Permutation Rewriting and Algorithmic Verification

Ahmed Bouajjani   Anca Muscholl   Tayssir Touili

LIAFA, Université Paris VII
2, place Jussieu, case 7014
F-75251 Paris Cedex 05
e-mail: {Ahmed.Bouajjani,Anca.Muscholl,Tayssir.Touili}@liafa.jussieu.fr

Abstract

We propose a natural subclass of regular languages (Alphabetic Pattern Constraints, APC) which is effectively closed under permutation rewriting, *i.e., under iterative application of rules of the form $ab \to ba$. It is well-known that regular languages do not have this closure property, in general. Our result can be applied for example to* regular model checking, *for verifying properties of parametrized linear networks of regular processes, and for modeling and verifying properties of asynchronous distributed systems.*

We also consider the complexity of testing membership in APC and show that the question is complete for PSPACE when the input is an NFA, and complete for NLOGSPACE when it is a DFA. Moreover, we show that both the inclusion problem and the question of closure under permutation rewriting are PSPACE-complete when we restrict to the class APC.

1 Introduction

Regular languages in their various representations (finite state automata, regular expressions, monadic first or second order logics, temporal logics, etc) are extensively used for modelling and verifying properties of concurrent systems. The main reason is that regular languages enjoy important closure and decidability properties. They were used for modelling behaviors of systems in form of sets of computational sequences, often modulo some abstraction relation [6, 14, 23]. Recently, *regular model checking* was proposed as a technique of symbolic representation of sets of configurations in the analysis of infinite state systems like pushdown automata, fifo-channel systems, and parametrized networks of processes, see e.g. [1, 3, 4, 5, 11, 19, 24]. A fundamental problem which appears in all these areas is then the following one: Given a regular language L and a relation $\mathcal{R}$ on sequences given either by a transducer or a rewriting system, we want to compute – if possible– the set $\mathcal{R}^*(L)$, which is the $\mathcal{R}$-closure of L ($\mathcal{R}^*$ denotes the reflexive, transitive closure of $\mathcal{R}$). Since unrestricted rewriting systems have full computational power, we have to impose restrictions on the rewriting rules and on the regular languages we consider, in order to be able to compute $\mathcal{R}^*(L)$. In this paper we consider permutation rewriting rules of the form $ab \to ba$, where a, b are letters of a given alphabet Σ. Such rewriting rules are usually called *semi-commutation rules* in Mazurkiewicz trace theory [7]. Our primary goal is to determine a suitable subclass of regular languages for which we can effectively compute the $\mathcal{R}$-closure, for *any* semi-commutation rewriting system $\mathcal{R}$.

The problem of computing the closure of a language under a semi-commutation rewriting systems appears naturally in several areas. For instance, partial-order reduction methods [9, 17, 22] applied in traditional model-checking rely on the fact that the property we want to verify does not distinguish different linearizations of the same partial order. This allows to perform an improved, reduced exploration of large systems. In the simplest setting, a partial-order property means that the property is closed under *partial commutation rules*, i.e., (symmetric) rules of the form $ab \leftrightarrow ba$, meaning that two actions a and b are causally independent. However, it is often much more convenient to express a property (or its negation) as a set of behaviors (or bad behaviors), regardless of all possible interleavings between independent actions. Therefore, if a given property ϕ is not a partial-order property, then we can first compute its closure $\mathcal{R}^*(\phi)$. The interest in doing this is that closing ϕ is in general much less expensive than a full exploration of the system.

In the context of *regular model checking* [5, 11, 19], a set of configurations is represented as a regular language and the actions of a system are modeled as a rewriting system $\mathcal{R}$. Then, the verification problem amounts to compute the $\mathcal{R}$-closure $\mathcal{R}^*(L)$ for a given set of initial configurations L. This allows for instance to analyze parameterized systems with arbitrarily many identical finite state processes which are connected linearly. Here, a configuration is a sequence of control states of individual processes, the i-th element of the sequence being the state of the i-th process. Thus, sets of configurations of arbitrary lengths, corresponding to systems with arbitrary number of processes, are described by a regular language. This allows a *uniform verification*, i.e., for any number of processes. In protocols based on information exchange between neighbors (e.g., token exchange, mutual exclusion, leader election), certain transitions can be modeled by semi-commutation rewriting rules of the form $ab \to ba$. Being able to compute the $\mathcal{R}$-closure $\mathcal{R}^*(L)$ allows for instance to compute the effect of meta-transitions corresponding to the semi-commutation rewriting rules. Take as an example a simple mutual exclusion protocol, where linearly ordered processes can exchange a token which gives the right to enter a critical section. Suppose that the state of a process is 1 if it owns the token, and 0 otherwise. The initial configuration is then the regular expression 10^* (note that the number of processes is not fixed). An (abstract) transition rule of the system can be represented by the semi-commutation one-rule system $\mathcal{R} = \{10 \to 01\}$. We can now compute the reachable set of configurations $\mathcal{R}^*(10^*) = 0^*10^*$ and check for instance that the intersection with the set of bad configurations $(0+1)^*1(0+1)^*1(0+1)^*$ is empty.

Thus, given a regular language L and a *semi-commutation relation* $\mathcal{R}$, we want to compute the reflexive, transitive closure $\mathcal{R}^*(L)$. However, it is not hard to see that semi-commutation rewriting does not preserve regularity. In our setting we would like to have a subclass of regular languages which is effectively closed under several operations, such as *union, intersection* and *semi-commutation rewriting*. Closure under these operations allows us to perform automatically a sequence of operations as required for example in the iterative fixed point computations of regular model checking. Clearly, we want a subclass of regular languages with a decidable membership problem. The solution proposed by this paper is the class of *Alphabetic Pattern Constraints* (APC), which appears naturally in many contexts of verification of concurrent systems. APC corresponds to finite unions of languages of the form $\Sigma_0^* a_1 \Sigma_1^* \cdots a_n \Sigma_n^*$, where every Σ_i denotes a subset of the alphabet Σ and every $a_i \in \Sigma$ denotes a single letter. For instance, the regular expressions in the token ring example above are APC expressions. APCs can be used for example for (negated) safety properties expressing the presence of patterns within computations or configurations, such as required for mutual exclusion. The class of APCs actually corresponds to the Σ_2-level of the quantifier-alternation hierarchy of the first-order logic of sequences [21]. We show that this class satisfies all the closure properties stated above. In particular, our first main result is that APC is closed under semi-commutation rewriting and we provide an effective algorithm that computes the closure $\mathcal{R}^*(L)$, given a semi-commutation system $\mathcal{R}$ and an APC language L.

