Permutation Rewriting and Algorithmic Verification *

Ahmed Bouajjani[1], Anca Muscholl[1], and Tayssir Touili[2]

[1] LIAFA, University of Paris 7, 2 place Jussieu, 75251 Paris cedex 5, France.
email: {abou, anca}@liafa.jussieu.fr

[2] School of Comput. Sci., Carnegie Mellon University, Pittsburgh, PA 15213, USA.
email: tayssirt@cs.cmu.edu

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 second order logics, temporal logics, etc) are extensively used for modeling and verifying properties of concurrent systems. The main reason is that regular languages enjoy important closure and decidability properties. They can model behaviors of systems in form of sets of computational sequences, often modulo some abstraction relation [7, 17, 30]. Furthermore, *regular model checking* has been proposed recently as a generic (automata/regular languages) based technique for symbolic verification of infinite state systems such as pushdown automata, fifo-channel systems, and parametrized networks of processes, see e.g. [2, 4–6, 13, 25, 31]. A fundamental problem which arises in these areas is then the following one: Given a regular language L and a relation $\mathcal{R}$ on sequences (represented either by a finite transducer or by a rewriting system), compute — if possible — the set $\mathcal{R}^*(L)$, i.e., the $\mathcal{R}$-closure of L ($\mathcal{R}^*$ denotes the reflexive-transitive closure of $\mathcal{R}$). Since unrestricted rewriting systems have the computational power of Turing machines, we must impose restrictions on the rewriting rules and on the regular languages we consider, in order to be able to compute the $\mathcal{R}$-closure. In this paper we focus on 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 [8]. However, semi-commutation rewriting does not preserve regularity in general. To see that, it suffices to consider the closure of the language $(ab)^*$ under the semi-commutation rule $ab \to ba$. Therefore,

* An earlier version of this article was presented at the 16th Annual IEEE Symposium on Logic in Computer Science, Boston (Ma), USA, June 2001.

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 related to formal modeling and verification of systems. For instance, partial-order reduction methods [11, 22, 29] 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 more convenient to express a property (or its negation) regardless of all possible interleavings of independent actions. Therefore, if a given property ϕ is not a partial-order property, then we are led to 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* [6, 13, 25], 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. In that case, 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 length (corresponding to systems with an arbitrary number of processes) are described by a regular language. This allows to perform 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 $ab \rightarrow ba$. Being able to compute the $\mathcal{R}$-closure thus allows to compute the effect of meta-transitions corresponding to the semi-commutation rewriting rules. Take as a very simple example a mutual exclusion protocol for linearly ordered processes. Suppose that the state of a process is 1 if it owns the token giving access to the critical section, 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 $\mathcal{R} = \{10 \rightarrow 01\}$, meaning that the token can be passed from left to right between neighbors. 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* $\mathcal{R}$, our aim is to compute the reflexive, transitive closure $\mathcal{R}^*(L)$. In our setting we would like to have a subclass of regular languages which enjoys several closure and decidability properties. First, we require that this class must be (1) effectively closed under basic operations such as union and intersection, and (2) effectively closed under semi-commutation rewriting. Moreover, we require that that the problems of (1) inclusion checking, and (2) membership to the class (of regular languages) are decidable. Indeed, all these features are needed in the context of the applications we consider, especially for regular

model checking where semi-commutation rewriting steps can be used iteratively during the reachability analysis of a system.

The solution proposed in this paper is the class of *Alphabetic Pattern Constraints* (APC), which appears naturally in many contexts of verification of concurrent systems. An APC is a finite union 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 passing example above are APC expressions. APCs can be used for example to express (negated) safety properties corresponding to the presence of (bad) 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 first-order logic with the order relation [27].

We show that this class satisfies all the closure properties stated above and establish complexity bounds for several basic decision problems concerning this class. 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 [2], for which it can be shown that inclusion can be checked in polynomial time.

The first main result of this paper 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 applications in 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 all its *conjugated* words $x_k x_{k+1} \cdots x_n x_1 \cdots x_{k-1}$ represent the same (circular) configuration. Thus, we consider that the set of configurations of a ring network is a set of words L which is closed under conjugacy, i.e. $L = \mathrm{Conj}(L)$. 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 set $\mathcal{R}^*(L)$ is computable. More precisely, we prove that $(\mathrm{Conj} \circ \mathcal{R}^*)^*(L) = (\mathrm{Conj} \circ \mathcal{R}^*)^{2|\Sigma|}(L)$. Therefore, for each APC language L the circular $\mathcal{R}$-closure is an effectively computable APC set.

Related work: The notion of semi-commutations was proposed in the late 70's by Mazurkiewicz [14] for describing the semantics of 1-safe Petri nets. Problems related to the closure of languages under semi-commutations have been studied in the community

of Mazurkiewicz trace theory (see e.g. chapter 12 in [8] for a survey). However, the problems we address here and our results are of different flavor.

Our aim is to identify subclasses of regular languages which are *closed* under *any* semi-commutation rewriting relation whereas classical results of trace theory aim at providing for a *given* semi-commutation relation $\mathcal{R}$ sufficient conditions on regular languages ensuring that their $\mathcal{R}$-closure remains regular. Such a condition was proposed first by Ochmanski [21] (see also [18]) and is known as "star-connectedness". This property restricts the sets of symbols labeling cycles in an automaton (resp., the iteration operator in a regular expression is restricted) and it depends on the relation $\mathcal{R}$. Since loops in APC are very simple (self-loops), one can easily see that APCs are star-connected. Nevertheless, our result is stronger in the sense that it shows that the closure of an APC *remains an APC*, whereas [21, 18] allows only to obtain that it is a regular language. As mentioned before, and as it will be shown in more details in Section 6, we do need such a stronger statement especially in the context of regular model checking where we must apply iteratively a sequence of rewriting steps including semi-commutation rewriting steps during the reachability analysis of a system. Moreover, our construction is different and has better complexity than the one described in [9]. Indeed, the proof of [9] is based on a notion of *rank* of a regular language, and on showing that if X is a regular language of finite rank, then the closure of X is regular. More precisely, if the rank of X is k, then Theorem 4.2 p. 481 in [9] leads to a deterministic finite automaton accepting the closure of X that has $2^{s^{k+1}}$ states, with s the size of a monoid recognizing X. The size s is *exponential* in the automaton or the regular expression given for X, as well as in the size of the alphabet, and the rank k is polynomial. In contrast, our algorithm gives a singly exponential size expression for the closure of the APC describing X (see Theorem 5, and Corollary 3). Furthermore, starting from an APC an applying the construction in [15], it is not possible to deduce that the resulting language is an APC, since the reasoning based on the rank leads only to a bound on the number of states of an automaton, and not to any statement about the structure of the automaton. So, our algorithm is not a simple instantiation of known results, as described in [8]. Let us also mention that the complexity of *deciding* whether a regular ω-language is closed under commutation rewriting was considered in [20, 26].

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 and uses as atomic predicates the order and the letter labeling. 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 [23].

Finally, several works on regular model checking propose automata-based constructions for computing the closure of a regular language under a rewriting system (regular relations/finite transducers) [3, 6, 10, 12, 25]. While the proposed techniques are generic and have a wide range of application, they are not complete in general. In contrast, our work proposes a complete construction for the specific class of semi-commutation systems and the class of APC languages. As it will be shown in Section 6, this con-

struction could be integrated with the other generic construction in order to deal with the semi-commutation rules which may appear in the definition of a system.

2 Alphabetic Pattern Constraints

We define in this section the class of *Alphabetic Pattern Constraints (APC)* and present some of its characterizations.

2.1 Definitions and notations

Definition 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 $a_1, a_2, \ldots, a_n \in \Sigma$.*

A product p over Σ^* is a (possibly empty) concatenation $e_1 e_2 \cdots e_n$ 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(Σ) 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 4, 5 will be an APC expression.

Notation. The length of $p = e_1 \cdots e_n$, 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 $L \subseteq \Sigma^*$, we denote by $\alpha(L)$ the set of letters of Σ appearing in words from 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.

2.2 Characterizations of APCs

APC corresponds to he set of expressions defining the languages of level 3/2 of the Straubing-Thérien hierarchy [23]. In this hierarchy, level 0 corresponds to $\emptyset, \Sigma^*$ and the higher levels are defined recursively as follows: level $n + 1/2$ is the polynomial closure of level n and level $n + 1$ is the boolean closure of level $n + 1/2$, where the polynomial closure of a class $\mathcal{L}$ of languages is the set of languages of Σ^* that are finite union of languages of the form $L_0 a_0 L_1 \cdots a_n L_n$ where the a_i's are letters and the L_i's are elements of $\mathcal{L}$. It is well-known that the Straubing-Thérien hierarchy is strict.

