BRNO UNIVERSITY OF TECHNOLOGY FACULTY OF INFORMATION TECHNOLOGY

Excel@FIT 2020
http://excel.fit.vutbr.cz

Inclusion of Regular Expressions with Counting

David Mikšaník*

Abstract
We present an algorithm solving the inclusion problem for regular expressions with the counting operator limited to character classes, the so-called extended regular expressions (eREs), which are common in practice. Such regular expressions do not extend expressiveness beyond regularity, but allow one to succinctly express repeated patterns. Our algorithm is based on the transformation eREs into monadic counting automata (MCAs), i.e., finite automata with counting loops on character class where each counter is bounded. Similarly to the classical algorithm, we transform eREs into automata, but now we use MCAs instead of nondeterministic finite automata (NFAs). Following by building the product of MCAs and searching for a final state in the product. MCAs are compact representation of eREs because the number of states in MCAs does not depend on the bounds used in the counting operator, in contrast to NFAs where the number of states grows linearly. These bounds can be large in practice, thus MCAs are often significantly smaller than NFAs. We provide several examples for which the classical algorithm working with NFAs does not terminate in a reasonable amount of time, but our algorithm does. We also hope that our algorithm outperforms the classical algorithm in general, especially if the bounds of the counting operators are large.

Keywords: regular expressions — language inclusion — finite automata — counting automata

Supplementary Material: *N/A*

*xmiksa05@stud.fit.vutbr.cz, *Faculty of Information Technology, Brno University of Technology*

1. Introduction

Regular expressions with the counting operator limited to character classes, we called them *extended regular expressions* (eREs), have the same expressive power as the standard regular expressions (REs). The purpose of eREs is a succinct expression of some regular expressions such as `[abc]{5,10}` representing all strings of the length 5–10 where each symbol of the string is `a`, `b`, or `c`. Usually the eREs also contain other well-known operators, e.g., `+` or `?`, but such operators do not bring the succinctness as the counting operator. Sometimes the restriction of appearing counting operator only in the character classes is omitted (e.g., `(abc){5,10}` denoting all strings where `abc` appears 5–10 times). In [1], it was observed that in practice the regular expressions mostly use only counting operator limited to character classes (e.g., in the Snort rules [2] used for finding attacks in network traffic; or in the RegExLib library [3], which collects expressions for recognizing URIs, markup code to name a few). For these reasons, we limit ourselves only to eREs with counting operator limited to character classes.

Although there are several automata models that are capable of representing regular expressions in as succinct way as the eREs (e.g., [1, 4, 5]), for the best of our knowledge all algorithms solving the inclusion problem for eREs work with NFAs—eREs are transforming to NFAs directly or from any of the model above. In each case the advantage of the compact representation is lost. In this work we develop an algorithm that avoids such transformation to NFAs.

The inclusion problem for REs consists of two inputs REs r_1, r_2 and the question whether the language that r_1 denotes is included in the language that r_2 denotes, symbolically we need to decide whether $\mathcal{L}(r_1) \subseteq \mathcal{L}(r_2)$. The language of regular expressions is defined as usual. The inclusion problem for eREs is the same as the problem above except that r_1, r_2 can contain the counting operator limited to character classes. This problem can arise in any Intrusion Detection System (IDS). Suppose the following scenario. Let α be eRE that representing all packets in network traffic that are possible dangerous (or interesting). But you want to detect more packets that are dangerous. Thus you construct an eRE β. Because you want to be sure that β is as strong as α, you need test whether $\mathcal{L}(\alpha) \subseteq \mathcal{L}(\beta)$. If it is true, then you know that the replacement α by β is safe.

General solutions to the inclusion problem for eREs start with constructing NFAs N_1, N_2 for each of the input eRE. Then is used the algorithm for testing language inclusion for the NFAs. How was proposed in [6], these algorithms can be categorized into two types: (1) those based on *subset construction* build a product automaton $N_1 \times \overline{N_2}$ of N_1 and the complement of N_2 and search for a final state; (2) those based on *simulation* (e.g., [7]) first compute a simulation relation on $N_1 \cup N_2$ and then check if all initial states of N_1 can be simulated by some initial state of N_2. Usually (2) is better than (1) because the computation of simulation is done in polynomial time. But the main drawback of (2) is that it is incomplete—simulation implies language inclusion, but not vice-versa. Contrary, the methods based on the subset construction are complete, but the complementation of N_2 requires that N_2 is deterministic, which can easily exponentially blow up in the number of states.