For regular model checking we consider also circular semi-commutation rewriting. Indeed, the simplest interconnection topology in distributed computing is the ring topology. A (parameterized) configuration corresponds then to a circular word, i.e. a word $x_1 \cdots x_n$ with the understanding that x_1 follows x_n. This means that $x_1 \cdots x_n$ and its *conjugated* words $x_k x_{k+1} \cdots x_n x_1 \cdots x_{k-1}$ represent the same configuration. Thus, the set of configurations of a ring network is a set of words L which is closed under conjugacy, i.e. $L = \mathrm{Conj}(L)$. For instance, for the Token Ring Protocol the set of initial configurations on a ring is $\mathrm{Conj}(10^*) = 0^*10^*$. Our second main result shows that for any semi-commutation rewriting system $\mathcal{R}$, the circular $\mathcal{R}$-closure $(\mathrm{Conj} \circ \mathcal{R}^*)^*(L)$ of any language $L \subseteq \Sigma^*$ can be computed as long as the reflexive, transitive closure $\mathcal{R}^*(L)$ is computable. For this we show that $(\mathrm{Conj} \circ \mathcal{R}^*)^*(L) = (\mathrm{Conj} \circ \mathcal{R}^*)^{2|\Sigma|}(L)$. This implies that for each APC language L the circular $\mathcal{R}$-closure $(\mathrm{Conj} \circ \mathcal{R}^*)^*(L)$ is in APC and can be effectively computed.

In the last part of this paper we establish complexity bounds for basic problems concerning the class of APC languages. We show that deciding whether a regular language belongs to APC is complete for PSPACE when the language is given by a non-deterministic automaton, respectively complete for NLOGSPACE, when the input is a deterministic automaton. Moreover, we show that testing whether an APC language is closed under a semi-commutation rewriting relation, as well as the inclusion problem for APC, are both PSPACE-complete problems. These results suggest that APC is as "hard" as the whole class of regular languages, which means in some sense that APCs are expressive enough for specifying interesting properties. It is also interesting to note that APCs correspond to the smallest level in the quantifier-alternation hierarchy of first-order logic which has this "hardness property". Indeed, languages in Σ_1 and Π_1 correspond respectively to upward and downward subword-closed sets. For example, Π_1 is precisely the class SRE [1], for which it can be shown that inclusion can be checked in polynomial time.

Related work: Problems related to closure of languages under semi-commutations have been studied in the community of trace theory (see e.g. chapter 12 in [7] for a survey). However, the problems addressed here and our results have a different flavor. Our aim is to identify subclasses of regular languages which are closed under *all* semi-commutation rewriting relations, whereas classical results of trace theory aim at providing for a given semi-commutation relation $\mathcal{R}$ sufficient conditions on regular languages L ensuring that the $\mathcal{R}$-closure of L remains regular. Moreover, these conditions on the languages always depend on the relation $\mathcal{R}$.

APC languages have been intensively studied in logic and algebra. As mentioned above, they correspond to the Σ_2-level of the quantifier-alternation hierarchy of first order logic, i.e., to formulas of the form $\exists^*\forall^*\phi$, where ϕ is quantifier-free. The class APC has also an algebraic characterization, it corresponds to level 3/2 of Straubing's concatenation hierarchy of star-free sets. Moreover, it is the largest hierarchy level known to be decidable [18].

The complexity of deciding whether a regular ω-language is closed under commutation rewriting was considered in [16, 20]. Several works on regular model checking deal with the problem of computing the closure of a regular language under a rewriting system [2, 5, 8, 10, 19]. However, the techniques proposed in these papers are not complete, in general. Moreover, they do not cover the case of semi-commutation rewriting.

2 Alphabetic Pattern Constraints

In this section we define the class of *Alphabetic Pattern Constraints (APC)* and show that APC is closed under union, intersection and conjugacy, but not under complementation.

Definition 2.1 *Let Σ be a finite alphabet. An atomic expression over Σ is either a letter a of Σ or a star expression $(a_1+a_2+\cdots+a_n)^*$, where $a_1, a_2, \ldots, a_n \in \Sigma$. The set of star expressions is denoted by $S(\Sigma)$.*

A product p over Σ^* is a (possibly empty) concatenation $e_1e_2\cdots e_n$ of atomic expressions $e_1, \ldots, e_n$ over Σ. We use ϵ to denote the empty product.

An Alphabetic Pattern Constraint (APC) over Σ^* is an expression of the form $p_1+\cdots+p_n$, where $p_1, \ldots, p_n$ are products over Σ^*. By $APC(\Sigma)$ we denote the set of regular languages described by some APC over Σ^*.

In the rest of the paper we will not distinguish between a regular expression and the language that it defines. However, the input for our algorithms in Sections 3, 5 will be an APC expression.

It can be easily noted that the class of APCs is not closed under complementation. Consider for example the alphabet $\Sigma = \{a, b\}$ and the APC language $\Sigma^*aa\Sigma^* + \Sigma^*bb\Sigma^* + b\Sigma^* + \Sigma^*a$. It is not difficult to check that its complement $(ab)^*$ does not belong to APC.

Let us introduce some notations which will be used in the analysis of operations on APCs. Let $p = e_1\cdots e_n$ be a product, then the length of p, denoted $l(p) = n$, is the number of atomic expressions in p. Let $e = \sum_i p_i$ be an APC expression, then the *length* of e is defined as $l(e) = \max_i l(p_i)$. The *size* of an expression is the sum of the lengths of its products. For a language L we denote by $\alpha(L)$ the set of letters of Σ appearing in L. As usual, $|L|$ denotes the cardinality of L. For a string $w \in \Sigma^*$ and a letter $a \in \Sigma$, we denote by $|w|_a$ the number of occurrences of a in w.

We recall that two words x and $y \in \Sigma^*$ are called *conjugated* if $x = uv$ and $y = vu$ for some $u, v \in \Sigma^*$. For a language L, we denote by $\mathrm{Conj}(L)$ the set $\{uv \in \Sigma^* \mid vu \in L\}$ of conjugates of words from L. For a class of languages $\mathcal{C}$ to be *closed under conjugacy* we require that $L \in \mathcal{C}$ implies that $\mathrm{Conj}(L) \in \mathcal{C}$.