Closely related is the well known Σ_n-hierarchy of first order logic (over the order relation) obtained by counting the number of alternating blocks of existential and universal quantifiers. The level Σ_n denotes formulas with n alternating blocks of quantifiers, starting with an existantial block. Thomas [27] showed that the level $n+1/2$ in the Straubing-Thérien hierarchy corresponds precisely to Σ_{n+1}. Thus, it follows that each language in APC can be defined by a formula of the form $\phi = \exists^* \forall^* \psi$ where ψ is quantifier free.

Furthermore, it can be easily shown that APC corresponds to the following fragment of the linear-time temporal logic LTL [24]:

$$\phi ::= A \mid \phi \vee \phi \mid \phi \wedge \phi \mid A \mathcal{U} \phi \mid \Box A \mid \bigcirc \phi$$

where A is of the form $A = a_1 \vee \cdots \vee a_n$. Indeed, every APC can be described by a formula in this fragment since a product of the form $\Sigma_0^* a_0 \Sigma_1^* a_1 \cdots a_{n-1} \Sigma_n^*$ is equivalent to the formula

$$\Sigma_0 \mathcal{U} (a_0 \wedge \bigcirc (\Sigma_1 \mathcal{U} (a_1 \wedge \bigcirc \cdots (\Sigma_{n-1} \mathcal{U} (a_{n-1} \wedge \bigcirc \Box \Sigma_n)) \cdots)$$

For the other direction, we use the fact that APC are closed under union, intersection, and concatenation (see Proposition 1).

3 Closure properties and decision problems

We consider in this section closure properties of APC under basic operations such as boolean operations and conjugacy, and address decision proplems (and their complexity) such as the test of inclusion, closure under a given semi-commutation relation, and membership of regular language to APC. These closure properties and decision problems are particularly relevant for the applications we consider in this paper, that is, verification of partial-order models and regular model checking.

3.1 Closure properties

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

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 $\{uv \in \Sigma^* : vu \in L\}$ 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 $\mathrm{Conj}(L) \in \mathcal{C}$.

Proposition 1. *APC is closed under product, union, intersection and conjugacy.*

Proof. APC is closed under product and union by definition. Let p_1 and p_2 be two products over Σ. Then, the regular expression corresponding to $p_1 \cap p_2$ is inductively defined by:

- $\emptyset$, if $p_1 = \emptyset$ or $p_2 = \emptyset$,
- ε, if $p_1 = \varepsilon$ and $\varepsilon \in p_2$, or $p_2 = \varepsilon$ and $\varepsilon \in p_1$,
- $a(p_1' \cap p_2')$, if $p_1 = a p_1'$ and $p_2 = a p_2'$,
- $\emptyset$, if $p_1 = a p_1'$, $p_2 = b p_2'$, and $a \neq b$,
- $a(p_1' \cap p_2)$, if $p_1 = a p_1'$ and $p_2 = A^* p_2'$ with $a \in A$,
- $b(p_1 \cap p_2')$, if $p_1 = B^* p_1'$ and $p_2 = b p_2'$ with $b \in B$,
- $C^*(p_1 \cap p_2') + C^*(p_1' \cap p_2)$, if $p_1 = A^* p_1'$, $p_2 = B^* p_2'$, and $C = A \cap B \neq \emptyset$.

To show that APC is closed under conjugacy, it suffices to show that for any product p, the set of conjugates $Conj(p)$ is an APC. Let $p = e_1 \cdots e_n$ be a product, then it is easy to see that :

$$Conj(p) = \sum_i e_i \cdots e_n e_1 \cdots e_{i-1} e'_i$$

such that (1) $e'_i = e_i$ if e_i is a star expression, and (2) $e'_i = \epsilon$ if e_i is a letter. Since the expression above is an APC, the result follows. □

Remark 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 $\{w \in (a+b)^* \ : \ |w|_a = |w|_b = n\} = p_n \cap q_n$ is a finite set with the property that every APC expression for $p_n \cap q_n$ is of exponential size.

3.2 Decision properties and complexity issues

We consider in this section basic decision problems concerning the class APC. We establish the complexity for the problems of testing inclusion, deciding whether an APC language is closed under a given semi-commutation relation, and deciding whether a given regular language is in the APC class. Basically, we show that all these problems are PSPACE-complete (which means that they are already as complex as for the case of the whole class of regular languages), except for the last problem in the case where the considered regular language is given as a deterministic automaton. We show that the problem is NLOGSPACE-complete in that case.

Let us start with the universality problem.

Theorem 1. *The following problem is PSPACE-complete:*
Input: *An APC expression L over Σ^*.*
Question: *Is $L = \Sigma^*$?*

Proof. We already know that the universality problem for languages in APC(Σ) is in PSPACE since it is in PSPACE for regular languages. Thus we need only to show the PSPACE-hardness.

Let M be a P-space-bounded, single-tape deterministic Turing machine that accepts L (where P is some polynomial). Let Q be the set of states of M, let Σ be its tape alphabet, and let q_0, q_f its initial and final states respectively. The blank symbol is denoted by b.

The reduction is classical, iconsists of showing that for every input x we can construct in polynomial time a language R_x in APC(Σ) such that $\overline{R_x} = \emptyset$ if and only if M does not accept x. Let x be such an input to M of length n. The language R_x that we construct is the language representing all the sequences of configurations of M that do not correspond to accepting runs of M on x.

Let $p = P(n)$. We impose that all the configurations of M are of length p (we can add blanks to a configuration if necessary to make it of length p). Thus, a configuration

of M can be represented by a string $a_1 a_2 \cdots a_p$ where all symbols a_i are letters from Σ, except for one, which is in $Q \times \Sigma$. This symbol is of the form $[qX]$ and corresponds to the cell of the tape which is scanned by the head of the machine and holds the letter X, while M is in the control state q.

A run of M can be represented by a string $\sharp w_1 \sharp w_2 \cdots w_k \sharp$ for some $k \geq 1$, where w_i is an encoding of the configuration C_i of M after $i-1$ moves. The encodings w_i have the following form: $w_i = a_1 \theta \beta a_2 \theta \beta^2 \cdots a_p \theta \beta^p$, where $a_1 a_2 \cdots a_p$ is the configuration C_i. The additional counters β^i used in our proof are needed in order to obtain an expression of the required form *and* of polynomial size.

Let $\Delta = \Sigma \cup \{[qX] \ : \ q \in Q \text{ et } X \in \Sigma\}$ and $\Delta' = \Delta \cup \{\theta, \beta\}$. A string y in $(\Delta' \cup \{\sharp\})^*$ represents a non-accepting computation of M if and only if at least one of the following conditions holds:

1. y is not of the form $\sharp w_1 \sharp w_2 \cdots w_k \sharp$ for some $k \geq 1$, $w_i \in \Delta'^*$,
2. The initial configuration w_1 is wrong. That is, y does not begin with $\sharp[q_0 c_1]\theta\beta c_2 \theta \beta^2 \cdots c_n \theta \beta^n b \theta \beta^{n+1} b \theta \beta^{n+2} \cdots b \theta \beta^p \sharp$, where $x = c_1 \cdots c_n$,
3. The last configuration is not final. That is y does not end with the encoding of a final configuration,
4. y contains two consecutive configurations which do not respect the transition relation of M.

We define the expression R_x as the union of four expressions A, B, C and D corresponding respectively to conditions 1, 2, 3, and 4 given above. The expression A contains:

1. strings not beginning or not ending with $\sharp$,
2. strings with no or more than one symbol of $Q \times \Sigma$ between two $\sharp$'s,
3. strings having more than one letter of Σ between two consecutive β and θ,
4. strings not having the right syntax between two consecutive $\sharp$'s.

