

$$\psi \overset{\text{def}}{=} \bigvee_{q \in Q'_s} \mathsf{s} = q \vee \bigvee_{q \in Q'_c} \Big(\mathsf{s} = q \wedge \bigvee_{0 \leq i \leq \boldsymbol{max}'_q} c_q = c_q[i]\Big)$$

for some $Q'_s \subseteq Q_s$, $Q'_c \subseteq Q_c$, and $\boldsymbol{max}'_q \leq \boldsymbol{max}_q$. That is, a sphere ψ records which states may be reached in the original MCA when ψ is reached in the determinised MCA and also which variants of the counter c_q may record the value of c_q when q is reached.

Observation 2. Variants of exact counting states can be sorted. For dealing with any exact counting state $q \in Q_e$, we may use the following facts: (1) If executed, the increment transition of q increments all variants of c_q whose values are smaller than $\boldsymbol{max}_q$. (2) New variants of c_q are initialised to 0 by the entry transitions. (3) Variants whose value is $\boldsymbol{max}_q$ can take an exit transition, after which they become dead and their values do not need to be propagated to the next configuration. It is therefore easy to enforce that the values of the variants $c_q[i]$ stay sorted, so that $i < j$ implies $\alpha(c_q[i]) < \alpha(c_q[j])$ in every configuration α of A^d. The sortedness invariant implies that the variant of c_q with the highest index, called *highest variant*, has the highest value. This, together with the invariant of boundedness by $\boldsymbol{max}_q$ and mutual distinctness of values of variants of c_q, means that the highest variant is the only one that may satisfy the tests $c_q = \boldsymbol{max}_q$ on exit transitions or fail the test $c_q < \boldsymbol{max}_q$ on the incrementing transition. Hence, the deterministic MCA does not need to test all variants of c_q but the highest one only.

Observation 3. Only the smallest variants of range counting states are important. For range counting states, we adapt the *simulation pruning* technique from [10]. The technique optimizes the standard subset-construction-based determinisation of NFAs by exploiting a *simulation* relation [7] such that any *macrostate* (which has the form of a set of states of the original NFA) obtained during the determinisation can be pruned by removing those NFA states that are simulated by other NFA states included in the same macrostate. The pruning does not change the language: the resulting DFA is bisimilar to the one constructed without pruning. For our DCA construction, we use the simulation that implicitly exists between configurations α and α' of A with the same range counting

leave q and has thus no chance of accepting. We include these guards only to conform to the condition on boundedness of counter values in the definition of CAs.

[10] Notice that maintaining a fixed association of a parameter to a counter is a difference from Algorithms 1 and 2, where one parameter may represent different counters.

state $q = \alpha(\mathsf{s}) = \alpha'(\mathsf{s})$, where $\alpha(c_q) \geq \alpha'(c_q)$ implies that α' simulates α.[11] Hence, the spheres only need to remember the smallest possible counter value for every range counting state q, which may be always stored in $c_q[0]$, and discard all other variants.

Determinisation of MCAs. Observations 1–3 above allow for representing spheres using a simple data structure, namely, a multiset of states. By a slight abuse of notation, we use ψ for the sphere itself as well as for its multiset representation $\psi : Q \to \mathbb{N}$. The fact that $\psi(q) > 0$ means that q is present in the sphere (i.e., $\mathsf{s} = q$ is a predicate in the normal form of ψ), and for a counting state q, the counters $c_q[0], \ldots, c_q[\psi(q) - 1]$ are the $\psi(q)$ variants of c_q tracked in the sphere (i.e., $\psi(q) - 1 = \boldsymbol{max}'_q$ in the normal form of ψ).

The MCA determinisation is then an analogy of Algorithm 1 that uses the multiset data structure and preserves the sortedness and uniqueness of variants of exact counters. The initial sphere ψ_I assigns 1 to all initial states of I, and the initial configuration I^d assigns 0 to $c_q[0]$ for each counting state q in I. Further, we modify the part of Algorithm 1 after popping a sphere ψ from *Worklist* in the main loop (lines 7–11).