We conclude this section by stating some straightforward closure properties of APC. The proofs are not difficult and can be found in the full version of the paper.

Proposition 2.1 *The class APC is closed under union, intersection and conjugacy.*

Remark 2.1 *While union and conjugacy are polynomial operations, computing the intersection of two APC languages yields an expression of exponential size. The worst-case is indeed exponential, as shown by the following example. Consider the products $p_n = b^*(ab^*)^n$ and $q_n = (a^*b)^n a^*$ each of size $2n+1$. Then $\{w \in (a+b)^* \mid |w|_a = |w|_b = n\} = p_n \cap q_n$ is a finite set with the property that every APC expression for $p_n \cap q_n$ is of exponential size.*

3 Semi-Commutation Rewriting and APC

Semi-commutations are a natural way of expressing causal independence in concurrent systems in an algebraic way. The original notion was proposed in the late 70's by Mazurkiewicz [12] for the semantics of Petri nets. Mazurkiewicz traces and semi-traces are a model of true concurrency with nice algorithmical properties, which can be exploited for automatic verification methods.

A semi-commutation relation $\mathcal{R}$ defined over an alphabet Σ of actions is an irreflexive binary relation, i.e., a subset of $\Sigma \times \Sigma \setminus \{(a,a) \mid a \in \Sigma\}$. The idea is that two actions a, b with $(a,b) \in \mathcal{R}$ are (partially) causally independent, in the sense that we can rewrite ab into ba in every context. In many cases the relation $\mathcal{R}$ is asymmetric, for instance in a producer-consumer model we may rewrite $cp \to pc$, but not the other way round.

It is not difficult to see that semi-commutation rewriting does not preserve regulairty. Consider for example the set $L = (ab)^*$ and the semi-commutation system $\mathcal{R} = \{ba \to ab\}$. Then, $\mathcal{R}^*(L)$ is the (non-regular) set of all words having the same number of a's and b's, and such that all their prefixes contain at least as many a's as b's. Therefore, we cannot hope to represent the relation $\mathcal{R}^*$ by a finite transducer, in general.

We associate with each semi-commutation relation $\mathcal{R}$ a rewriting relation $\rho_{\mathcal{R}} \subseteq \Sigma^* \times \Sigma^*$, which is defined by $(w, w') \in \rho_{\mathcal{R}}$ if there exist $w_1, w_2 \in \Sigma^*$ and $a, b \in \Sigma$ such that $(a,b) \in \mathcal{R}$, $w = w_1 ab w_2$, and $w' = w_1 ba w_2$. As usual, we denote by $\rho^*_{\mathcal{R}}$ the reflexive, transitive closure of $\rho_{\mathcal{R}}$. For a language $L \subseteq \Sigma^*$, we denote its $\mathcal{R}$-*closure* $\{v \in \Sigma^* \mid \exists u \in L, (u,v) \in \rho^*_{\mathcal{R}}\}$ by $\mathcal{R}^*(L)$.

The notation of semi-commutations can be extended to sets by letting for each subsets $X, Y \subseteq \Sigma$:

$$(X,Y) \in \mathcal{R} \text{ if } X \times Y \subseteq \mathcal{R}\,.$$

Let $\mathcal{R}$ be a semi-commutation relation, then we denote by $\delta_{\mathcal{R}}$ the value

$$\delta_{\mathcal{R}} = \max_{a \in \Sigma}\{|Y| \mid Y \subseteq \Sigma \text{ such that } (a,Y) \in \mathcal{R}\}\,.$$

We will assume throughout the paper that $\mathcal{R} \neq \emptyset$, thus $\delta_{\mathcal{R}} > 0$.

Our first main result is stated in the theorem below. The remaining of this section consists in describing the algorithm underlying Theorem 3.1. Several proofs are omitted and can be found in the full version of the paper.

Theorem 3.1 *For each APC expression L, the $\mathcal{R}$-closure $\mathcal{R}^*(L)$ belongs to APC and can be computed effectively. Moreover, the length of the computed expression is in $\mathcal{O}((\delta_{\mathcal{R}}+1)^{l(L)})$.*

Since $L \in \mathrm{APC}(\Sigma)$ is a finite union of products, its closure $\mathcal{R}^*(L)$ is the union of closures of its products. Hence, it suffices to show how to compute effectively $\mathcal{R}^*(p)$ for a given product p. For this we use the $\mathcal{R}-$shuffle operation defined below. The idea is to compute $\mathcal{R}^*(e_1 \cdots e_n)$ recursively, i.e., computing first $\mathcal{R}^*(e_2 \cdots e_n)$ and using that $\mathcal{R}^*(e_1) = e_1$. The recursive step means that we need to compute $\mathcal{R}^*(eL)$, for an $\mathcal{R}$-closed APC expression L and an atomic expression e, an operation which will be performed also recursively. For our computations we need the following notations:

Definition 3.1 *Let $\mathcal{R}$ be a semi-commutation relation. Given two words x and y of Σ^*, the $\mathcal{R}-$shuffle of x and y, denoted by $x \sqcup\!\sqcup_{\mathcal{R}} y$, is the set of words of the form $x_1 y_1 \cdots x_n y_n$ with $x = x_1 \cdots x_n$, $y = y_1 \cdots y_n$, $x_i, y_i \in \Sigma^*$ for all $1 \le i \le n$ and such that $(\alpha(x_i), \alpha(y_j)) \in \mathcal{R}$ for all $j < i$.*

The $\mathcal{R}-$shuffle extends to sets $X, Y \subseteq \Sigma^*$ by letting

$$X \sqcup\!\sqcup_{\mathcal{R}} Y = \{x \sqcup\!\sqcup_{\mathcal{R}} y \mid x \in X, y \in Y\}.$$

Note that for all $x, y \in \Sigma^*$, we have $\mathcal{R}^*(xy) = \mathcal{R}^*(x) \amalg_{\mathcal{R}} \mathcal{R}^*(y)$. The next lemma shows how to compute $\mathcal{R}^*(LK)$ when L and K are already $\mathcal{R}$-closed.