It is easy to see that the first conditions can be described by a language in APC(Σ). We only consider the last condition. The strings meeting this condition are described by the following APC expression of length $O(p^2 + |M|)$:

$$\sum_{0 \leq i,j \leq p, j \neq i+1} (\Delta' + \sharp)^* \Delta \theta \beta^i \Delta \theta \beta^j (\Delta' + \sharp)^* +$$
$$\sum (\Delta' + \sharp)^* \beta^{p+1} (\Delta' + \sharp)^* +$$
$$\sum_{k<p} (\Delta' + \sharp)^* \beta^k \sharp (\Delta' + \sharp)^* +$$
$$\sum_{k>1} (\Delta' + \sharp)^* \sharp \Delta \theta \beta^k (\Delta' + \sharp)^*$$

The languages B and C can be constructed in the same way. The language D is defined by the expression:

$$\sum_{a,b,c \not\to b'} (\Delta' + \sharp)^* a \theta \beta^{i-1} b \theta \beta^i c \Delta'^* \sharp \Delta'^* b' \theta \beta^i (\Delta' + \sharp)^*$$

This expression says that if we are in a configuration where the $i-1^{th}$, the i^{th} and the $i+1^{th}$ cells are respectively represented by the symbols a, b and c, then, after one move, the Turing machine must be in a configuration where the symbol corresponding to the i^{th} cell is uniquely determined and is equal to b_1. The expression above considers the computations that have two consecutive configurations $w_j = \cdots a\theta\beta^{i-1}b\theta\beta^i c\theta\beta^{i+1}\cdots$ and $w_{j+1} = \cdots b'\theta\beta^i\cdots$ such that the letter b' is different from b_1.

Thus, the expression R_x is in APC(Σ) and can be constructed in time $O(p^2+|M|)$. Moreover, we have that R_x equals $(\Delta' + \sharp)^*$ if and only if x is not accepted by M.

□

An immediate consequence of the previous result is the following fact:

Corollary 1. *Deciding inclusion for languages in APC is PSPACE-complete.*

Let us now consider the problem of deciding whether a language is closed under a given semi-commutation rewriting system.

Theorem 2. *The following problem is PSPACE-complete:*
Input: *An APC expression L over Σ and a semi-commutation rewriting system $\mathcal{R}$*
Question: *Does $\mathcal{R}^*(L) = L$ hold?*

Proof. The fact that this problem is in PSPACE is immediate, it suffices to check that $\mathcal{R}(L) \subseteq L$. Since the inclusion test is in PSPACE, we can decide in PSPACE whether L is closed under $\mathcal{R}$ or not.

To show hardness, we reduce the universality problem to the above closure problem.

Let L be a language in APC and $\mathcal{R}$ the semi-commutation relation $\mathcal{R} = \{(a, \$) : a \in \Sigma\}$. Consider the APC language $K = L\$\Sigma^* \cup \Sigma^*\$$. We show that K is closed under $\mathcal{R}$ if and only if $L = \Sigma^*$.

By definition, K is closed under $\mathcal{R}$ if and only if $K = \mathcal{R}^*(K)$. It is easy to see that $\mathcal{R}^*(K) = \{u\$vw \ : \ uv \in L\} \cup \Sigma^*\$\Sigma^* = \Sigma^*\$\Sigma^*$. Thus, we have $K = \mathcal{R}^*(K)$ if and only if $L\$\Sigma^* \cup \Sigma^*\$ = \Sigma^*\$\Sigma^*$, if and only if $L = \Sigma^*$. Thus, K is closed under $\mathcal{R}$ if and only if $L = \Sigma^*$, which ends the proof. □

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 [23]. It is worth noting that the algorithm obtained in [23] 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 4 below improves the result by giving an algorithm which is polynomial in both $|A|$ and $|\Sigma|$.

Theorem 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.*

Proof. Let us show the containment in PSPACE. Let L be a regular language given by a non-deterministic automaton A. Let Δ be its transition relation. The characterization given in [23] states that L is in APC if and only if for all words x, y with $\alpha(x) = \alpha(y)$ such that x satisfies the following:

$$\forall z, w \in \Sigma^* : zxw \in L \text{ iff } zxxw \in L$$

we have the following implication:

$$\forall u, v \in \Sigma^* : uxv \in L \Rightarrow uxyxv \in L$$

We now describe how to check the negation of the implication above in PSPACE. A Turing machine M guesses the word u symbol by symbol (on-the-fly). M uses an array to keep track of the set of states which A reaches after reading u. Let S_0 be the set of initial states and let T_1 be the set of states reached after reading u, then we have $\Delta(S_0, u) = T_1$. Then, M guesses sets of states T_2, T_3 and T_4 and two strings x and y on-the-fly, verifying that $\alpha(x) = \alpha(y)$, $\Delta(T_1, x) = T_2$, $\Delta(T_2, y) = T_3$ and $\Delta(T_3, x) = T_4$. Moreover, M verifies during the guess of x that $zxw \in L$ iff $zxxw \in L$, for all strings z, w (it's easy to see how to verify the negation of this condition in PSPACE, then we use the fact that PSPACE is closed under complement).

Finally, M guesses v such that $\Delta(T_2, v) \cap F \neq \emptyset$ and $\Delta(T_4, v) \cap F = \emptyset$. Hence, we have that $uxv \in L$ and $uxyxv \notin L$.

The PSPACE-hardness is shown using the proof given for Theorem 1. Let M be a polynomial-space-bounded Turing machine and x an input of M. Let $R_{M,x}$ be the APC expression computed in the proof of Theorem 1. We have $R_{M,x} = \Sigma^*$ if and only if M does not accept x. Consider now the set $K = (R_{M,x}\$)^*$ with $\$ \notin \Sigma$. We show that M does not accept x if and only if K is in APC.

- If M does not accept x, then by definition of $R_{M,x}$ we have that $R_{M,x} = \Sigma^*$ which implies that $K = (\Sigma \cup \{\$\})^*$. Thus, K is in APC.
- If M accepts x, then there exists some word $y \notin R_{M,x}$ encoding an accepting computation of M on x. Let z be some word with $\alpha(y) = \alpha(z)$ and such that $z \in R_{M,x}$ does not encode an accepting computation.
  We show by contradiction that K does not belong to APC. Suppose that K is in APC, then let $p_1, p_2, \ldots, p_q$ be products such that $K = p_1 + p_2 + \cdots + p_q$ and let $n = \max_i l(p_i)$.
  Let $w = (z\$)^{n+1} \in K$, then there exists some product p_j such that $w \in p_j = A_0^* a_1 A_1^* a_2 \cdots a_m A_m^*$, $m \leq n$. Since $n + 1 > m$, there exists some star expression A_k^* in p such that some occurrence of the factor $z\$$ of w lies completely within A_k^*. Hence, since $\alpha(z\$) = \alpha(y\$) \subseteq A_k$ we obtain that $z\$z\$ \cdots z\$y\$z\$ \cdots z\$ \in K = (R_{M,x}\$)^*$. This contradicts the fact that $y \notin R_{M,x}$. Therefore, K cannot be in APC.

□

Theorem 4. *Deciding whether a regular language, given by a deterministic automaton, is an APC language, is an NLOGSPACE-complete problem.*

Proof. We use a second characterization for APC languages given in [23]. As shown there (see Theorem 8.9) a deterministic, complete automaton A accepts an APC language if and only if for all words $u, v \in \Sigma^*$ and all states q_1, q_2, q_3, q_4 of A satisfying the following conditions:

- u is a loop around q_1 and q_2, i.e., $q_1 \in \delta(q_1, u)$, $q_2 \in \delta(q_2, u)$,
- $q_2 \in \delta(q_1, v)$,
- $\alpha(v) = \alpha(u)$,
- $q_3 \in \delta(q_1, w)$ and $q_4 \in \delta(q_2, w)$
- q_1 is reachable from an initial state,

we have that if q_3 is a final state, then q_4 is also a final state.[1] This situation is depicted in Figure 1.

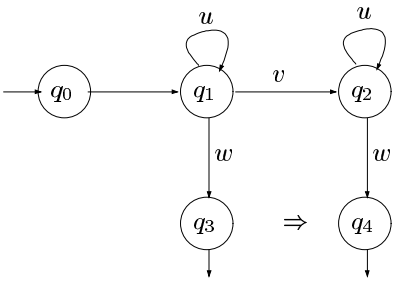

Fig. 1. Characterization of DFA of an APC language.

We cannot check the above conditions directly in NLOGSPACE. However, note that we can replace the condition $\alpha(u) = \alpha(v)$ by $\alpha(v) \subseteq \alpha(u)$ (since we can use instead of u, v the words u, uv). With this modification we can guess u, v (and w) without storing them and verify on-the-fly that $\alpha(v) \subseteq \alpha(u)$.