Let Δ_ψ denote the set of transitions of A originating from states q with $\psi(q) > 0$. Processing of ψ starts by removing guard predicates of the form $c_q < \boldsymbol{max}_q$ from increment transitions of exact counting states in Δ_ψ (since they have no semantic effect as mentioned already above). Subsequently, we compute minterms of the set of guard formulae of the transitions in Δ_ψ. Each minterm μ then gives rise to a transition $\psi \text{-}\{g,f\}\!\!\rightarrow \psi'$ of A^d. The guard formula g, assignment formula f, and the target sphere ψ' are constructed as follows.

First, the guard g is obtained from the minterm μ by replacing, for all $q \in Q_c$, every occurrence of c_q by $c_q[\psi(q)]$, i.e., the highest variant of c_q. Intuitively, the counter guards of transitions of Δ_ψ present in μ will on the constructed transition of A^d be testing the highest variants of the counters. This is justified since (a) only the highest variant of c_q needs to be tested for exact counting states, as concluded in Observation 2 above, and (b) we keep only a single variant of c_q for range counting states (which is also the highest one), as concluded in Observation 3.

We then initialise the target multiset ψ' as the empty multiset $\{q \mapsto 0 \mid q \in Q\}$ and collect the set Δ_μ of all transitions from Δ_ψ that are compatible with the minterm μ (recall that increment self-loops of exact states in Δ_ψ have counter guards removed, hence counter guards do not influence their inclusion in Δ_μ). The transitions of Δ_μ will be processed in the following three steps.

Step 1 (simple states). Simple states with an incoming transition in Δ_μ get $\psi'(q) = 1$.

Step 2 (increment self-loops). For exact states with the increment self-loop in Δ_μ, $\psi'(q)$ is set to $\psi(q) - 1$ if an exit transition of q is in Δ_μ, and to $\psi(q)$ otherwise. Indeed, if (and only if) an exit transition of q is included in Δ_μ, and Δ_μ is enabled in some sphere, then the highest variant of c_q has reached $\boldsymbol{max}_q$ in that sphere, and the self-loop cannot be taken by the highest variant of c_q. The lower variants of c_q always have values smaller than $\boldsymbol{max}_q$, and hence can take the self-loop. The assignment f then gets the conjunct $c_q[i]' = c_q[i] + 1$ for each $0 \leq i < \psi'(q)$ since the variants that take the self-loop are

[11] The fact that this relation is indeed a simulation can be seen from that both the higher and lower value of c_q can use any exit transition of q at any moment regardless of the value of c_q, but the lower value of c_q can stay in the counting loop longer.

incremented. For range states with the increment self-loop in Δ_μ, we set $\psi'(q)$ to 1, and $c_q[0]' = c_q[0] + 1$ is added to f (only one variant is remembered).

Step 3 (entry transitions). For each counting state q with an entry transition in Δ_μ, $\psi'(q)$ is incremented by 1 and the assignment $c_q[0]' = 0$ of the fresh variant of c_q is added to f. If the new value of $\psi'(q)$ exceeds $\boldsymbol{max}_q + 1$, then the whole transition generated from μ is discarded, since c_q cannot have more than $\boldsymbol{max}_q + 1$ distinct values. Otherwise, if q is an exact counting state, then f is updated to preserve the invariant of sorted and unique values of c_q: the increments of older variants of c_q are *right-shifted* to make space for the fresh variant, meaning that each conjunct $c_q[i]' = c_q[i] + 1$ in f is replaced by $c_q[i+1]' = c_q[i] + 1$. If $q \in Q_r$, then if the assignment $c_q[0]' = c_q[0] + 1$ is present in f, it is removed (as the fresh variant has the smallest value 0).

Example 6. Determinising the CA from Fig. 1 using the algorithm described in this section would result in the DCA shown in Fig. 2. □

The monadic determinisation has a much lower worst-case complexity than the general algorithm. Importantly, the number of states depends on $\boldsymbol{max}_A$ only polynomially, which is a major difference from the exponential bounds of the naïve determinisation and our general construction.