Lemma 3.1 *Let L and K be two $\mathcal{R}$-closed sets, i.e., we suppose that we have both $\mathcal{R}^*(L) = L$ and $\mathcal{R}^*(K) = K$. Then we have $\mathcal{R}^*(LK) = L \amalg_{\mathcal{R}} K$.*

Since any atomic expression is $\mathcal{R}$-closed we can state the following lemma:

Lemma 3.2 *Let $e_1, e_2, \ldots, e_n$ be atomic expressions and let $p = e_1 e_2 \cdots e_n$ be a product, then we have:*

$$\mathcal{R}^*(p) = e_1 \amalg_{\mathcal{R}} (e_2 \amalg_{\mathcal{R}} (\cdots (e_{n-1} \amalg_{\mathcal{R}} e_n) \cdots)) .$$

By the preceding lemma we can compute $\mathcal{R}^*(p)$ recursively. Lemma 3.3 and Proposition 3.1 below are the basic cases of our algorithm.

Lemma 3.3 *Let E be a subset of Σ and let $a \in \Sigma$ be a letter, then we have:*

$$E^* \amalg_{\mathcal{R}} a = \mathcal{R}^*(E^* a) = E^* a E'^* ,$$

where $E' = \{x \in E \mid (x, a) \in \mathcal{R}\}$.

Example 3.1 *Consider the product $p = (e+f+g)^* d$, and the semi-commutation relation $\mathcal{R}_1 = \{(f,d),(g,d)\}$. Then the previous lemma yields*

$$\mathcal{R}_1^*(p) = (e+f+g)^* \amalg d = (e+f+g)^* d (f+g)^* .$$

The next proposition is the main technical result needed for the proof of Theorem 3.1. It shows that the $\mathcal{R}$-closure of the product of two star expressions belongs to APC. In particular, note that the length of the products in the expression given below is bounded above by a constant which is polynomial in Σ and $\mathcal{R}$.

Proposition 3.1 *Let E and F be two subsets of Σ, then $E^* \amalg_{\mathcal{R}} F^* = \mathcal{R}^*(E^* F^*)$ equals*

$$\sum E^*(E_1+F_1)^* \cdots (E_n+F_n)^* F^* ,$$

where the sum is taken over all subsets E_i and F_i of Σ satisfying the following conditions:

- $\emptyset \neq E_n \subsetneq \cdots \subsetneq E_1 \subseteq E$,
- $\emptyset \neq F_1 \subsetneq \cdots \subsetneq F_1 \subseteq F$,
- $(E_i, F_j) \in \mathcal{R}$ *for all* $1 \leq j \leq i \leq n$.

Proof. The first equality can be inferred as previously from Lemma 3.1 since E^* and F^* are closed under $\mathcal{R}$.

Let us consider now the second equality. It is obvious that $E^*(E_1+F_1)^* \cdots (E_n+F_n)^* F^* \subseteq \mathcal{R}^*(E^* F^*)$, whenever E_i and F_i satisfy $(E_i, F_j) \in \mathcal{R}$ for all $j \leq i$.

Conversely, let $w \in E^* \amalg_{\mathcal{R}} F^* = \mathcal{R}^*(E^* F^*)$. We can write $w = u_1 v_1 u_2 v_2 \cdots u_m v_m$ with $u_i \in E^*$, $v_i \in F^*$, and such that $(\alpha(u_i), \alpha(v_j)) \in \mathcal{R}$ holds for all $j < i$. Clearly, we can assume that $u_i, v_j \neq \epsilon$ for all $i \neq 1$ and $j \neq m$.

Consider the sequences $(k_i)_{1 \leq i \leq n}$, $(E_i)_{1 \leq i \leq n}$ and $(F_i)_{1 \leq i \leq n}$ defined inductively by:

- $k_1 = 1$, $k_i = \min\{j \mid k_{i-1} < j < m, v_j \notin F_{i-1}^*\}$,
- $E_i = \alpha(u_{k_i+1} \cdots u_m)$,
- $F_i = \{y \in F \mid \forall x \in E_i, (x,y) \in \mathcal{R}\}$.

By definition we have $E_{i+1} \subsetneq E_i \subseteq E$, hence $F_i \subsetneq F_{i+1} \subseteq F$ for all i. Moreover, $(E_i, F_i) \in \mathcal{R}$ holds for all i, therefore $(E_i, F_j) \in \mathcal{R}$ for all $j \leq i$. Finally, we note that $u_{k_i+1} \cdots u_{k_{i+1}} \in E_i^*$ and $v_{k_i} \cdots v_{k_{i+1}-1} \in F_i^*$, which yields the result.

□

Remark 3.1 *Note that the cardinality of E_1 is at most $\delta_{\mathcal{R}}$, since we require that $F_1 \neq \emptyset$ and $(E_1, F_1) \in \mathcal{R}$. Moreover, since there is a strict inclusion between the E_i's, the length of the products in the expression for $\mathcal{R}^*(E^* F^*)$ is at most $\delta_{\mathcal{R}} + 2$.*

Example 3.2 *Consider the product $p = (a+b+c)^*(e+f+g)^*$, and the semi-commutation relation $\mathcal{R}_2 = \{(a,e),(c,g),(b,e),(b,f)\}$. From the proposition above it follows that $\mathcal{R}_2^*(p) = (a+b+c)^* \amalg_{\mathcal{R}_2} (e+f+g)^* = (a+b+c)^*(c+g)^*(e+f+g)^* + (a+b+c)^*(a+b+e)^*(b+e+f)^*(e+f+g)^*$.*

We are now going to compute effectively $\mathcal{R}^*(p) = \mathcal{R}^*(e_1 e_2 \cdots e_n)$ and show that it belongs to APC. By Lemma 3.3 and Proposition 3.1 we have shown the result for $n = 2$. Suppose now that $\mathcal{R}^*(e_2 \cdots e_n) = \sum f_1 f_2 \cdots f_k$, with f_i denoting atomic expressions, and let us show that $\mathcal{R}^*(e_1 e_2 \cdots e_n)$, which equals $\sum e_1 \amalg_{\mathcal{R}} (f_1 f_2 \cdots f_k)$, also belongs to APC. Thus, we only need to compute $e_1 \amalg_{\mathcal{R}} (f_1 f_2 \cdots f_n)$ and to show that it is of the required form. To do this we will distinguish two cases, depending on e_1

being a letter or a star expression. The first case is straightforward:

Lemma 3.4 *Let $a \in \Sigma$ and $f_1, \ldots, f_n$ be atomic expressions, then*

$$a \shuffle_{\mathcal{R}} (f_1 f_2 \cdots f_n) = \sum_j g_1 \cdots g_j a h_j f_{j+1} \cdots f_n$$

such that, for all $i \leq j$ we have:

- *if $f_i \in S(\Sigma)$, then $g_i \in S(\Sigma)$ with $\alpha(g_i) = \{x \in \alpha(f_i) \mid (a,x) \in \mathcal{R}\}$,*
- *if $f_i = b \in \Sigma$ and $(a,b) \in \mathcal{R}$, then $g_i = b$.*

Moreover, $h_j = f_j$ when $f_j \in S(\Sigma)$ and $h_j = \varepsilon$ when $f_j \in \Sigma$.