For the NLOGSPACE-hardness we reduce from GAP, the question whether there exists a path from a vertex s to a vertex t in a directed graph $G = (V, E)$. Without restriction we assume that G is acyclic and has out-degree 2 for each vertex $v \neq t$ (and t has out-degree 0). Let $\Sigma = \{a, b, c, d\}$ and let us define a deterministic, complete automaton A over Σ as follows. The set of states of A will be $Q = \{p, q, r\} \cup V$, where p, r are the only final states. Moreover, p is the initial state. Each state from $V \setminus \{t\}$ has its two outgoing edges from E, labeled deterministically by a, b. The initial state p has two self-loops labeled a, b, an edge to s, labeled d, and an edge to r, labeled c. State q has two self-loops labeled a, b, an edge to t, labeled d, and an edge to r, labeled c. State r is a sink state, i.e., with self-loops labeled a, b, c, d. State s has a c-transition to p, and state t has a c-transition to q. Finally, we complete the automaton by adding edges to r for all vertices in V which do not yet have 4 outgoing edges (see Figure 2).

Note that the labels of loops around s and t must be labeled by words from $(c(a+b)^*d)^*$, whereas loops around p, q must be labeled by words from $((a+b)^*dc)^*(a+b)^*$.

[1] [23] states that this condition is necessary, provided that the automaton A is minimal. However, the given proof does not use the minimality of A.

The only interesting combinations for (q_1, q_2) as in Figure 1 are (s, t) and (p, q) (the case $q_2 = r$ never violates the conditions in Figure 1). The last two pairs are possible if and only if there is a path from s to t in A. In this case, both pairs violate the condition on final states: From s there is a c-transition to the final state p, whereas t has a c-transition to the non-final state q (we take $w = c$). This concludes the proof.

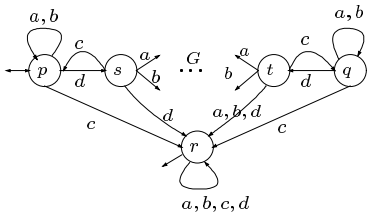

Fig. 2. Reduction from GAP

□

4 Semi-Commutation Rewriting and APC

We address now the problem of closing an APC under semi-commutation rewriting. A semi-commutation relation $\mathcal{R}$ defined over an alphabet Σ is an irreflexive binary relation, i.e., a subset of $\Sigma \times \Sigma \setminus \{(a, a) \ : \ a \in \Sigma\}$. A pair (a, b) in $\mathcal{R}$ can also be represented by the rule $ab \to ba$. 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 $(a, b) \in \mathcal{R}$, $w = w_1 abw_2$, and $w' = w_1 baw_2$. 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* $\{v \in \Sigma^* \ : \ \exists u \in L, (u, v) \in \rho^*_{\mathcal{R}}\}$ 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

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

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

It is not difficult to see that semi-commutation rewriting does not preserve regularity. 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, the relation $\mathcal{R}^*$ cannot be represented by a finite transducer, in general.

Our main result is stated in the theorem below. The remaining of this section consists in describing the algorithm underlying Theorem 5.

Theorem 5. *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 $\mathcal{O}(l(L)\delta_{\mathcal{R}})$.*

Since any $L \in \mathrm{APC}(\Sigma)$ 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 $\mathcal{R}^*(e_1) = e_1$. 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 some notations and basic definitions:

Definition 2. *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 $x_1y_1 \cdots x_ny_n$ with $x = x_1 \cdots x_n$, $y = y_1 \cdots y_n$, $x_i, y_i \in \Sigma^*$ for all $1 \leq i \leq n$ and such that $(\alpha(x_i), \alpha(y_j)) \in \mathcal{R}$ for all $j < i$.*

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

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

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

Lemma 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 $\mathcal{R}^*(LK) = L \sqcup\!\sqcup_{\mathcal{R}} K$.*

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

Lemma 2. *Let $e_1, e_2, \ldots, e_n$ be atomic expressions and let $p = e_1e_2 \cdots e_n$ be a product, then we have:*

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

By the preceding lemma we can compute $\mathcal{R}^*(p)$ recursively. Lemma 3 and Proposition 2 below are the basic cases of our algorithm.

Lemma 3. *Let E be a subset of Σ and let $a \in \Sigma$ be a letter, then we have:*

$$E^* \sqcup\!\sqcup_{\mathcal{R}} a = \mathcal{R}^*(E^*a) = E^*aE'^*, a \sqcup\!\sqcup_{\mathcal{R}} E^* = \mathcal{R}^*(aE^*) = E''^*aE^*,$$

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

Proof. The first equality can be inferred from Lemma 1, since E^* and a are closed by $\mathcal{R}$. The second equality is straightforward since the symbols of E' are precisely the letters of E that are able to cross the letter a.

□

Example 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

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

The next proposition is the main technical result needed for the proof of Theorem 5. It shows that the $\mathcal{R}$-closure of the product of two star expressions is an APC. In particular, note that the length of the products in the expression given below is bounded above by a constant n which is polynomial in $|\Sigma|$ and $|\mathcal{R}|$. More precisely, $n \leq \delta_{\mathcal{R}} \leq |\Sigma|$.

Proposition 2. *Let E and F be two subsets of Σ, then*

$$E^* \sqcup\!\sqcup_{\mathcal{R}} F^* = \mathcal{R}^*(E^*F^*) = \sum E^*(E_1+F_1)^* \cdots (E_n+F_n)^* F^*,$$

where $n \leq \delta_{\mathcal{R}}$, and the sum is taken over all subsets E_i and F_i of Σ satisfying the following conditions:

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

Proof. The first equality can be inferred as previously from Lemma 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 $w \in E^* \sqcup\!\sqcup_{\mathcal{R}} F^* = \mathcal{R}^*(E^*F^*)$. 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 $j \neq m$.

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

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

By definition we have $E_{i+1} \subsetneq E_i \subseteq E$, and $F_i \subsetneq F_{i+1} \subseteq F$ 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 $j \leq i$. These two facts imply of course that $n \leq \delta_R$. 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 2. In the expression given for $\mathcal{R}^*(E^*F^*)$, it suffices to list only the star expressions $(E_i+F_i)^*$ such that both E_i and F_i are maximal, in the sense that:

- there is no letter $b \in F \backslash F_i$ such that $(E_i, b) \in \mathcal{R}$,
- there is no letter $a \in E_{i-1} \backslash E_i$ such that $(a, F_i) \in \mathcal{R}$, where $E_0 = E$.

Remark 3. Note that the length of the products in the expression for $\mathcal{R}^*(E^*F^*)$ is at most $\delta_{\mathcal{R}} + 2$.

Example 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 $\mathcal{R}_2^*(p) = (a+b+c)^* \sqcup\!\sqcup_{\mathcal{R}_2} (e+f+g)^* = (a+b+c)^*(c+g)^*(e+f+g)^* + (a+b+c)^*(a+b+e)^*(b+e+f)^*(e+f+g)^*$.

We show now how to compute effectively $\mathcal{R}^*(p) = \mathcal{R}^*(e_1 e_2 \cdots e_n)$ and obtain that it is an APC. By Lemma 3 and Proposition 2 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 $\sum e_1 \sqcup\!\sqcup_{\mathcal{R}} (f_1 f_2 \cdots f_k)$, also belongs to APC. Thus, we only need to compute $e_1 \sqcup\!\sqcup_{\mathcal{R}} (f_1 f_2 \cdots f_n)$ 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 4. *Let $a \in \Sigma$ and $f_1, \ldots, f_n$ be atomic expressions, then*

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

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

- *if $f_i = E^*$, then $g_i = \{x \in E \ : \ (a,x) \in \mathcal{R}\}^*$,*
- *if $f_i = b \in \Sigma$ and $(a,b) \in \mathcal{R}$, then $g_i = b$.*

Moreover, $h_j = \varepsilon$ when $f_j \in \Sigma$, and $h_j = f_j$, otherwise.

Example 3. Let $\mathcal{R}_3$ be the semi-commutation relation $\mathcal{R}_3 = \{(h,a),(h,e)\}$. Then the previous lemma implies that $h \sqcup\!\sqcup_{\mathcal{R}_3} (a+b+c)^*(a+b+e)^*(b+e+f)^* = a^* h (a+b+c)^*(a+b+e)^*(b+e+f)^* + (a+e)^* h (a+b+e)^*(b+e+f)^*$.