As was mentioned above, there are several automata models for succinct representation of eREs. From these models we choose one, called *monadic counting automata* (MCAs), which are introduced in [1, section 4.1]. Informally, they are finite automata with bounded counters. Our algorithm for the inclusion problem for eREs works similar as the methods based on the subset construction. First, we transform the input eREs into MCAs M_1, M_2. Second, we create $\overline{M_2}$, the complement of M_2, by determinizing and completing M_2 and then complementing the accepting condition. Third, we build the product automaton $M_1 \times \overline{M_2}$ and search for a final state. We note that [1] provide the algorithm for determinization of MCAs. The problem is how to efficiently test whether a state is reachable from the initial states. Such a test is not as easy as for NFAs, because the next move of the MCA does not depend only on the input symbol, but also on the values of counters. The main contribution of this paper is the algorithm for testing reachability of states in the product automaton (see Section 3).

The advantages of our algorithm are showing when the eREs contain a lot of counting operators and the upper bound of the counting operator is large. For example, if $r_1 = .^*\mathtt{a}.\{\mathtt{k}\}, r_2 = .^*\mathtt{a}.\{\mathtt{k-1,k+1}\}$ the methods based on NFAs do not terminate in a reasonable time for relatively small values of k, but if we use MCAs, then the problem is solvable for such values of k. Eventually, our algorithm converges to the methods based on the subset construction if the eREs do not contain any counting operator. In this case, it is appropriate to use some method based on simulation relation.

2. Counting Automata

A finite, non-empty set Σ of *symbols* is called an *alphabet*. A *string* is a sequence of symbols $a_1 a_2 \ldots a_n$ where $a_i \in \Sigma$, for $1 \leq i \leq n$. The *length* of w is defined as $|w| = n$. We use $\varepsilon \notin \Sigma$ to denote the *empty string*, so $|\varepsilon| = 0$. The set of all strings over the alphabet Σ is denoted by Σ^*.

The abstract syntax of the extended regular expressions (eREs) is the following:

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

where σ is a predicate denoting a set of alphabet symbols, and $n, m \in \mathbb{N}$. The semantics is defined as in the standard regular expressions, with $\sigma\{n,m\}$ denoting a string w with $n \leq |w| \leq m$, where $n \leq m$, symbols each of them satisfying σ.

Monadic counting automata (MCAs) naturally arise from eREs (see Figure 1). They are a restriction of a more general concept called counting automata (CAs). In this section, we provide necessary theory background and give the definitions of CAs and MCAs, following [1].

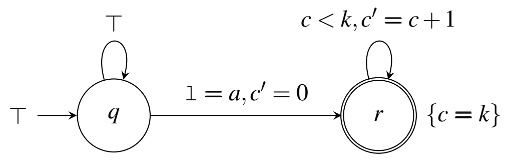

Figure 1. A CA for the eRE $.^*\mathtt{a}.\{\mathtt{k}\}$, which is also an MCA with $k \in \mathbb{N}$, $I : \mathtt{s} = q$, $F : \mathtt{s} = r \wedge c = k$, and $\Delta : q \text{-}\{\top,\top,\top\}\text{→} q \vee q \text{-}\{\mathtt{l}=a,\top,c'=0\}\text{→} r \vee r \text{-}\{\top, c<k, c'=c+1\}\text{→} r$.

2.1 Logical background

Let V be a set of variables V, and let Q be a set of constants (disjoint with the set of natural numbers $\mathbb{N}$ including zero), which will correspond to the set of states in CAs. We define a *Q-formula over V* to be a quantifier-free formula φ of Presburger arithmetic [8, Section 3.3] 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 $sat(\varphi)$ to denote that φ has a model and we say that φ is *satisfiable*. The *semantics* of a formula φ is the set $[\![\varphi]\!]$ of all possible tuples of the free variables in φ which make φ true. Suppose that $C = \{c_1, \dots, c_n\}$ are the free variables in φ. The *projection* of φ on C is the formula $\exists c_1, \dots, c_n : \varphi$.

2.2 Labelled Transitions Systems

A *labelled transition system* (LTS) over Σ is a five-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 symbol variable and allow it as the only term that can occur with a predicate $\mathtt{l} = a$ for $a \in \Sigma$, called an *atomic symbol guard*. Moreover, $\mathtt{l}$ is also not allowed to occur in any other predicates in Δ.

A *configuration* of LTS T is a function $\alpha : V \to \mathbb{N} \cup Q$ that maps every configuration variable to a number from $\mathbb{N}$ or a state from Q. We will denote by $\mathfrak{C}$ the set of all configuration of the LTS T. The transition relation $[\![\Delta]\!] \subseteq \mathfrak{C} \times \Sigma \times \mathfrak{C}$ is encoded by the transition formula Δ as follows $(\alpha, a, \alpha') \in [\![\Delta]\!]$ iff $\alpha \cup \{x' \mapsto k \mid \alpha'(x) = k\} \cup \{\mathtt{l} \mapsto a\} \models \Delta$. For a string $w \in \Sigma^*$, we define inductively that a configuration α' is a *w-successor* of α, written $\alpha \xrightarrow{w} \alpha'$, such that $\alpha \xrightarrow{\varepsilon} \alpha$ for all $\alpha \in \mathfrak{C}$, and $\alpha \xrightarrow{av} \alpha'$ iff $\alpha \xrightarrow{a} \overline{\alpha} \xrightarrow{v} \alpha'$ for some configuration $\overline{\alpha}$, $a \in \Sigma$, and $v \in \Sigma^*$. A configuration α is initial if $\alpha \models I$, and final if $\alpha \models F$. 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.