Example 3.3 *Let $\mathcal{R}_3$ be the semi-commutation relation $\mathcal{R}_3 = \{(h,a),(h,e)\}$. Then the previous lemma implies that $h \shuffle_{\mathcal{R}_3} (a+b+c)^*(a+b+e)^*(b+e+f)^* = a^*h(a+b+c)^*(a+b+e)^*(b+e+f)^* + (a+e)^*h(a+b+e)^*(b+e+f)^*$.*

The next proposition generalizes Lemma 3.3 and Proposition 3.1.

Proposition 3.2 *Let E and F be two subsets of Σ, $a \in \Sigma$ a letter, and L be a language of Σ^*, then we have:*

1. *$E^* \shuffle_{\mathcal{R}} (aL) = (E^* \shuffle_{\mathcal{R}} a)(E'^* \shuffle_{\mathcal{R}} L)$, where $E' = \{b \in E \mid (b,a) \in \mathcal{R}\}$.*
2. *$E^* \shuffle_{\mathcal{R}} (F^*L)$ equals*

$$\sum_{\substack{(E',F') \in \mathcal{R} \\ E' \subseteq E,\, F' \subseteq F}} (E^* \shuffle_{\mathcal{R}} F'^*)(E'^* \shuffle_{\mathcal{R}} L).$$

Corollary 3.1 *Let E and F be two subsets of Σ, and let L be a language of Σ^*, then $E^* \shuffle_{\mathcal{R}} (F^*L)$ equals:*

$$\sum E^*(E_1+F_1)^*(E_2+F_2)^* \cdots (E_k+F_k)^*(E_k^* \shuffle_{\mathcal{R}} L),$$

where the union is taken over all subsets E_i and F_i of Σ satisfying:

- *$E_k \subsetneq \cdots \subsetneq E_1 \subseteq E$,*
- *$\emptyset \neq F_1 \subsetneq \cdots \subsetneq F_k \subseteq F$,*
- *$(E_i, F_j) \in \mathcal{R}$ for all $1 \leq j \leq i \leq k$.*

Proof. The inclusion from right to left is straightforward. By Proposition 3.2 it remains to show that

$$(E^* \shuffle_{\mathcal{R}} F'^*)(E'^* \shuffle_{\mathcal{R}} L) \subseteq \sum E^*(E_1+F_1)^* \cdots (E_k+F_k)^*(E_k^* \shuffle_{\mathcal{R}} L),$$

where $E' \subseteq E$ and $F' \subseteq F$ are subsets satisfying $(E',F') \in \mathcal{R}$. This can be obtained from Proposition 3.1 applied to $E^* \shuffle_{\mathcal{R}} F'^*$, by noting that the sequence of $(E_i)_i$ can be chosen such that each E_i is maximal with the property that $(a,b) \in \mathcal{R}$ for all $a \in E_i, b \in F_i$. Hence, $E' \subseteq E_i$ for all i yields the claimed expression.

□

Example 3.4 *Let $\mathcal{R}_4$ be the semi-commutation relation $\mathcal{R}_4 = \{(a,e),(c,g),(b,e),(b,f),(a,d)\}$. Then from the last proposition and from example 3.2 it follows that*
$(a+b+c)^* \shuffle_{\mathcal{R}_4} (e+f+g)^*d(f+g)^* =$
$(a+b+c)^*(c+g)^*(e+f+g)^*d(f+g)^* +$
$(a+b+c)^*(a+b+e)^*(b+e+f)^*(e+f+g)^*d(f+g)^* +$
$(a+b+c)^*(a+b+e)^*da^*(f+g)^*$.

Proposition 3.2 and Corollary 3.1 yield the recursive step for computing $E^* \shuffle_{\mathcal{R}} (f_1 f_2 \cdots f_n)$:

Proposition 3.3 *Let $E \subseteq \Sigma$ and let $f_1, \ldots, f_n$ be atomic expressions. Then $E^* \shuffle_{\mathcal{R}} (f_1 f_2 \cdots f_n)$ equals*

1. *For a star expression f_1:*
   $\sum E^*(E_1+F_1)^* \cdots (E_k+F_k)^*(E_k^* \shuffle_{\mathcal{R}} f_2 \cdots f_n)$, where the union is taken over all subsets E_i, F_i satisfying $E_{i+1} \subsetneq E_i \subseteq E$, $\emptyset \neq F_i \subsetneq F_{i+1} \subseteq \alpha(f_1)$ and $(E_i, F_j) \in \mathcal{R}$ for all $j \leq i$.
2. *For a single letter $f_1 = a$:*

$$E^* a (E'^* \shuffle_{\mathcal{R}} f_2 \cdots f_n).$$

We can now describe the algorithm for computing the closure of an APC expression $\sum e_1 \cdots e_n$ under a semi-commutation rewriting relation $\mathcal{R}$. We compute recursively $\mathcal{R}^*(e_2 \cdots e_n) = \sum f_1 \cdots f_k$. The recursive step is given by Lemma 3.4, if e_1 is a letter. Otherwise, for $e_1 = E^*$ we apply Proposition 3.3, which is itself a recursive algorithm. It is easily seen that each step preserves containment in APC. This shows Theorem 3.1.