The next proposition generalizes Lemma 3 and Proposition 2.

Proposition 3. *Let E and F be two subsets of Σ, $a \in \Sigma$, and $L \subseteq \Sigma^*$, then we have:*

1. $E^* \sqcup\!\sqcup_{\mathcal{R}} (aL) = (E^* \sqcup\!\sqcup_{\mathcal{R}} a)(E'^* \sqcup\!\sqcup_{\mathcal{R}} L)$, *where* $E' = \{b \in E \ : \ (b,a) \in \mathcal{R}\}$.
2. $E^* \sqcup\!\sqcup_{\mathcal{R}} (F^* L) = \sum_{\substack{(E',F') \in \mathcal{R} \\ E' \subseteq E, F' \subseteq F}} (E^* \sqcup\!\sqcup_{\mathcal{R}} F'^*)(E'^* \sqcup\!\sqcup_{\mathcal{R}} L)$.

Proof. The first identity is straightforward, since E' are precisely the letters of E that can cross a and then commute with L.

We show now the second identity. The inclusion from right to left is clear.

Conversely, let $w \in E^* \sqcup\!\sqcup_R F^* L$, with $w \in x \sqcup\!\sqcup_R yz$ such that $x \in E^*, y \in F^*$ and $z \in L$. Assume that $w = uv$ and $x = x_1 x_2$ with $u \in x_1 \sqcup\!\sqcup_R y, v \in x_2 \sqcup\!\sqcup_R z$. Let $F' = \alpha(y)$ and $E' = \alpha(x_2)$, then obviously $(E', F') \in R$, which ends the proof. □

Corollary 2. *Let E and F be two subsets of Σ, and let $L \subseteq \Sigma^*$, then*

$$E^* \sqcup\!\sqcup_{\mathcal{R}} (F^* L) = \sum E^*(E_1 + F_1)^*(E_2 + F_2)^* \cdots (E_k + F_k)^*(E_k^* \sqcup\!\sqcup_{\mathcal{R}} L)\,,$$

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 it remains to show that

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

where $E' \subseteq E$ and $F' \subseteq F$ are subsets satisfying $(E', F') \in \mathcal{R}$. Proposition 2 implies that:

$$(E^* \sqcup\!\sqcup_{\mathcal{R}} F'^*)(E'^* \sqcup\!\sqcup_{\mathcal{R}} L) = \sum E^*(E_1 + F_1')^* \cdots (E_n + F_n')^* F'^*(E'^* \sqcup\!\sqcup_{\mathcal{R}} L)$$

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

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

It remains then to show that for such subsets E_i, F_j', there exist two sequences $(\widetilde{E_i})_i$ and $(\widetilde{F_i})_i$ satisfying the above conditions such that:

$$E^*(E_1 + F_1')^* \cdots (E_n + F_n')^* F'^*(E'^* \sqcup\!\sqcup_{\mathcal{R}} L) \subseteq E^*(\widetilde{E_1} + \widetilde{F_1})^* \cdots (\widetilde{E_k} + \widetilde{F_k})^*(\widetilde{E_k}^* \sqcup\!\sqcup_{\mathcal{R}} L)$$

These sequences can be defined inductively as follows:

– $\widetilde{E_1} = E_1 + E'$, and $\widetilde{F_1} = F_j'$, where $j = \max\{i \,:\, (E_i + E') = (E_1 + E')\}$,
– If $\widetilde{F_i} = F_l'$ then:
  • If $l < n$, then $\widetilde{E_{i+1}} = E_{l+1} + E'$, and $\widetilde{F_{i+1}} = F_j'$, where $j = \max_i\{l+1 \leq i \leq n \,:\, (E_i + E') = (E_{l+1} + E')\}$,
  • If $l = n$ and $F' \neq F_n'$, $\widetilde{E_{i+1}} = E'$, and $\widetilde{F_{i+1}} = F'$.

Then obviously, $\widetilde{E_k} \subsetneq \cdots \subsetneq \widetilde{E_1} \subseteq E$, $\emptyset \neq \widetilde{F_1} \subsetneq \cdots \subsetneq \widetilde{F_k} \subseteq F'$, and $(\widetilde{E_i}, \widetilde{F_j}) \in \mathcal{R}$ for all $1 \leq j \leq i \leq k$ since $(E', F') \in \mathcal{R}$ and $(E_i, F_j') \in \mathcal{R}$ for all $1 \leq j \leq i \leq n$.

□

Example 4. Let $\mathcal{R}_4 = \{(a,e),(c,g),(b,e),(b,f),(a,d)\}$. Then from the last proposition and from example 2 it follows that
$(a+b+c)^* \sqcup\!\sqcup_{\mathcal{R}_4} (e+f+g)^* d (f+g)^* = (a+b+c)^*(c+g)^*(e+f+g)^* d(f+g)^* +$
$(a+b+c)^*(a+b+e)^*(b+e+f)^*(e+f+g)^* d(f+g)^* + (a+b+c)^*(a+b+e)^* da^*(f+g)^*$.

Summarizing the previous computations, Proposition 3 and Corollary 2 yield the recursive step for computing $E^* \sqcup\!\sqcup_{\mathcal{R}} (f_1 f_2 \cdots f_n)$:

Proposition 4. *Let $E \subseteq \Sigma$ and let $f_1, \ldots, f_n$ be atomic expressions. Then, $E^* \sqcup\!\sqcup_{\mathcal{R}} (f_1 f_2 \cdots f_n)$ is equal to one of the following expressions:*

1. *For a star expression* $f_1 = F^*$*:*

$$\sum E^*(E_1 + F_1)^* \cdots (E_k + F_k)^*(E_k^* \sqcup\!\sqcup_{\mathcal{R}} f_2 \cdots f_n),$$

where the union is taken over all subsets E_i, F_i *satisfying* $E_{i+1} \subsetneq E_i \subseteq E$, $\emptyset \neq F_i \subsetneq F_{i+1} \subseteq F$ *and* $(E_i, F_j) \in \mathcal{R}$ *for all* $j \leq i$.

2. *For a single letter* $f_1 = a$*:*

$$E^* a(E'^* \sqcup\!\sqcup_{\mathcal{R}} f_2 \cdots f_n),$$

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

The algorithm for computing the closure of an APC expression $\sum e_1 \cdots e_n$ under a semi-commutation rewriting relation $\mathcal{R}$ is the following: We compute recursively $\mathcal{R}^*(e_2 \cdots e_n) = \sum f_1 \cdots f_k$. The recursive step is given by Lemma 4, if e_1 is a letter. Otherwise, for $e_1 = E^*$ we apply Proposition 4, which is itself a recursive step. It is easily seen that each step preserves containment in APC. This shows Theorem 5.

Corollary 3. *Let* p *be a product of length* n*, then* $l(\mathcal{R}^*(p)) \in \mathcal{O}(n\delta_{\mathcal{R}})$*, and* $size(\mathcal{R}^*(p)) \in 2^{O(|\Sigma| n\delta_{\mathcal{R}})}$.

Proof. It can be easily seen that each recursive step adds at most $\delta_{\mathcal{R}}$ atomic expressions. Since there are n recursive steps, then the length of the products of the expression computed for $\mathcal{R}^*(p)$ is at most $\mathcal{O}(n\delta_{\mathcal{R}})$. Moreover, since there exist $2^{|\Sigma|} + |\Sigma|$ different atomic expressions, it follows that the size of $\mathcal{R}^*(p)$ is at most $2^{O(|\Sigma| n\delta_{\mathcal{R}})}$. □

We conclude this section with an example that illustrates the computation of the closure of a product under a semi-commutation relation.