2.3 Counting Automata

A *(nondeterministic) counting automaton* (CA) is a five-tuple $N = (Q, C, I, F, \Delta)$ 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 $\mathtt{s}$ such that $\mathtt{s} \notin C$.
2. The transition formula Δ is a disjunction of *transitions*, which are conjunctions of the form $(\mathtt{s} = q) \wedge \sigma \wedge g \wedge f \wedge (\mathtt{s}' = r)$, denoted by $q \xrightarrow{\sigma, g, f} r$, where $q, r \in Q$, σ is the transition's *guard formula* over $\{\mathtt{l}\}$, g is the transition's *guard formula* over V, 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 $\boldsymbol{max}_N \in \mathbb{N}$ such that no counter can ever grow above that value.

Moreover, for every transition $\varphi = q \xrightarrow{\sigma, g, f} r \in \Delta$, we define the function *sym* that returns the transition's guard formula over $\{\mathtt{l}\}$, that is $sym(\varphi) := \sigma$.

A *deterministic counting automaton* (DCA) is a CA N where I has at most one model and, for every $a \in \Sigma$, every reachable configuration α has at most one a-successor. A CA is *complete* if for any configuration $\alpha \in \mathfrak{C}$ and every symbol $a \in \Sigma$ the a-successor $\alpha' \in \mathfrak{C}$.

2.4 Monadic Counting Automata

A (nondeterministic) *monadic counting automaton* (MCA) is a CA $M = (Q, C, I, F, \Delta)$ where the following holds:

1. The set of control states is $Q = Q_s \uplus Q_c$, where Q_s is a set of *simple states* and Q_c is a set of *counting states*.
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 $q \xrightarrow{\sigma, c_q < \boldsymbol{max}_q, c'_q = c_q + 1} q$ with the value of c_q limited by the bound $\boldsymbol{max}_q$ of q, and possibly several *entry transitions* of the form $r \xrightarrow{\sigma, \top, c'_q = 0} q$, which set c_q to 0. As for *exit transitions*, every counting states is either *exact* or *range* where exact counting states have exit transitions of the form $q \xrightarrow{\sigma, c_q = \boldsymbol{max}_q, \top} s$ and range counting state have exit transitions of the from $q \xrightarrow{\sigma, \top, \top} s$ with $s \in Q$ such that $s \neq q$.
4. The initial condition I is of the form

$$I : \bigvee_{q \in Q^I_s} \mathtt{s} = q \vee \bigvee_{q \in Q^I_c} (\mathtt{s} = q \wedge c_q = 0)$$

   for some sets of initial simple and counting states $Q^I_s \subseteq Q_s$ and $Q^I_c \subseteq Q_c$, respectively.
5. The final condition F is of the form

$$F : \bigvee_{q \in Q^F_s \cup Q^F_r} \mathtt{s} = q \vee \bigvee_{q \in Q^F_e} (\mathtt{s} = q \wedge c_q = max_q)$$

   where $Q^F_s \subseteq Q_s$ is a set of simple final states, $Q^F_r \subseteq Q_r$ is a set o final range counting states, and $Q^F_e \subseteq Q_e$ is a set of final exact counting states.

3. The Inclusion Problem for Extended Regular Expressions

Let r_1, r_2 be the input eREs of the inclusion problem. Our solution starts with creating MCAs M_1, M_2 such that $\mathcal{L}(r_1) = \mathcal{L}(M_1)$ and $\mathcal{L}(r_2) = \mathcal{L}(M_2)$. We need to decide whether $\mathcal{L}(M_1) \subseteq \mathcal{L}(M_2)$, or equivalently $\mathcal{L}(M_1) \cap \overline{\mathcal{L}(M_2)} = \mathcal{L}(M_1) \cap \mathcal{L}(\overline{M_2}) = \emptyset$. For convenience, the automaton that recognize such a language is called the *product automaton*, written $M_1 \times \overline{M_2}$.