Theorem 2. *The specialised monadic CA determinisation constructs a DCA with* $|Q^d| \leq (\boldsymbol{max}_A + 1)^{|Q|}$ *and* $|\Delta^d| \leq |\Sigma| \cdot (4 \cdot (\boldsymbol{max}_A + 1))^{|Q|}$.

Proof (idea). The bound on the number of states is given by the number of functions $Q \to \{0, \ldots, \boldsymbol{max}_A\}$. The bound on the number of transitions is given by the fact, that if a sphere multiset maps a state q to n, then the successors of the sphere can map q to 0 (when q is not a successor), $n-1$, n, or $n+1$. Therefore, for every symbol from Σ and every macrostate from at most $(\boldsymbol{max}_A + 1)^{|Q|}$ many of them, there are at most $4^{|Q|}$ successors, and $|\Sigma| \cdot (\boldsymbol{max}_A + 1)^{|Q|} \cdot 4^{|Q|} = (4 \cdot (\boldsymbol{max}_A + 1))^{|Q|}$. □

5 Experimental Evaluation

The main purpose of our experimentation was to compare the proposed approach with the naïve determinisation and confirm that our method produces significantly smaller automata and mitigates the risk of the state space explosion causing a complete failure of determinisation (and the implied impossibility to use the desired deterministic automaton for the intended application, such as pattern matching). To this end, we extended the Microsoft's Automata library [18] with a prototype support for CAs, implemented the algorithm from Section 4 (denoted **Counting** in the following), and compared it to the standard determinisation already present in the library (denoted as **DFA**). For the evaluation, we collected 2,361 regexes from a wide range of applications—namely, those used in network intrusion detection systems (Snort [17]: 741 regexes, Yang [29]: 228 regexes, Bro [21]: 417 regexes, HomeBrewed [28]: 55 regexes), the Microsoft's security leak scanning system (Industrial: 17 regexes), the Sagan log analysis engine (Sagan [26]: 14 regexes), and the pattern matching rules from RegExLib (RegExLib [20]: 889 regexes). We only selected regexes that contain an occurrence of the counting operator, and from these, we selected only monadic ones (there were over 95 % of them, confirming the fragment's importance). All benchmarks were run on a Xeon E5-2620v2@2.4GHz CPU with 32 GiB RAM with a timeout of 1 min (we take the mean time of 10 runs). In the following, we use μ, m, and σ to denote the statistical indicators mean, median, and standard deviation, respectively. All times are reported in milliseconds.

The number of timeouts was 110 for **Counting**, and 238 for **DFA**. The two methods were to some degree complementary, there were only 62 cases in which both timed out. This confirms that our algorithm indeed mitigates the risk of failure due to state space explosion in determinisation. The remaining comparisons are done only with respect to benchmarks for which neither of the methods timed out.

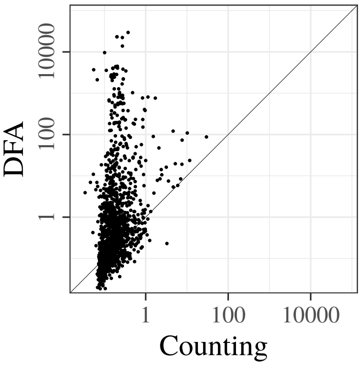

Fig. 3. Comparison of running times given in ms (the axes are logarithmic).

In Fig. 3, we compare the running times of the conversion of an NFA for a given regex to a DFA (the **DFA** axis) and the determinisation of the CA for the same regex (the **Counting** axis). If we exclude the easy cases where both approaches finished within 1 ms, we can see that **Counting** is almost always better than **DFA**. Note that the axes are logarithmic, so the advantage of **Counting** over **DFA** grows exponentially wrt the distance of the data point from the diagonal. The statistical indicators for the running times are $\mu = 110$, $m = 0.17$, $\sigma = 1{,}177$ for **DFA** and $\mu = 0.23$, $m = 0.13$, $\sigma = 0.09$ for **Counting**.