Example 5. Consider the product $p = h(a + b + c)^*(e + f + g)^* d$, and the semi-commutation relation $\mathcal{R} = \{(a, e), (h, a), (h, e), (c, g), (b, e), (b, f), (f, d), (g, d), (a, d)\}$. Then $\mathcal{R}^*(p) = h \sqcup\!\sqcup_{\mathcal{R}} ((a + b + c)^* \sqcup\!\sqcup_{\mathcal{R}} ((e + f + g)^* \sqcup\!\sqcup_{\mathcal{R}} d))$. Hence using the previous examples we have

$$\begin{aligned}
&\mathcal{R}^*(p) = a^* h(a + b + c)^*(c + g)^*(e + f + g)^* d(f + g)^* + \\
&a^* h(a + b + c)^*(a + b + e)^*(b + e + f)^*(e + f + g)^* d(f + g)^* + \\
&a^* h(a + b + c)^*(a + b + e)^* da^*(f + g)^* + \\
&(a + e)^* h(e + f + g)^* d(f + g)^* + \\
&(a + e)^* h(a + b + e)^*(b + e + f)^*(e + f + g)^* d(f + g)^* + \\
&(a + e)^* h(a + b + e)^* da^*(f + g)^*.
\end{aligned}$$

Remark 4. We observe that APC is the largest level in both Straubing-Thérien's hierarchy and the Σ_n-hierarchy that is closed under semi-commutations. Indeed, $(ab)^*$ is the complement of a language in APC. Thus, it belongs to level 2 of Straubing-Thérien's hierarchy and it can be described by a Σ_3-formula. The closure of $(ab)^*$ by the semi-commutation relation $\{(a, b), (b, a)\}$ is of course not regular (it equals the set $\{w \in (a + b)^* \ : \ |w|_a = |w|_b\}$).

Moreover, let us note that APC is also the largest level of both hierarchies with the following property: if the closure under semi-commutation of a language is regular, then it belongs to the same level. Take as an example $L = (abcbac)^*$ and $\mathcal{R} = \{(a, b), (b, a)\}$. Then $\mathcal{R}^*(L) = ((ab + ba)c)^*$ is regular, but not star-free anymore.

5 Circular Rewriting

We consider in this section 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 $\mathrm{Conj}(L) = \{vu \ : \ uv \in L\}$ 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:

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

We show 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 6. *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 4. *Let L be a APC expression, then $\mathcal{R}_c^*(L)$ is in APC and is effectively computable.*

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 the fact that $\mathrm{APC}(\Sigma)$ is closed under semi-commutation rewriting and conjugacy (Theorem 5 and Proposition 1). □

In the remaining of the section we prove Theorem 6. The proof uses ideas from [8][Ch. 3]. It generalizes (and simplifies) the proof given there for the case where $\mathcal{R}$ is a symmetric relation. As in [8] 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. *$\mathcal{R}_c \subseteq \mathcal{C}_\mathcal{R}$ 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 $(x,z) \in \mathcal{C}_\mathcal{R}$. □

The theorem below is used in showing a kind of converse of Lemma 5:

Theorem 7. *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. Let $x, y, z \in \Sigma^*$ be such that $zy \in R^*(xz)$. We show that there exist $m \leq 2|\Sigma|$, $t_0, \ldots, t_m \in \Sigma^*$ satisfying the properties above and such that $\alpha(t_0 \cdots t_{m-1}) \subseteq \alpha(z)$. We proceed by induction on $|x| + |z|$. If $|x| + |z| = 1$ then $x = y = a$ for some letter $a \in \Sigma$ and $z = \epsilon$. We may thus assume that $|x| + |z| > 1$. Levi's lemma for semi-traces [8][Ch. 12] implies that there exist words $u, v, p, q \in \Sigma^*$ such that:

- $up \in R^*(x)$ and $qv \in R^*(z)$,
- $z \in R^*(uq)$ and $y \in R^*(pv)$,
- $(\alpha(p), \alpha(q)) \in R$.

Since $qv \in R^*(z) \in R^*(uq)$ and $|u| + |q| = (|x| + |z|) - |x|$, by the induction hypothesis applied to $qv \in R^*(uq)$ we obtain that there exist $m \leq 2|\Sigma|$, $t'_0, \ldots, t'_m \in \Sigma^*$ such that:

- $t'_0 \cdots t'_m \in R^*(u)$ and $v \in R^*(t'_m \cdots t'_0)$,
- $(\alpha(t'_j), \alpha(t'_i)) \in R$ for all $j > i+1$,
- $\alpha(t'_0 \cdots t'_{m-1}) \subseteq \alpha(q)$.

From the above conditions it follows that:

- $t'_0 \cdots t'_m p \in R^*(up) \subseteq R^*(x)$ and $y \in R^*(pv) \subseteq R^*(pt'_m \cdots t'_0)$,
- $(\alpha(p), \alpha(t'_i)) \in R$ for all $0 \leq i \leq m-1$, since $(\alpha(p), \alpha(q)) \in R$,
- $\alpha(t'_0 \cdots t'_m) \subseteq \alpha(z)$ since $\alpha(t'_0 \cdots t'_m) = \alpha(u)$.

It then suffices to set $t_i = t'_i$ for $i = 0, \ldots, m$ and $t_{m+1} = p$. It remains to show that $m \leq 2|\Sigma|$. This is due to the fact that for each $i \leq m-2$, $\alpha(t_0 \cdots t_i) \subsetneq \alpha(t_0 \cdots t_{i+2})$ since $(\alpha(t_j), \alpha(t_i)) \in R$ for all $j > i+1$. □

Remark 5. The converse of this theorem also holds, it suffices to take

$$z = t_0 \cdots t_{p-1} t_0 \cdots t_{p-2} \cdots t_0 t_1 t_0.$$

Corollary 5. $\mathcal{R}_c^* = \mathcal{R}_c^{2|\Sigma|}$.

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 7. 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 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 $\mathcal{R}_c^* = \mathcal{R}_c^{2|\Sigma|}$. □

6 Applications

We illustrate in this section the use of our results in two examples of applications. The first one concerns the verification problem of partial-order models (such as message sequence charts), and the second one concerns the verification problem of parametrized networks of processes.

6.1 Verification of partial-order models: message sequence charts

Let us call a *partial-order model* a model M such that its set of behaviors (execution paths) can be characterized as the closure of the a regular set of behaviors corresponding to a finite-state transition systems T_M under a semi-commutation system $\mathcal{R}_M$. Typically, T_M and $\mathcal{R}_M$ can be easily extracted from the "syntactical" description of the model M, and are polynomial in the size of M, T_M corresponding to the underlying control structure of M and $\mathcal{R}_M$ defining the independence relation between the actions of M. An example of partial-order models are Petri nets and the semi-commutation system $\mathcal{R}_M$ captures the independence of transitions (e.g., transitions that have no common place in their pre/post domain are called independent). We consider in this section another example of such models, High-level Message Sequence Charts (HMSCs for short).

Given a property (i.e., a set of behaviors) ϕ, the verification problem of M against ϕ, i.e., checking whether M satisfies ϕ, consists in deciding whether the set of behaviors of M is included in ϕ. However, since the set of behaviors of M corresponds to $\mathcal{R}_M^*(T_M)$, the verification problem is hard or even impossible to solve by computing precisely this set.

The idea of partial-order based verification methods is to reduce the verification problem of M against ϕ to the problem of checking whether the set of behaviors in T_M is included in ϕ (i.e., solving the verification problem without computing the closure of T_M), provided that ϕ is closed under the semi-commutation system $\mathcal{R}_M$. Indeed, since ϕ is closed under $\mathcal{R}_M$, this is also the case for $\neg\phi$, and then it is easy to see that $T_M \cap \neg\phi = \emptyset$ if and only if $M \cap \neg\phi = \emptyset$.

This approach is interesting in practice since it uses directly the system T_M and therefore it avoids the computation of its closure under semi-commutation rewriting which is an expensive operation, and even impossible in general (since closure of regular sets are not regular). On the other hand, this approach is crucially based on the fact that the property ϕ *must* be closed under $\mathcal{R}_M$. However, it is often more convenient to express properties regardless of all possible interleavings of independent actions in the considered model. Therefore, it is important to have algorithms for computing closures of properties under semi-commutation rewriting. We advocate here to define properties to check or their negations as APC expressions, and to use our algorithm (Theorem 5) in this context in order to close them under semi-commutation systems. In fact, typical properties to check are safety properties expressing for instance that some "bad" patterns never appear in the computations. It is quite natural to express the negations of such properties by enumerating the bad patterns, and this can be done using APC expressions. Therefore, depending whether the property ϕ or its negation $\neg\phi$ is definable as an APC expressions, we can use our techniques to close this expression and use it either positively (i.e., by checking that $T_M \subseteq \mathcal{R}_M^*(\phi) \neq \emptyset$) or negatively (i.e. by checking that $T_M \cap \mathcal{R}_M^*(\neg\phi) \neq \emptyset$) in order to decide whether T_M satisfies ϕ.