Traditionally, to complement M_2 we need first determinize M_2. Since the determinization of an MCA is not again an MCA, in Section 3.1 following [1], we give a brief overview how the states in such automaton are represented. In Sections 3.2–3.4, we present our algorithm for the inclusion problem for eREs, namely how to build $M_1 \times \overline{M_2}$ and how to test whether $\mathcal{L}(M_1) \cap \mathcal{L}(\overline{M_2}) = \emptyset$.

3.1 Determinization of Monadic Counting Automata

The crucial step in building the product automaton is the determinization of M_2. Fortunately, [1, Section 4.2] provides the algorithm for determinization of MCAs. We note that the result automaton of the algorithm is not again an MCA, but the automaton has still somewhat restricted structure as we show in Section 3.4. For that reason, all MCAs that are determinized by the algorithm in [1, Section 4.2] are called *determinized* MCAs (DMCAs).

Each state of a DMCA is represented by the notion of *sphere* [1]:

$$\Psi := \bigvee_{q \in Q'_s} \mathtt{s} = q \vee \bigvee_{q \in Q'_c} \Big(\mathtt{s} = q \wedge \bigvee_{0 \leq i \leq \boldsymbol{max}'_q} c_q = c_q[i]\Big) \tag{1}$$

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 determinized MCA and also which variants of the counter c_q may record the value of c_q when q is reached.

The spheres can be also represented by 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., $\mathtt{s} = q$ is a predicate in (1), and for a counting state q, the counters $c_q[0], \ldots, c_q[\Psi(q) - 1]$ are the $\Psi(q)$ variants c_q tracked in the sphere. i.e., $\boldsymbol{max}'_q = \Psi(q)$ in (1).

3.2 Product Construction of Monadic Counting Automata

First, we need to compute $\overline{M_2}$, the complement of M_2. Since we know how to determinize MCAs, it remains to complete the determinization of MCA. Let $D = (Q^D, C^D, I^D, F^D, \Delta^D)$ be a DMCA with the same language as M_2. The completion of D is the following: we add a new non-final state q_{sink} and the transition $q_{sink}\text{-}\{\top,\top,\top\}\!\rightarrow q_{sink}$. For every state q, let $P_q = \{\varphi \mid q\text{-}\{\varphi\}\!\rightarrow r \in \Delta^D\}$. Then, for every state $q \neq q_{sink}$ we add a new transition $q\text{-}\{\psi\}\!\rightarrow q_{sink}$ where $\psi = \wedge_{\varphi \in P} \neg\varphi$. Intuitively, if no outgoing transition from q can be executed, then we can use this new added one. For this reason the procedure preserves the determinism. Also the langauge of D is preserved because q_{sink} is not a final state. To finish to complementation of M_2 it is sufficient to complement the final condition F^D of D by simply putting $\neg F^D$. We define the function *complement* that takes an MCA and produces a DMCA as described above.

Second, we build the product automaton $N = M_1 \times \overline{M_2}$ that recognizes the language $\mathcal{L}(M_1) \cap \overline{\mathcal{L}(M_2)}$ similarly as for NFAs. The states of $N = (Q, C, I, F, \Delta)$ are pairs (q, R), where $q \in Q_1$ and $R \in Q^D$, i.e., $Q \subseteq Q_1 \times Q^D$. The set of counters of N is $C \subseteq C_1 \cup C^D$ (some counter might not be needed if its corresponding state is not reachable, see below the function *ground*). The initial formula I of N labels pairs of states as initial if both states are also initial in M_1 and $\overline{M_2}$, respectively. Formally, we transform $I = I_1 \wedge I^D$ into disjunctive normal formal such that each part of disjuncts of the form $\mathtt{s} = q \wedge \mathtt{s} = R$ is replaced by $\mathtt{s} = (q, R)$. The initial value of counters are then the combinations of initial values of M_1 and $\overline{M_2}$. This transformation is denoted by dnf, so $I = dnf(I_1 \wedge I^D)$. The final formula F of N is computed analogously. The transition formula Δ of N is computed as follows: let (q, R) be a reachable state (at the start all initial states are reachable). We combine the outgoing transitions from q and R, that is if $\varphi = q\text{-}\{\omega\}\!\rightarrow q' \in \Delta_1$, $\psi = R\text{-}\{\Omega\}\!\rightarrow R' \in \Delta^D$ and $sym(\varphi) \wedge sym(\psi)$ is satisfiable, then we add a new transition $(q, R)\text{-}\{\omega \wedge \Omega\}\!\rightarrow (q', R')$ to Δ and the pair (q', R') is marked as reachable. After we process all combinations of outgoing transitions, we pick other reachable state, but each state is process at most once, thus the algorithm terminates. We note that the generated states might not be reachable in real (see Section 3.3), because we do not only combine the transitions $\varphi \in \Delta_1$ and $\psi \in \Delta^D$ for which $sym(\varphi) \wedge sym(\psi)$ is unsatisfiable and completely ignore the guards on the counters, which can cause that the transitions are unsatisfiable—this is purpose of the next section.