Remark 3.2 *We note that for a product p of length n the length of the products of the expression computed for $\mathcal{R}^*(p)$ is at most $\mathcal{O}((\delta_{\mathcal{R}}+1)^n)$. Moreover, since there exist $2^{|\Sigma|}+|\Sigma|$ different atomic expressions, it follows that the size of $\mathcal{R}^*(p)$ is at most $2^{O(|\Sigma|(\delta_{\mathcal{R}}+1)^n)}$.*

4 Applications

As mentioned in the introduction, we can use our results for applying partial-order reduction techniques in model-checking, even if the original property is not a partial-order property. This idea can be used for example in the validation of scenarios described by High-Level Message Sequence Charts (HMSC). HMSCs are a graphical specification language for communications protocols, standardized by the ITU and integrated in UML. An HMSC scenario is a partial-order model for asynchronous fifo message exchange of concurrent processes. Assume for example that we have a system S including two processes P and Q and that we want to verify that P cannot send more than two messages to Q before getting an acknowledgement back from Q. Let us denote the set of possible actions by Σ, the send action of P to Q by s, the receive action of Q from P by r and let Σ_P (resp. Σ_Q) denote events on P (resp. on Q). Hence, a bad scenario contains for example an occurrence of the sequence $srsr$, which means that two messages have been transmitted from P to Q without an acknowledgement between them. So, let $\phi = \Sigma^* srsr \Sigma^*$ be the set of sequences containing this bad subsequence, and suppose we want to verify that an HMSC system S satisfies $\neg\phi$. Clearly, ϕ is not a partial-order property, since ϕ does not contain for example the sequence $ssrr$. We can consider the semi-commutation rule $rs \to sr$ which expresses that communication is asynchronous. By applying our algorithm with suitable rules such as $\mathcal{R} = \{rs \to sr\}$ we obtain the partial-order property $\mathcal{R}^*(\phi)$:

$$\Sigma^* s(\Sigma \setminus \Sigma_P)^* r(\Sigma \setminus (\Sigma_P \cup \Sigma_Q))^* s(\Sigma \setminus \Sigma_Q)^* r\Sigma^*$$
$$+\Sigma^* s(\Sigma \setminus \Sigma_P)^* s\Sigma^* r(\Sigma \setminus \Sigma_Q)^* r\Sigma^* .$$

Now, for verifying that S satisfies $\neg\phi$ we can consider a succinct representation of the system S, which corresponds to the transition system underlying S and which is polynomial in the size of the given HMSC system, and then check that $S \cap \mathcal{R}^*(\phi)$ is empty. Since we consider an $\mathcal{R}$-closed property, it is not necessary to compute the closure of the system S, which is an expensive operation, and even impossible in general (linearizations of HMSCs are not regular [13]). The same holds also in the case of "positive reasoning": for verifying that S satisfies a property ϕ, it suffices to construct the $\mathcal{R}$-closure of ϕ and check that $S \subseteq \mathcal{R}^*(\phi)$.

Further examples showing that APC properties occur naturally in the verification of concurrent systems is the so-called "matching with gaps" problem in HMSCs [15], which is a kind of weak model-checking. Other examples from distributed computing are negations of (some) safety properties when APCs are used to express bad patterns (scenarios) like in the examples shown above. Furthermore, in the context of regular model checking, it turns out that the reachability sets of many infinite-state systems and parameterized systems, including communication protocols like the alternating bit and the sliding window, and parameterized mutual exclusion protocols such as the token ring, Szymanski's, Burns', or Dijkstra's protocols, are all expressible as APCs. Being able to compute the $\mathcal{R}$-closure $\mathcal{R}^*(L)$ for a semi-commutation system $\mathcal{R}$ allows us to compute the effect of meta-transitions corresponding to the semi-commutation rules, and hence to accelerate the process of computing the set of reachable configurations.

5 Circular Rewriting

In this section we consider the problem of computing $\mathcal{R}^*(L)$ when L consists of circular words. This amounts to assume that L is closed under conjugacy, $L = \mathrm{Conj}(L)$. Recall that $\mathrm{Conj}(L) = \{vu \mid uv \in L\}$ denotes the closure of L under conjugacy. The question of computing the $\mathcal{R}$-closure in this framework arises naturally in regular model checking when processes are ordered circularly in a ring.

Let $\mathcal{R} \subseteq \Sigma \times \Sigma$ be a semi-commutation relation over Σ. We associate with $\mathcal{R}$ the *circular rewriting relation* $\mathcal{R}_c \subseteq \Sigma^* \times \Sigma^*$ defined as follows. For any pair of words x and y in Σ^*, we define $(x,y) \in \mathcal{R}_c$ if we can write

$$uv \in \mathcal{R}^*(x) \text{ and } y \in \mathcal{R}^*(vu)\,,$$

for some $u, v \in \Sigma^*$. Note that the circular rewriting relation $\mathcal{R}_c$ is the composition of the (rewriting) relations $\mathcal{R}^* \circ \mathrm{Conj} \circ \mathcal{R}^*$. As usual, $\mathcal{R}_c^*$ denotes the reflexive, transitive closure of $\mathcal{R}_c$. For a language

L we denote by $\mathcal{R}_c^*(L)$ the *circular $\mathcal{R}$-closure* of L, defined as the set:

$$\mathcal{R}_c^*(L) = \{v \in \Sigma^* \mid \exists\, u \in L \text{ such that } (u,v) \in \mathcal{R}_c^*\}.$$

We will show in this section that the circular $\mathcal{R}$-closure $\mathcal{R}_c^*(L)$ of *any* language L (not necessarily regular) can be obtained by applying alternatively conjugation and permutation rewriting a finite number of times.

The main result of this section can be stated as follows:

Theorem 5.1 *Let $L \subseteq \Sigma^*$, then $\mathcal{R}_c^*(L) = \mathcal{R}_c^{2|\Sigma|}(L)$.*

As a first corollary, we obtain the closure of the class APC under circular rewriting.