Let us present a concrete example of this approach. We consider the verification problem of scenarios described by High-Level Message Sequence Charts (HMSC). MSCs are a graphical specification language for communications protocols, standardized by the ITU and integrated in UML. An MSC scenario is a partial-order model for asynchronous FIFO message exchange of concurrent processes. It depicts a sequence of totally ordered events on each process, relating every send event with a receive event by message arrows. For simplicity we assume that arrows are drawn in such a way that they do not cross. An HMSC is a finite transition system with nodes labeled by MSCs. Consider now a system HMSC S including two processes p and q for which we want to verify that p cannot send two consecutive messages to 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).

The set of behaviors of the system S can be seen as the closure of the transition system T_S underlying its description, and which is polynomial in the size of S, under the commutation system $\mathcal{R}_S$ containing all pairs of actions (e, f) that belong to different processes and such that they do not represent a send-receive pair on the same channel. Then, a bad scenario contains for example an occurrence of the sequence $srsr$, which means two consecutive messages from p to q. So, suppose that we want to verify that the HMSC S satisfies ϕ where $\neg\phi = \Sigma^* srsr \Sigma^*$ (i.e., the set of sequences containing the bad pattern). Clearly, $\neg\phi$ is not closed under $\mathcal{R}_S$ since there might be some actions on other processes happening, e.g. the sequences $sarbsr$, $asrsbr$, $srsrab$ with a, b happening on some process $t \neq p, q$ are all equivalent. We can use our algorithm to close $\neg\phi$ under $\mathcal{R}_S$ and we obtain the property $\mathcal{R}_S^*(\neg\phi)$:

$$\Sigma^* s(\Sigma \setminus \Sigma_p)^* r(\Sigma \setminus (\Sigma_p \cup \Sigma_q))^* s(\Sigma \setminus \Sigma_q)^* r \Sigma^*$$

Now, as explained above, checking the fact that S satisfies ϕ it is equivalent to check that $T_S \cap \mathcal{R}_S^*(\neg\phi)$ is empty.

6.2 Regular Model Checking

Regular model checking is a uniform framework based on automata/regular languages for the verification of infinite state systems, and in particular parametrized networks of identical processes [3, 12, 6, 25, 28]. In this context, configurations of systems are encoded as words, regular languages (finite-state automata) are used to represent and manipulate potentially infinite sets of configurations, and actions of the systems are modeled as regular relations between words (or as word rewrite systems).

Therefore, let us represent a system by a pair $\langle \phi_0, \mathcal{R} \rangle$, where ϕ_0 is a regular language corresponding to the initial configurations, and $\mathcal{R}$ is a regular relation representing the different actions of the system. The verification of such systems is reduced to reachability analysis, i.e., to computing, when possible, the regular language $\mathcal{R}^*(\phi_0)$ representing the set of all reachable configurations. 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, or Dijkstra's protocols, are all expressible as APCs (see [2, 1, 3, 12, 6, 28]).

However, it is easy to see that computing the closure of regular languages (or even APC langauges) by regular relations is impossible in general (the transition relation of any Turing machine can be straightforwardly encoded as a regular relation). Then, the main issue in regural model checking is the design of powerful techniques which help the termination of the computation of the reachable configurations in practical cases (without guarantee of termination in general).

Results such as Theorem 5 and 6 can be used in this context to compute the effect of *meta-transitions* in order to *accelerate* the iterative process of computing the set of reachable configurations. Let us explain this approach. Let $\mathcal{C}$ be a subclass of regular languages, and assume that the system $\mathcal{R}$ can be decomposed into $\mathcal{R} = \mathcal{R}_1 \cup \cdots \cup \mathcal{R}_n \cup \mathcal{R}'$ where,

- for every $i \in \{1, \ldots, n\}$, the relation $\mathcal{R}_i$ is in some class such that for every set $L \in \mathcal{C}$, the set $\mathcal{R}_i^*(L)$ is again in $\mathcal{C}$ and it is effectively constructible,
- $\mathcal{R}'$ is a regular relation such that, for every set $L \in \mathcal{C}$, the set $\mathcal{R}'(L)$ is an effectively constructible $\mathcal{C}$ set.

Then, the reachability set $\mathcal{R}^*(\phi_0)$ can be computed as the limit of the increasing sequence $(X_i)_{i \geq 0}$ defined by:

$$X_0 = \phi_0$$
$$\forall i \geq 0.\ X_{i+1} = X_i \cup \mathcal{R}_1^*(X_i) \cup \cdots \cup \mathcal{R}_n^*(X_i) \cup \mathcal{R}'(X_i)$$

We present below an example illustrating this approach using our results about the class of APC languages.

Actually, we could consider an alternative principle where we relax the requirement that images of $\mathcal{C}$ languages by the $\mathcal{R}_j^*$'s and by $\mathcal{R}'$ are effectively constructible $\mathcal{C}$ sets, and require only that these sets should be effectively *regular*. However, if $\mathcal{R}_j^*$ images can be computed only $\mathcal{C}$ languages, we need to be able to check if a given regular language is effectively representable in the class $\mathcal{C}$. Then, the principle we can adopt is

to compute at each step $i+1$, and for each relation $\mathcal{R}_j$, either (1) the $\mathcal{R}_j^*$ image of X_i if X_i is an effectively $\mathcal{C}$ definable language, or (2) the $\mathcal{R}_j$ image of X_i otherwise. This principle can be applied for instance in the case of the class of APC languages due to Theorems 3 and 4. However, checking at each iteration if the obtained language is in $\mathcal{C}$ could be expensive (as shown for instance by Theorems 3 and 4 for the case of APC). Therefore, we prefer to adopt the previous schema whenever possible, and thus, we are in general more interested in having effective closure results of classes of languages under classes of relations than in having results showing only the regularity of the closures.

Let us present now a simple example illustrating our approach. We consider a *lift controller* which has the following behavior: People can arrive at any time to any floor and declare their intention to move up or down. The lift is initially at the lowest floor, and it keeps moving from the lowest floor to the uppermost one, and back. In its ascending (resp. descending) phase, it takes everyone who is waiting for moving up (resp. down) and ignores the others (they are taken into account in the next phase). The problem we address is to analyzing the set of possible behaviors of this system for an *arbitrary number of floors*.

For every number of floors n, a configuration of this system can be represented by a word of the form $\#x_1 \cdots x_j y x_{j+1} \cdots x_n \#$, where $y \in \{L\uparrow, L\downarrow\}$, and $x_i \in \{\bot, p\uparrow, p\downarrow, p\updownarrow\}$, for $i \in \{1, \ldots, n\}$. The symbol corresponding to x_i represents the state of the i^{th} floor: $x_i = p\updownarrow$ if there are people waiting for moving up and down at floor i, $x_i = p\uparrow$ (resp. $x_i = p\downarrow$) means that at floor i people waiting only want to move up (resp. down), and $x_i = \bot$ means that nobody is waiting at floor i. The symbol corresponding to y gives the position of the lift: in the configuration given above, if $y = L\uparrow$ (resp. $y = L\downarrow$) then the lift is at floor $j+1$ (resp. j), and it is moving up (resp. down).

The set of all initial configurations (for an arbitrary number of floors) is the set $\phi_0 = \#L\uparrow \bot^*\#$, which means that the lift is initially at the lowest floor and there is no request at any floor. The dynamics of the system can be modeled by the rewriting rules (1)-(14).

Rules 1, 2, 3, and 4 are *symbol substitutions* modeling the arrival of users. Let us call request their corresponding action. Rules 5 and 6 (resp. 10 and 11) are *semi-commutations* modeling the moves of the lift upward (resp. downward). They correspond to the action move-up (resp. move-down). Rules 7 and 8 (resp. 12 and 13) represent the action of taking at some floor the people who want to move up (resp. down). We call the corresponding actions take-up (resp. take-down). Finally, rules 9 and 14 represent the actions of switching from the ascending to the descending phase (action up2down), and vice-versa (action down2up).