The whole procedure of the product construction is summarized in Algorithm 1, where the function *ground* on Line 14 removes from $C_1 \cup \neg C^D$ all counters that do not appear in any guards of I and Δ and the function *ground* on Line 15 removes predicates from $dnf(F_1 \wedge \neg F^D)$ that contain states that are not in Q or counters that are not in C.

```
Algorithm 1: Product of MCAs
   Input  : MCAs $M_1 = (Q_1, C_1, I_1, F_1, \Delta_1)$,
            $M_2 = (Q_2, C_2, I_2, F_2, \Delta_2)$ with
            $Q_1 \cap Q_2 = C_1 \cap C_2 = \emptyset$
   Output : An CA $N = M_1 \times \overline{M_2}$ such that
            $\mathcal{L}(N) = \mathcal{L}(M_1) \cap \overline{\mathcal{L}(M_2)}$
1  $(Q^D, C^D, I^D, \neg F^D, \Delta^D) \leftarrow complement(M_2)$;
2  $I \leftarrow dnf(I_1 \wedge I^D)$; $\Delta \leftarrow \emptyset$;
3  $Q \leftarrow \{(q,R) \mid s = (q,R)$ appears in $I\}$;
4  $W \leftarrow Q$;
5  while $W \neq \emptyset$ do
6      pick and remove $(q,R)$ from $W$;
7      foreach $q \text{-}\{\varphi\}\text{-}> q' \in \Delta_1$, $R \text{-}\{\psi\}\text{-}> R' \in \Delta^D$ do
8          Let $\sigma_1 = sym(q \text{-}\{\varphi\}\text{-}> q')$;
9          Let $\sigma_2 = sym(R \text{-}\{\psi\}\text{-}> R')$;
10         if $sat(\sigma_1 \wedge \sigma_2)$ then
11             if $(q', R') \notin Q$ then
12                 $Q \leftarrow Q \cup \{(q', R')\}$;
13                 $W \leftarrow W \cup \{(q', R')\}$;
14             $\Delta \leftarrow \Delta \cup \{(q,R) \text{-}\{\varphi \wedge \psi\}\text{-}> (q', R')\}$;
15 $C \leftarrow ground(C_1 \cup C^D)$;
16 $F \leftarrow ground(dnf(F_1 \wedge \neg F^D))$;
17 return $(Q, C, I, F, \Delta)$;
```

3.3 Reachability of Final States

In principle, we apply breath-first search on the product automaton $M_1 \times \overline{M_2}$ where the starting points are the initial states. If we encounter a final state, then we need to check whether its final condition is reachable. If so, we stop and know that $\mathcal{L}(M_1) \cap \overline{\mathcal{L}(M_2)} \neq \emptyset$, otherwise we continue in searching.

In detail, for every state q in $M_1 \times \overline{M_2}$ the formula β_q denotes the possible values of counters if q is reached. We start from the initial states where the initial values of counters are given by the initial condition. These initial states are pushed in the list *Worklist*. For each state q that is not initial, we set β_q to $\perp$. Until *Worklist* is empty, we take a state q from *Worklist*. If q has self-loops, i.e., $q \text{-}\{\alpha\}\text{-}> q \in \Delta$ for some α, then the possible values of counters represented by β_q can be changed by executing the self-loops (the details are provided in Section 3.4, see function *accelerate*), thus suppose that β_q is updated accordingly to the self-loops of q. Then for all states $r \neq q$ such that $q \text{-}\{\varphi\}\text{-}> r \in \Delta$, we test whether $\beta_q \wedge \varphi$ is satisfiable. If so, then we compute new values of counters of the states r. This is done by the projection $\beta_q \wedge \varphi$ of the unprimed counters used in the formula, followed by an application of the unprime function. The *unprime* function replaces every occurrence of a primed counter c' by its corresponding unprimed counter c. Let $\psi = unprime(projection(\beta_q \wedge \varphi))$. If $[\![\psi]\!] \subseteq [\![\beta_r]\!]$, then we do not add the state q to *Worklist* because we do not get any new information—if some transition from r is unsatisfiable, then it will still be unsatisfiable if β_r is modified by the new values represented by ψ. Otherwise $\beta_r := \beta_r \vee \psi$ and we push r in *Worklist*. To test whether a state q is final, we test whether $\beta_q \wedge F$ is satisfiable.

We note that some states q can appear in *Worklist* more than once. But always the semantics of the formula β_q got larger if the state appears again in the *Worklist*. Since the number of possible configurations of the product automaton, i.e., the number of possible values of counters, is finite (due to bounded counters), the state q cannot be added to *Worklist* infinite number of times. Therefore, the algorithm always terminates.