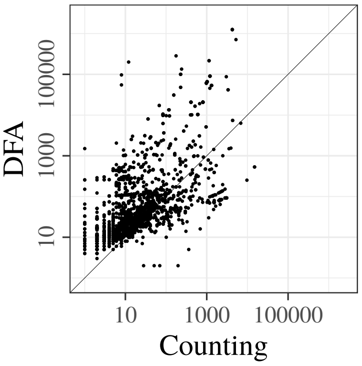

Fig. 4. Comparison of numbers of states (the axes are logarithmic).

In Fig. 4, we compare the number of states of the results of the determinisation algorithms (DCA for **Counting** and DFA for **DFA**). Also here, **Counting** significantly dominates **DFA**. The statistical indicators for the numbers of states are $\mu = 4{,}543$, $m = 41$, $\sigma = 57{,}543$ for **DFA** and $\mu = 241$, $m = 13$, $\sigma = 800$ for **Counting**. To better evaluate the conciseness of using DCAs, we further selected 184 benchmarks that suffered from state explosion during determinisation (our criterion for the selection was that the number of states increased at least ten-fold in **DFA**) and explored how the CA model can be used to mitigate the explosion. Fig. 5 shows histograms of how DCAs were more compact than DFAs and also how much the number of counters rose during the determinisation. From the histograms, we can see that there are indeed many cases where the use of DCAs allows one to use a significantly more compact representation, in some cases by the factor of hundreds, thousands, or even tens of thousands. Furthermore, the other histogram shows that, in many cases, no blow-up in the number of counters happened; though there are also cases where the number of counters increased by the factor of hundreds.

In terms of numbers of transitions, the methods compare similarly as for numbers of states, as shown in Fig. 6. We obtained $\mu = 14{,}282$, $m = 77$, $\sigma = 213{,}406$ for **DFA** and $\mu = 2{,}398$, $m = 23$, $\sigma = 8{,}475$ for **Counting**. (We emphasize the number of states over

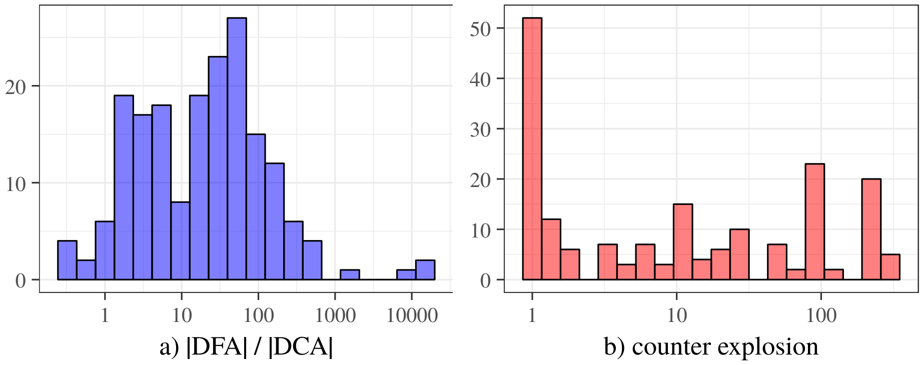

Fig. 5. Histograms of (a) the ratio of the number of states of a DFA and of the corresponding DCA (i.e., a bar at value x of a height h denotes that the size of the DCA was h times around x times smaller than the size of the corresponding DFA) and (b) the ratio of the number of counters used by a CA after and before determinisation. Note that the x-axes are logarithmic in both cases.

the number of transitions in our comparisons since the performance and complexity of automata algorithms is usually more sensitive to the number of states, and large numbers of transitions are amenable for efficient symbolic representations [12, 6, 16].)

Benefits of the **Counting** method were the most substantial on the Industrial dataset. For the regex `".*A[^AB]{0,800}C[D-G]{43,53}DFG[^D-H]"` (which was obtained from the real one, which is confidential, by substituting the used character classes by characters `A`–`H`), the obtained DFA contains 200,132 states, while the DCA contains only 12 states (and 2 counters), which is 16,667 times less. When minimised, the DFA still has 65,193 states. There were other regexes where **Counting** achieved a great reduction, in total two regexes had a reduction of over 10,000, three more regexes had a reduction of over 1,000, and 45 more had a reduction of over 100.