Corollary 5.1 *Let L be a APC expression, then $\mathcal{R}_c^*(L)$ is in APC and is effectively computable. The length of the expression computed for $\mathcal{R}_c^*(L)$ is at most $(\delta_{\mathcal{R}}+1)^{\mathcal{O}(l(L)|\Sigma|)}$.*

Proof. This follows directly from $\mathcal{R}_c^*(L) = \mathcal{R}_c^{2|\Sigma|}(L) = (\mathcal{R}^* \circ \mathrm{Conj} \circ \mathcal{R}^*)^{2|\Sigma|}(L)$, together with and APC($\Sigma$) being closed under semi-commutation rewriting and conjugacy (Theorem 3.1 and Proposition 2.1). $\square$

In the remaining of the section we show Theorem 5.1. The proof uses ideas from [7][Ch. 3]. It generalizes (and simplifies) the proof given there for the case where $\mathcal{R}$ is a symmetric relation. As in [7] we need a second relation $\mathcal{C}_{\mathcal{R}}$, called *conjugacy relation*, which is defined as follows for $x, y \in \Sigma^*$:

$$(x,y) \in \mathcal{C}_{\mathcal{R}} \text{ if } \exists\, z \in \Sigma^* \text{ such that } zy \in \mathcal{R}^*(xz)\,.$$

Lemma 5.1 *$\mathcal{R}_c \subseteq \mathcal{C}_{\mathcal{R}}$ and $\mathcal{C}_{\mathcal{R}}$ is reflexive and transitive.*

Proof. For the first claim let $x, y \in \Sigma^*$ be such that $(x,y) \in \mathcal{R}_c$. By definition, there exist u and $v \in \Sigma^*$ such that $uv \in \mathcal{R}^*(x)$ and $y \in \mathcal{R}^*(vu)$, then $uy \in \mathcal{R}^*(uvu) \subseteq \mathcal{R}^*(xu)$.

For the second claim it is easy to see that $\mathcal{C}_{\mathcal{R}}$ is reflexive. Let now $x, y, z \in \Sigma^*$ be such that $(x,y) \in \mathcal{C}_{\mathcal{R}}$ and $(y,z) \in \mathcal{C}_{\mathcal{R}}$. Let then w and $t \in \Sigma^*$ be such that $wy \in \mathcal{R}^*(xw)$ and $tz \in \mathcal{R}^*(yt)$. Then, $(wt)z \in \mathcal{R}^*(wyt) \subseteq \mathcal{R}^*(x(wt))$, which shows that $(x,z) \in \mathcal{C}_{\mathcal{R}}$. $\square$

Theorem 5.2 *Let $x, y \in \Sigma^*$. Suppose that $z \in \Sigma^*$ is such that $zy \in \mathcal{R}^*(xz)$. Then there exist $m \leq 2|\Sigma|$, and words $t_0, \ldots, t_m \in \Sigma^*$ satisfying the following properties:*

- $t_0 \cdots t_m \in \mathcal{R}^*(x)$,
- $y \in \mathcal{R}^*(t_m \cdots t_0)$,
- $(\alpha(t_j), \alpha(t_i)) \in \mathcal{R}$ *for all* $j > i+1$.

Proof. We only sketch the proof idea. We suppose that $zy \in \mathcal{R}^*(xz)$. Then a combinatorial lemma (Levi's Lemma for semi-traces, see [7][Ch. 12]) implies that there exist words $u, v, p, q \in \Sigma^*$ such that $up \in \mathcal{R}^*(x)$, $qv \in \mathcal{R}^*(z)$, $z \in \mathcal{R}^*(uq)$, and $y \in \mathcal{R}^*(pv)$, such that $(\alpha(p), \alpha(q)) \in \mathcal{R}$. Since $qv \in \mathcal{R}^*(uq)$ and $|u|+|q| < |x|+|z|$ if x is nonempty, we can apply induction on $|x|+|z|$ in order to obtain the result. $\square$

Corollary 5.2 $\mathcal{R}_c^* = \mathcal{R}_c^{2|\Sigma|}$.

Proof. First, we show that $\mathcal{C}_{\mathcal{R}} \subseteq \mathcal{R}_c^{2|\Sigma|}$. Let $(x,y) \in \mathcal{C}_{\mathcal{R}}$ with $zy \in \mathcal{R}^*(xz)$ for some z. Let $t_0, \ldots, t_p$ be as stated in the Theorem 5.2. It suffices to show that $(t_0 \cdots t_p, t_p \cdots t_0) \in \mathcal{R}_c^{2|\Sigma|}$. This is due to the fact that $(t_0 t_p \cdots t_1, t_p \cdots t_0) \in \mathcal{R}_c$ and that for each $i \in \{1, \ldots, p-1\}$:

$$(t_0 \cdots t_i t_p \cdots t_{i+1}, t_0 \cdots t_{i-1} t_p \cdots t_i) \in \mathcal{R}_c$$

since

$$t_0 \cdots t_{i-1} t_p \cdots t_i \in \mathcal{R}^*(t_p \cdots t_{i+1} t_0 \cdots t_i).$$

Indeed, to obtain the word $t_0 \cdots t_{i-1} t_p \cdots t_i$ from $t_p \cdots t_{i+1} t_0 \cdots t_i$ by applying $\mathcal{R}$, we start by moving t_{i+1} from left to right, then t_{i+2}, etc.

From Lemma 5.1 we obtain that $\mathcal{R}_c^* \subseteq \mathcal{C}_{\mathcal{R}}$. Since $\mathcal{C}_{\mathcal{R}} \subseteq \mathcal{R}_c^{2|\Sigma|}$, we conclude finally that $\mathcal{R}_c^* = \mathcal{R}_c^{2|\Sigma|}$. $\square$

6 Complexity results

In this section we consider basic complexity questions concerning languages in APC. First, we obtain that both the problem of testing inclusion (or universality) and the problem of deciding whether a language in APC is closed under a semi-commutation relation are PSPACE-complete. Clearly, these are basic operations when we want to perform model-checking on APC properties. For example, we might ask whether an APC property ϕ_1 is covered by another property ϕ_2, i.e., whether $\phi_1 \subseteq \phi_2$. The test for $\mathcal{R}$-closure is important when we want to know whether a property ϕ is already closed under semi-commutation rewriting, since it avoids computing the $\mathcal{R}$-closed expression which has products of exponential size. Moreover, in the fixed point computations of regular model checking we check whether we have already computed the set of all reachable configurations by an equality test.

For lack of space we omit all proofs of the section and refer to the full version of the paper.