3.4 Acceleration of Self-loops

We have built the product automaton $N = M_1 \times \overline{M_2}$ in Section 3.2. We also provide the test of reachability of states in the product automaton in Section 3.3. However, we do not say how the formula β_q is updated if a state q has a self-loop, which we fix in this section.

Let (q,R) be a state of N. The state (q,R) has a self-loop in N if and only if q has a self-loop in M_1 and R has a self-loop in $\overline{M_2}$. Since M_1 is an MCA, we know that each state of M_1 has at most one self-loop by the definition. This is not true in general in the DMCA $\overline{M_2}$. It follows that the state (q,R) in the product automaton can have more than one self-loop.

The trivial solution to update β_q is that we do not distinguish between the self-loops and the outgoing transitions of the state q in Section 3.3. If such a solution is used, then the algorithm for the inclusion problem for eREs runs as slowly as the methods based on NFAs. First, we give a small example showing why

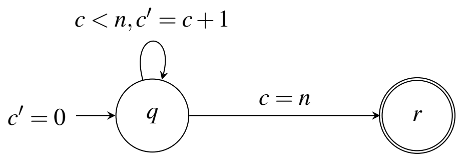

Figure 2. Example of an MCA M with $n \in \mathbb{N}$ where q is an exact counting state. The language of M consists of all strings of length n.

Table 1. Possible forms of the self-loops on state q in a DMCA.

Name	Form of the label of a self-loop in DMCAs
(I)	$c_q[0] < \boldsymbol{max}_q \wedge c_q[0]' = 0$
(II)	$c_q[\Psi(q)-1] \neq \boldsymbol{max}_q \wedge \bigwedge_{i=0}^{\Psi(q)-1} c_q[i]' = c_q[i]+1$
(III)	$c_q[\Psi(q)-1] = \boldsymbol{max}_q \wedge c_q[0]' = 0 \wedge \bigwedge_{i=0}^{\Psi(q)-2} c_q[i+1]' = c_q[i]+1$

this solution is strongly inefficient, and then we present an idea of how the self-loops can be accelerated.

Suppose that we have an MCA M_1 as in Figure 2 and assume that M_2 is a one-state MCA such that $\mathcal{L}(M_2) = \Sigma^*$. Then the product automaton $M_1 \times \overline{M_2}$ has the same structure as the MCA M_1. We simulate the procedure from Section 3.3 with the trivial solution for updating β_q. The initial formula gives $\beta_q = (c = 0)$. Next, we check whether the self-loop or the exit transition of q can be executed, in this case only the self-loop on q can be executed. Note that $unprime(projection(\beta_q \wedge c < n \wedge c' = c+1))$ is equal to $c = 1$. Because $[\![c = 1]\!] \not\subseteq [\![c = 0]\!]$, we set $\beta_q = (c = 0 \vee c = 1)$. And q is pushed to *Worklist*. We continue in a similar manner and after n steps we obtain $\beta_q = (c = 0 \vee \cdots \vee c = n)$. The next step $n+1$ produces the formula $unprime(projection(\beta_q \wedge c < n \wedge c' = c+1))$ whose semantics is already included in $[\![\beta_q]\!]$, thus we stop executing this self-loop.

From this simple example, we see that the algorithm based on the trivial solution spends the most time in updating β_q of states q with self-loops (in practice n can be large). In the example above, we can use the advantage of that the product automaton is an MCA. We use the following facts: (1) if we enter a counting state q, then c_q is set to zero; (2) if q is a counting state, then there is only one possible self-loop, called incremental, which have a fixed form. In our example, the self-loop can be accelerated by replacing the guard and update of the self-loop by $\exists k : (0 \leq k \leq n \wedge c' = c + k \wedge c' \leq n)$. Such a formula is called an *acceleration* of the self-loop. By using the same methods as above, we can obtain the same result after one iteration instead of n. In general, for any counting state p, the self-loop of p can be replaced by the acceleration formula $\exists k : (0 \leq k \leq \boldsymbol{max}_q \wedge c'_q = c_q + k \wedge c'_q \leq \boldsymbol{max}_q)$.

Now, suppose that M_1 is a one-state such that $\mathcal{L}(M_1) = \Sigma^*$.[1] Then the product automaton $M_1 \times \overline{M_2}$ is a DMCA. As we said, the structure of DMCAs is still somewhat restricted, but more complicated than the structure of MCAs. There are two main differences: (1) each state can have several self-loops; (2) the counter guards and updates of a transition consists of several different variants of counters. Exactly as for MCAs, the key for acceleration of the self-loops in DMCAs is to find the forms of the self-loops in DMCAs. Suppose that the DMCA is the determinization of the MCA, which using only one counter. Then from the determinization algorithm [1, Section 4.2] follows that there are three different forms of self-loops (see Table 1).