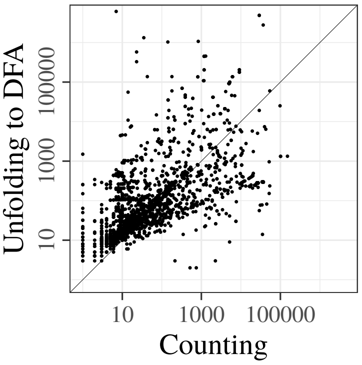

Fig. 6. Comparison of numbers of transitions (the axes are logarithmic).

Additionally, we also compared our approach against the naïve determinisation followed by the standard minimisation. Due to the space restrictions and since minimisation is not relevant to our primary target (preventing failure due to state space explosion during determinisation), we present the results only briefly. Minimisation increased the running times of **DFA** by about one half ($\mu = 150$, $m = 0.35$, $\sigma = 1,582$ for the running times of **DFA** followed by minimisation). The minimal DFAs were on average about ten times smaller than the original DFAs, and about ten times larger than our DCAs ($\mu = 385$, $m = 29$, $\sigma = 4,195$ for the numbers of states of the minimal DFAs).

6 Related Work

Our notion of CAs is close to the definition of FACs in [13], but our CAs are more general, by allowing input predicates and more complex counter updates. Also R-automata [1] are related but somewhat orthogonal to CAs because counters in R-automata do not need to have upper bounds and cannot be tested or compared. Counter systems are also related to CAs but allow more general operations over counters through Presburger formulas [2]. CAs can also be seen as a special case of extended finite state machines or EFSMs [5, 22, 24, 25], but these already go beyond regular languages.

Extended FAs (XFAs) augment classical automata with so-called scratch memory of bits and bit-instructions [23, 24], which can represent counters and also reduce nondeterminism. Regexes are compiled into deterministic XFAs by first using an extended version of Thompson's algorithm [27], then determinised through an extended version of the classical powerset construction, and finally minimised. Although a small XFA may exist, the determinisation algorithm incurs an intermediate exponential blowup of search space for inputs such as `.*a.{`k`}` (cf. [23, Section 6.2]), i.e., the regex from our running example, and handling of such cases remained an open problem.

Regular expressions with counters are also discussed in [13, 15, 8]. The automata with counters used in [13], called FACs, correspond closely, apart from our symbolic character predicates and transition representation, to the class of CAs considered in our work. A central result in [13] is that for *counter-1-unambiguous* regexes, the translation algorithm yields deterministic FACs and that checking determinism of FACs can be done in polynomial time. There are also works on regular expressions with counting that translate deterministic regexes to CAs and work with different notions of determinism [9, 4]. The related work in [14] studies membership in regexes with counting. None of these papers addresses the problem of determinising nondeterministic CAs.

7 Future Directions

Among future directions, we will consider optimisations of the current algorithm by means of avoiding construction of unreachable parts of DCAs or by finding efficient data structures, generalising the techniques used for monadic CAs to a larger class of CAs, and building a competitive pattern matching engine around the current algorithm. Since we believe that CAs have a lot of potential as a general succinct automata representation, we will work towards filling in efficient CA counterparts of standard automata algorithms, such as Boolean operations, minimisation, or emptiness test, that could also be used in other applications than pattern matching, such as verification and decision procedures of logics.

References

1. Abdulla, P.A., Krčál, P., Yi, W.: R-Automata. In: Proc. of CONCUR'08. LNCS, vol. 5201. Springer (2008)
2. Bardin, S., Finkel, A., Leroux, J., Petrucci, L.: FAST: Acceleration from Theory to Practice. STTT **10**(5) (2008)
3. Börklund, E., Martens, W., Timm, T.: Efficient Incremental Evaluation of Succinct Regular Expressions. In: Proc. of CIKM'15. ACM (2015)
4. Chen, H., Lu, P.: Checking Determinism of Regular Expressions with Counting. Information and Computation **241** (2015)
5. Cheng, K., Krishnakumar, A.S.: Automatic Functional Test Generation Using the Extended Finite State Machine Model. In: Proc. of DAC'93. ACM Press (1993)