Theorem 6.1 *The following problem is PSPACE-complete:*
Input: *An APC expression L over Σ^*.*
Question: *Is $L = \Sigma^*$?*

Corollary 6.1 *Deciding inclusion for languages in APC is PSPACE-complete.*

Theorem 6.2 *The following problem is PSPACE-complete:*
Input: *An APC expression L over Σ and a semi-commutation rewriting system $\mathcal{R} \subseteq \Sigma \times \Sigma$.*
Question: *Does $\mathcal{R}^*(L) = L$ hold?*

Next, we show that the membership problem for the class APC is PSPACE-complete when we are given a non-deterministic automaton. The same question is NLOGSPACE-complete, hence polynomial, when the input is a deterministic automaton. These two last results rely on the characterization of languages in APC by positive varieties given in [18]. It is worthnoting that the algorithm obtained in [18] has complexity in $O(|A| \cdot 2^{|\Sigma|})$, i.e., it is linear in the size of the automaton and exponential in the size of the alphabet. Theorem 6.4 below improves the result by giving an algorithm which is polynomial in both $|A|$ and $|\Sigma|$.

Theorem 6.3 *Deciding whether a regular language, given by a regular expression or a non-deterministic automaton, is an APC language, is a PSPACE-complete problem.*

Theorem 6.4 *Deciding whether a regular language, given by a deterministic automaton, is an APC language, is an NLOGSPACE-complete problem.*

7 Conclusion

We have identified a class of regular expressions which appears naturally in many contexts, in particular in modeling and verifying concurrent systems and in regular model checking, and we have studied its closure properties and its complexity.

In particular, we have shown that the class of APCs is effectively closed under semi-commutation rewriting (for any such rewriting system). As far as we know, this is the first time that a non-trivial subclass of regular properties has been shown to enjoy this property. As mentioned previously, APCs correspond to level 3/2 in Straubing's concatenation hierarchy, and to level Σ_2 in the quantifier-alternation hierarchy of first-order logic. It is interesting to note that this is the largest class in both hierarchies which is closed under semi-commutation rewriting. However, this raises the question of finding other subclasses of regular languages which have the same closure properties as APC. A minimal requirement on such classes is that Parikh images of their languages should correspond to Presburger formulas where linear constraints do not involve more than one free variable. It can be seen for instance that this property does not hold for $(ab)^*$ whereas it holds for all APC languages.

Another novel contribution of our paper is to show that APCs are also closed under circular semi-commutation rewriting. Actually, our proof holds for any class of languages which is effectively closed under semi-commutation rewriting and conjugacy, since we show that for any system $\mathcal{R}$, computing the circular $\mathcal{R}$-closure reduces to a finite iteration (two times the size of the alphabet) of the computation of the $\mathcal{R}$-closure in alternation with conjugacy. Our result on the closure of APC under semi-commutation rewriting can be applied in modeling and verifying automatically parametrized networks having a ring topology, where information is exchanged between neighbors. Then, an interesting problem is to extend this work to similar systems with other kinds of topologies such as trees and grids.

References

[1] P. Abdulla, A. Bouajjani, and B. Jonsson. On-the-fly analysis of systems with unbounded,

lossy fifo channels. In *Proc. of CAV'98*, LNCS 1427, pp. 305–318, 1998.

[2] P. Abdulla, A. Bouajjani, B. Jonsson, and M. Nilsson. Handling global conditions in parametrized system verification. In *Proc. of CAV'99*, LNCS 1633, pp. 134–145, 1999.

[3] B. Boigelot and P. Godefroid. Symbolic verification of communication protocols with infinite state spaces using QDDs. In *Proc. of CAV'96*, LNCS 1102, pp. 1–12, 1996.

[4] A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model checking. In *Proc. of CONCUR'97*, LNCS 1243, pp. 135–150, 1997.

[5] A. Bouajjani, B. Jonsson, M. Nilsson, and T. Touili. Regular Model Checking. In *Proc. of CAV'00*, LNCS 1855, pp. 403–418, 2000.

[6] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.

[7] V. Diekert and G. Rozenberg, editors. *The Book of Traces*. World Scientific, Singapore, 1995.

[8] L. Fribourg and H. Olsén. Reachability sets of parametrized rings as regular languages. *Electronic Notes in Theoretical Computer Science*, pp. 1–12, 1997.

[9] P. Godefroid and P. Wolper. A partial approach to model checking. *Information and Computation*, 110(2):305–326, 1994.

[10] B. Jonsson and M. Nilsson. Transitive closures of regular relations for verifying infinite-state systems. In *Proc. of TACAS 2000*, LNCS 1785, pp. 220–234, 2000.

[11] Y. Kesten, O. Maler, M. Marcus, A. Pnueli, and E. Shahar. Symbolic model checking with rich assertional languages. In *Proc. of CAV'97*, LNCS 1254, pp. 424–435, 1997.

[12] A. Mazurkiewicz. Concurrent program schemes and their interpretations. DAIMI Rep. PB 78, Aarhus University, Aarhus, 1977.

[13] A. Muscholl and D. Peled. Message sequence graphs and decision problems on Mazurkiewicz traces. In *Proc. of MFCS'99*, LNCS 1672, pp. 81–91, 1999.

[14] Z. Manna and A. Pnueli. *The temporal logic of reactive and concurrent systems*. Springer, 1991.

[15] A. Muscholl, D. Peled, and Z. Su. Deciding properties of message sequence charts. In *Proc. of FoSSaCS'98*, LNCS 1378, pp. 226–242, 1998.

[16] A. Muscholl. Über die Erkennbarkeit unendlicher Spuren. Teubner Verlag, Stuttgart-Leipzig, 1996.

[17] D. Peled. All from one, one from all: on model checking using representatives. In *Proc. of CAV'93*, LNCS 697, pp. 409–423, 1993.

[18] J.-E. Pin and P. Weil. Polynomial closure and unambiguous product. *Theory of Computing Systems*, 30:383–422, 1997.

[19] A. Pnueli and E. Shahar. Liveness and acceleration in parametrized verification. In *Proc. of CAV'00*, LNCS 1855, pp. 328–343, 2000.

[20] D. A. Peled, T. Wilke, and P. Wolper. An algorithmic approach for checking closure properties of temporal logic specifications and omega-regular languages. *Theoretical Computer Science*, 195(2):183–203, 1998.

[21] W. Thomas. Classifying regular events in symbolic logic. *Journal of Computer and System Sciences*, 25:360–376, 1982.

[22] A. Valmari. A stubborn attack on state explosion. *Formal Methods in System Design*, 1:297–322, 1992.

[23] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc. of LICS '86*, pp. 332–344, 1986.

[24] P. Wolper and B. Boigelot. Verifying systems with infinite but regular state spaces. In *Proc. of CAV'98*, LNCS 1427, pp. 88–97,1998.