We assume that $\boldsymbol{max}_q > 0$, otherwise the state is not counting. Thus the self-loops of the form (I) do not have to be accelerated, i.e., the acceleration formula looks the same as the form. The self-loops of the form (II) have a similar form as incremental self-loops in MCAs, except that we need to update more variants of counters. Since the highest variant of c_q has the highest value, the acceleration formula is $\exists k : \big(0 \leq k \leq \boldsymbol{max}_q \wedge \bigwedge_{i=0}^{\Psi(q)-1} c_q[i]' = c_q[i] + k \wedge c_q[i]' \leq \boldsymbol{max}_q\big)$. The self-loops of the form (III) do not have to be accelerated if $\Psi(q) - 1 = \boldsymbol{max}_q$. Otherwise we must have $\Psi(q) < \boldsymbol{max}_q$. In this case there is some largest number $k < \Psi(q) - 1$ such that $c_q[k] + 1 < c_q[k+1]$, or, equivalently. $c_q[\Psi(q) - 1 - k] + k \neq \boldsymbol{max}_q$. If there is no such k, then $c_q[0] > 0$. It follows that after $\Psi(q) - k$ iterations the self-loops do not add any new information. Thus the acceleration formula must need find to such a number k. So the acceleration formula is $\exists k : \big(0 \leq k \leq \boldsymbol{max}_q \wedge c_q[\Psi(q) - 1 - k] + k = \boldsymbol{max}_q \wedge \bigwedge_{i=0}^{k} c_q[i]' = i \wedge \bigwedge_{i=k+1}^{\Psi(q)-2} c_q[i]' = c_q[i-k] + k\big)$.

Suppose that we have a general product automaton. From the preceding paragraphs, we know how to accelerate a self-loop with counter variants of a single counter. But what would happen if the self-loop has counter variants of different counters? The counter guards and updates of the self-loops can be divided into $\varphi = \alpha_1 \wedge \cdots \wedge \alpha_n$ such that α_i contains only variants of single counter c_i. Each of the individual α_i is of the form (I)–(III), or incremental self-loop. For each of the forms we know its corresponding acceleration formula ψ_i, but the acceleration of φ is not equal to the conjunction of its corresponding acceleration formulae φ_i. We need to ensure that the k in each part of the counter update is the same. Replace the variable k in the counter guard of ψ by k_i. Now the acceleration formula of φ is equal to the conjunction of $\bigwedge_{i=0}^{n} \psi_i$ and $k_1 = k_2 = \cdots = k_n$, which ensures that the counters are

[1] Our algorithm can be also used for the *universality* problem of eRE. That is, we check whether $\Sigma^* \subseteq \mathcal{L}(M_2)$.

incremented by the same value.

The state q in the product automaton can have more than one self-loop. In this case we proceed as follows. Let $\varphi_1, \dots, \varphi_n$ be self-loops of q in an arbitrary, but fixed, order. For each self-loop φ_i we know the acceleration formula ψ_i. Thus we also know how β_q is updated. If $[\![\psi_i]\!] \subseteq [\![\beta_q]\!]$, then β_q is not updated. The function *accelerate* works as follows: we update β_q by processing the self-loops $\varphi_1, \dots, \varphi_n$ repeatedly until last n acceleration formulae do not update β_q.

Finally, we note that it is not necessary to build the whole product automaton and after that search for a reachable final state (with a reachable final condition). In practice, we can build the product automaton on the fly and if we encounter a reachable final state (with a reachable final condition), then we can stop. In this step, we have $\mathcal{L}(M_1) \cap \overline{\mathcal{L}(M_2)} \neq \emptyset$, or equivalently $\mathcal{L}(M_1) \not\subseteq \overline{\mathcal{L}(M_2)}$. Eventually, we stop if we build the whole product automaton.

4. Advantage of our algorithm over the methods based on NFAs

In this section, we give a comparison between our algorithm, which is introduced in Section 3, and the methods based on NFAs. We define the *size* of the automaton (classical or counting), written $|N|$, as the number of its states.

We have given eREs r_1 and r_2. Both approaches build the product automaton $N_1 \times \overline{N_2}$ where N_1, N_2 are NFAs or MCAs such that $\mathcal{L}(r_1) = \mathcal{L}(N_1)$ and $\mathcal{L}(r_2) = \mathcal{L}(N_2)$. In the following, N_i denotes an NFA and M_i denotes an MCA for $i \in \{1, 2\}$.