6. D'Antoni, L., Veanes, M.: Minimization of Symbolic Automata. In: Proc. of POPL'14. ACM (2014)
7. Dill, D.L., Hu, A.J., Wong-Toi, H.: Checking for language inclusion using simulation preorders. In: Proc. of CAV'91. LNCS, vol. 575. Springer (1992)
8. Gelade, W., Martens, W., Neven, F.: Optimizing Schema Languages for XML: Numerical Constraints and Interleaving. In: Proc. of ICDT'07. LNCS, vol. 4353. Springer (2007)
9. Gelade, W., Gyssens, M., Martens, W.: Regular Expressions with Counting: Weak versus Strong Determinism. In: Proc. of MFCS'09. LNCS, vol. 5734. Springer (2009)
10. van Glabbeek, R., Ploeger, B.: Five Determinisation Algorithms. In: Proc. of CIAA'08. LNCS, vol. 5148. Springer (2008)
11. Heizmann, M., Hoenicke, J., Podelski, A.: Software Model Checking for People Who Love Automata. In: Proc. of CAV'13. LNCS, vol. 8044. Springer (2013)
12. Henriksen, J., Jensen, J., Jørgensen, M., Klarlund, N., Paige, B., Rauhe, T., Sandholm, A.: Mona: Monadic second-order logic in practice. In: TACAS'95. LNCS, vol. 1019. Springer (1995)
13. Hovland, D.: Regular Expressions with Numerical Constraints and Automata with Counters. In: Proc. of ICTAC'09. LNCS, vol. 5684. Springer (2009)
14. Hovland, D.: The Membership Problem for Regular Expressions with Unordered Concatenation and Numerical Constraints. In: Proc. of LATA'12. LNCS, vol. 7183. Springer (2012)
15. Kilpeläinen, P., Tuhkanen, R.: One-unambiguity of Regular Expressions with Numeric Occurrence Indicators. Information and Computation **205**(6) (2007)
16. Lengál, O., Simácek, J., Vojnar, T.: VATA: A library for efficient manipulation of non-deterministic tree automata. In: TACAS'12. LNCS, vol. 7214. Springer (2012)
17. M. Roesch et al.: Snort: A Network Intrusion Detection and Prevention System, `http://www.snort.org`
18. Microsoft Automata library: Automata and Transducer Library for .NET, `https://github.com/AutomataDotNet/Automata`
19. OWASP Foundation and Checkmarx: Regular Expression Denial of Service: ReDoS (2017)
20. RegExLib.com: The Internet's First Regular Expression Library, `http://regexlib.com/`
21. Robin Sommer et al.: The Bro Network Security Monitor, `http://www.bro.org`
22. Shiple, T.R., Kukula, J.H., Ranjan, R.K.: A Comparison of Presburger Engines for EFSM Reachability. In: Proc. of CAV'98. LNCS, vol. 1427. Springer (1998)
23. Smith, R., Estan, C., Jha, S.: XFA: Faster Signature Matching with Extended Automata. In: Proc. of SSP'08. IEEE (2008)
24. Smith, R., Estan, C., Jha, S., Siahaan, I.: Fast Signature Matching Using Extended Finite Automaton (XFA). In: Proc. of ICISS'08. LNCS, vol. 5352. Springer (2008)
25. Sperberg-McQueen, M.: Notes on Finite State Automata with Counters, `https://www.w3.org/XML/2004/05/msm-cfa.html`, accessed: 2018-08-08
26. The Sagan team: The Sagan Log Analysis Engine, `https://quadrantsec.com/sagan_log_analysis_engine/`
27. Thompson, K.: Programming Techniques: Regular Expression Search Algorithm. Communications of the ACM **11**(6) (1968)
28. Češka, M., Havlena, V., Holík, L., Lengál, O., Vojnar, T.: Approximate Reduction of Finite Automata for High-Speed Network Intrusion Detection. In: TACAS'18. LNCS, vol. 10806. Springer (2018)
29. Yang, L., Karim, R., Ganapathy, V., Smith, R.: Improving NFA-Based Signature Matching Using Ordered Binary Decision Diagrams. In: RAID'10. LNCS, vol. 6307. Springer (2010)