Rules 7, 8, 9, and 14 are simple *word substitutions* rewrite rules (i.e., rules of the form $u \to v$ where u and v are words). It is easy to see that APC languages are effectively closed under the application of word substitutions. Therefore, the images of APC languages by the transitions take-up, take-down, up2down and down2up are APC computable languages. Moreover, it is quite obvious that APC are effectively closed under iterative symbol substitution rewriting, i.e., under iterative application of rules of the form $a \to b$. Therefore, the effect of applying the meta-transition request* is

computable. Finally, using our algorithm behind Theorem 5, the images by move-up* and move-down* can be computed.

$$\bot \rightarrow p\uparrow \tag{1}$$
$$\bot \rightarrow p\downarrow \tag{2}$$
$$p\uparrow \rightarrow p\updownarrow \tag{3}$$
$$p\downarrow \rightarrow p\updownarrow \tag{4}$$
$$L\uparrow \bot \rightarrow \bot L\uparrow \tag{5}$$
$$L\uparrow p\downarrow \rightarrow p\downarrow L\uparrow \tag{6}$$
$$L\uparrow p\uparrow \rightarrow \bot L\uparrow \tag{7}$$
$$L\uparrow p\updownarrow \rightarrow p\downarrow L\uparrow \tag{8}$$
$$L\uparrow \# \rightarrow L\downarrow \# \tag{9}$$
$$\bot L\downarrow \rightarrow L\downarrow \bot \tag{10}$$
$$p\uparrow L\downarrow \rightarrow L\downarrow p\uparrow \tag{11}$$
$$p\downarrow L\downarrow \rightarrow L\downarrow \bot \tag{12}$$
$$p\updownarrow L\downarrow \rightarrow L\downarrow p\uparrow \tag{13}$$
$$\#L\downarrow \rightarrow \#L\uparrow \tag{14}$$

Table 1 shows the computations of the reachable configurations of the lift controller. In the first column we give the set of configurations to which we apply the action given in column 2. The obtained sets are shown in columns 3 and 4.

ϕ_0	request*	$\#L\uparrow (\bot + p\uparrow + p\downarrow + p\updownarrow)^*\#$	ϕ_1
ϕ_1	move-up*	$\#(\bot + p\downarrow)^*L\uparrow (\bot + p\uparrow + p\downarrow + p\updownarrow)^*\#$	ϕ_2
ϕ_2	request*	$\#(\bot + p\uparrow + p\downarrow + p\updownarrow)^*L\uparrow (\bot + p\uparrow + p\downarrow + p\updownarrow)^*\#$	ϕ_3
ϕ_3	take-up	$\#(\bot + p\uparrow + p\downarrow + p\updownarrow)^*(\bot + p\downarrow)L\uparrow (\bot + p\uparrow + p\downarrow + p\updownarrow)^*\#$	$\subseteq \phi_3$
ϕ_3	up2down	$\#(\bot + p\uparrow + p\downarrow + p\updownarrow)^*L\downarrow \#$	ϕ_4
ϕ_4	move-down*	$\#(\bot + p\uparrow + p\downarrow + p\updownarrow)^*L\downarrow (\bot + p\uparrow)^*\#$	ϕ_5
ϕ_5	request*	$\#(\bot + p\uparrow + p\downarrow + p\updownarrow)^*L\downarrow (\bot + p\uparrow + p\downarrow + p\updownarrow)^*\#$	ϕ_6
ϕ_6	take-down	$\#(\bot + p\uparrow + p\downarrow + p\updownarrow)^*L\downarrow (\bot + p\uparrow)(\bot + p\uparrow + p\downarrow + p\updownarrow)^*\#$	$\subseteq \phi_6$
ϕ_6	down2up	$\#L\uparrow (\bot + p\uparrow + p\downarrow + p\updownarrow)^*\#$	$= \phi_1$

Table 1. Reachability Analysis of the Lift Controller

As shown in Table 1, the use of acceleration based on meta-transition computations allows to compute the reachability set of the lift controller in a finite number of iterations. We can also provide a *finite abstraction* of this infinite-state model. Indeed, Table 1 defines an abstract reachability graph of the lift controller (see Figure 3).

7 Conclusion

We considered the class of regular expressions called APC which appears naturally in many contexts such as modeling and verifying concurrent systems, and regular model

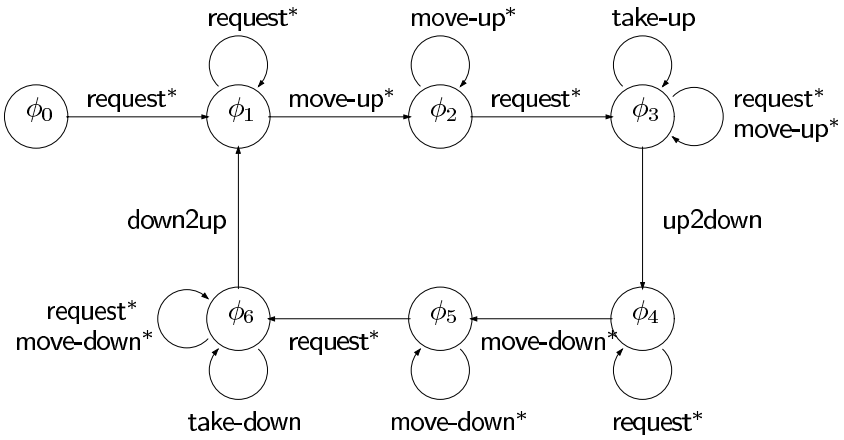

Fig. 3. Abstract reachability graph of the lift controller

checking. We have considered several closure properties and complexity issues of this class.

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. We have shown 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 with similar closure and decision 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 (twice 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 systems with other kinds of topologies, such as trees and grids.

References

1. P. Abdulla, A. Annichini, and A. Bouajjani. Symbolic Verification of Lossy Channel Systems: Application to the Bounded Retransmission Protocol. In *TACAS'99*. LNCS 1579, 1999.
2. 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.

3. 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.
4. 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.
5. 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.
6. A. Bouajjani, B. Jonsson, M. Nilsson, and T. Touili. Liveness and acceleration in parametrized verification. In *Proc. of CAV'00*, LNCS 1855, pp. 403–418, 2000.
7. E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.
8. V. Diekert and G. Rozenberg, editors. *The Book of Traces*. World Scientific, Singapore, 1995.
9. V. Diekert and Y. Métivier. Partial Commutation and Traces. In *Handbook of Formal Languages* (Eds. G. Rozenberg and A. Salomaa), volume 3, pp. 457–533. Springer, 1997.
10. L. Fribourg and H. Olsén. Reachability sets of parametrized rings as regular languages. *Electronic Notes in Theoretical Computer Science*, pp. 1–12, 1997.
11. P. Godefroid and P. Wolper. A partial approach to model checking. *Information and Computation*, 110(2):305–326, 1994.
12. 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.
13. 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.
14. A. Mazurkiewicz. Concurrent program schemes and their interpretations. DAIMI Rep. PB 78, Aarhus University, Aarhus, 1977.
15. Metivier. Title. Reference, year.
16. 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.
17. Z. Manna and A. Pnueli. *The temporal logic of reactive and concurrent systems*. Springer, 1991.
18. Y. Métivier. On recognizable subsets of free partially comutative monoids. In *Theoretical Computer Science*, 58:201–208, 1988.
19. A. Muscholl, D. Peled, and Z. Su. Deciding properties of message sequence charts. In *Proc. of FoSSaCS'98*, LNCS 1378, pp. 226–242, 1998.
20. A. Muscholl. Über die Erkennbarkeit unendlicher Spuren. Teubner Verlag, Stuttgart-Leipzig, 1996.
21. Edward Ochmański. Regular behaviour of concurrent systems. *Bulletin of EATCS*, 27:56–67, 1985.
22. D. Peled. All from one, one from all: on model checking using representatives. In *Proc. of CAV'93*, LNCS 697, pp. 409–423, 1993.
23. J.-E. Pin and P. Weil. Polynomial closure and unambiguous product. *Theory of Computing Systems*, 30:383–422, 1997.
24. Z. Manna, A. Pnueli. The Temporal Logic of Reactive and Concurrent Systems. Springer-Verlag, 1992.
25. A. Pnueli and E. Shahar. Liveness and acceleration in parametrized verification. In *Proc. of CAV'00*, LNCS 1855, pp. 328–343, 2000.
26. 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.
27. W. Thomas. Classifying regular events in symbolic logic. *Journal of Computer and System Sciences*, 25:360–376, 1982.
28. T. Touili. Widening Techniques for Regular Model Checking. In *Vepas Workshop*. Volume 50 of Electronic Notes in TCS, 2001.
29. A. Valmari. A stubborn attack on state explosion. *Formal Methods in System Design*, 1:297–322, 1992.
30. M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc. of LICS '86*, pp. 332–344, 1986.
31. P. Wolper and B. Boigelot. Verifying systems with infinite but regular state spaces. In *Proc. of CAV'98*, LNCS 1427, pp. 88–97,1998.