The size of the product automata is $\mathcal{O}(|N_1| \cdot |\overline{N_2}|)$ and $\mathcal{O}(|M_1| \cdot |\overline{M_2}|)$, respectively. Suppose that r_1 is a fixed eRE except that the bounds of the counting operator are parameters. If the bounds are increased, the size of M_1 does not change in contrast to the size of N_1 where the size grows linearly. Although $\mathcal{O}(|N_1|) = \mathcal{O}(|M_1|)$, for any value of the bounds in the counting operators we have $|M_1| \leq |N_1|$. The hidden constant in $\mathcal{O}$ may be large. Suppose that in r_1 the counting operator occurs $k > 0$ times and the smallest value of all lower bounds of the counting operators is $\ell > 1$. Because the size of N_1 grows linearly we know that $|M_1| \cdot k\ell \leq |N_1|$.

From [1] we know that the complemenation of MCAs may not have be always smaller than th unfolding MCAs to NFAs and complementing NFAs. But in many cases we have $|\overline{M_2}| \leq |\overline{N_2}|$ (cf. [1]). Therefore in many cases we have $|M_1| \cdot |\overline{M_2}| \cdot k\ell \leq |N_1| \cdot |\overline{N_2}|$. Thus the advantage of our approach is that we work with smaller automata. An open question is whether the reachability test will be efficient enough so that we outperform the methods based on NFAs.

At least one example was given in the introduction. Recall, let $r_1 =$ `.*a.{k}`, $r_2 =$ `.*a.{k-1,k+1}` for $k \geq 1$. N_1 representing r_1 has $k+1$ states, M_1 has always two states regardless of the values of k (see Figure 1). N_2 representing r_2 has $k+2$ states, the complement of N_2 has $2^{k+2}+1$ states, M_2 has always two states, the complement of M_2 has $(k+2)+1$ states. Note that we need to count the sink state of the complements. From the example we see that even for relatively small values of k the methods based on NFAs fail.

5. Conclusions

In this paper, we presented the algorithm for the inclusion problem of extended regular expressions (eREs). The algorithm is based on the transformation of eREs into monadic counting automata (MCAs). The MCAs are compact representations of eREs because the number of states in MCAs does not depend on the bounds in the counting operator. Thus the product of MCAs is often significantly smaller than if eREs are transformed into nondeterministic finite automata (NFAs).

We have shown that we can solve inclusion problem for new sets of eREs if the eREs are transformed into MCAs instead of NFAs, because the solution working with NFAs fail due to exponential blow up in the number of states. Moreover, we note that the advantage of our algorithm grows if the bounds and occurrences of the counting get larger.

Although the eREs are common in practice, there are still some regular expressions that do not satisfy our abstract syntax of eREs. The possible extension is to avoid the restriction that counting operator is limited to character classes, e.g., `(abc){5,10}` denoting all strings where `abc` appears 5–10 times.

Acknowledgements

I would like to thank my supervisor Ing. Ondřej Lengál, Ph.D. for his help.

References

[1] Lukáš Holík, Ondřej Lengál, Olli Saarikivi, Lenka Turoňová, Margus Veanes, and Tomáš Vojnar. Succinct determinisation of counting automata via sphere construction. In *In Proc. of 17th Asian Symposium on Programming Languages and Systems - APLAS'19*, number 11893, pages 468–489. Springer Verlag, 2019.

[2] Marty Roesch. Snort: A Network Intrusion Detection and Prevention System. https://www.snort.org/.

[3] RegExLib.com: The Internet's First Regular Expression Library. https://regexlib.com/.

[4] Dag Hovland. Regular expressions with numerical constraints and automata with counters. In *Proceedings of the 6th International Colloquium on Theoretical Aspects of Computing*, ICTAC '09, page 231–245, Berlin, Heidelberg, 2009. Springer-Verlag.

[5] R. Smith, C. Estan, S. Jha, and I. Siahaan. Fast signature matching using extended finite automaton (XFA). In *Proceedings of the 4th International Conference on Information Systems Security*, ICISS '08, page 158–172, Berlin, Heidelberg, 2008. Springer-Verlag.

[6] Parosh Aziz Abdulla, Yu-Fang Chen, Lukáš Holík, Richard Mayr, and Tomáš Vojnar. When simulation meets antichains: On checking language inclusion of nondeterministic finite (tree) automata. In *Proceedings of the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'10, page 158–174, Berlin, Heidelberg, 2010. Springer-Verlag.

[7] David L. Dill, Alan J. Hu, and Howard Wong-Toi. Checking for language inclusion using simulation preorders. In Kim G. Larsen and Arne Skou, editors, *Computer Aided Verification*, pages 255–265, Berlin, Heidelberg, 1992. Springer Berlin Heidelberg.

[8] Aaron R. Bradley and Zohar Manna. *The Calculus of Computation: Decision Procedures with Applications to Verification*. Springer Publishing Company, Incorporated, 1st edition, 2010.