Acta Informatica 39, 31–70 (2003)
Digital Object Identifier (DOI) 10.1007/s00236-002-0099-y

Acta Informatica
© Springer-Verlag 2003

Synchronized regular expressions

Giuseppe Della Penna[1], Benedetto Intrigila[1], Enrico Tronci[2], Marisa Venturini Zilli[2]

[1] Dip. di Informatica, Università di L'Aquila, Coppito 67100, L'Aquila, Italy (e-mail: {dellapenna,intrigila}@univaq.it)

[2] Dip. di Scienze dell'Informazione, Università di Roma "La Sapienza", Via Salaria 113, 00198 Roma, Italy (e-mail: {tronci,zilli}@dsi.uniroma1.it)

Received: 30 July 2001 / 16 May 2002

Abstract. Text manipulation is one of the most common tasks for everyone using a computer. The increasing number of textual information in electronic format that every computer user collects everyday also increases the need of more powerful tools to interact with texts. Indeed, much work has been done to provide simple and versatile tools that can be useful for the most common text manipulation tasks. Regular Expressions (RE), introduced by Kleene, are well known in the formal language theory. RE have been extended in various ways, depending on the application of interest. In almost all the implementations of RE search algorithms (e.g. the `egrep` [15] UNIX command, or the Perl [20] language pattern matching constructs) we find *backreferences*, i.e. expressions that make reference to the string matched by a previous subexpression. Generally speaking, it seems that all kinds of synchronizations between subexpressions in a RE can be very useful when interacting with texts. In this paper we introduce the Synchronized Regular Expressions (SRE) as an extension of the Regular Expressions. We use SRE to present a formal study of the already known backreferences extension, and of a new extension proposed by us, which we call the *synchronized exponents*. Moreover, since we are dealing with formalisms that should have a practical utility and be used in real applications, we have the problem of how to present SRE to the *final users*. Therefore, in this paper we also propose a user-friendly syntax for SRE to be used in implementations of SRE-powered search algorithms.

This research has been partially supported by MURST project TOSCA.

1 Introduction

Text manipulation is one of the most common tasks for everyone using a computer. The increasing number of textual information in electronic format that every computer user collects everyday (email, web pages, word processor documents, even paper documents usually converted to electronic format to save space) also increases the need of more powerful tools to interact with texts. This is true at every level – from the beginner to the advanced user – although expert users may have to perform tasks that can be accomplished only by writing ad-hoc programs.

Indeed, much work has been done to provide simple and versatile tools that can be useful for the most common text manipulation tasks.

Regular Expressions (RE from now on), introduced by Kleene, are well known in the formal language theory. Today, RE are present in almost all the text processing programs, mainly used in search/replace functions. Users are familiar with this formalism, and this suggests to exploit all the power of RE to perform much more complex operations on texts. RE have been extended in various ways, depending on the application of interest. As an example, wildcards like '?' or '*' are used as abbreviations of more complex RE in all the operating system command shells.

Patterns introduced by Angluin [2] and also studied by other authors (see [19] and [17] for an overview on patterns), are extended RE used to find identical substrings in the same string. In almost all the implementations of RE search algorithms (e.g. the `egrep` [15], `sed` and `awk` UNIX commands, or the Perl [20] language pattern matching constructs) we find *backreferences* (defined in [1], see also [9]) as a generalization of patterns, i.e. expressions that make reference to the string matched by a previous subexpression.

Another interesting extension that, to the best of our knowledge, has not been studied or implemented yet, may allow to find if certain subexpressions are repeated *the same number of times* in a text. This can be useful in a variety of cases, from integrity checks to advanced word count tools, especially when mixed with other extensions such as backreferences.

Generally speaking, it seems that all kinds of synchronizations between subexpressions in a RE (like backreferences) can be very useful when interacting with texts. In this paper, we introduce the Synchronized Regular Expressions (SRE) as an extension of RE. We use SRE to present a formal study of the already known backreferences extension, and of a new extension proposed by us, which we call the *synchronized exponents*. We focus on these kinds of synchronizations since they share very good properties:

- they appear to be useful in a significant number of cases;
- they can be implemented in a very user-friendly way strictly similar to the use of ordinary wildcards (see Sect. 6);

- they have an acceptable computational complexity, when used under reasonable constraints (see Sect. 5).

We also think that such extended regular expressions, besides the examples we consider (see Sect. 6), may be found useful in a variety of other data processing problems such as data compression and DNA sequences reconstruction.

In this paper we give a formal syntax and a semantics for both extensions. Then we study the classification of SRE in the formal languages hierarchy. Finally, we study the complexity of the pattern matching problem.

Since we are dealing with formalisms that should have a practical utility and be used in real applications, we also have the problem of how to present SRE to the *final users*. We started looking at how backreferences and other already implemented extensions are given to the user in well known programs and noticed that, despite their obvious utility, few people actually use them.

Indeed, many nonprogrammer users are not able to write the often complex commands (sometimes even small programs) needed to use these extensions. Moreover, the implementations are nonstandard: users find difficult to understand the new syntactic constructs proposed for such extensions in each application. Backreferences, for example, have a recursive definition that allows to nest expressions with their references, and to use the full RE power in the binding operation. Instead, most users do not need all this power, and their approach to backreferences is troublesome.

The second aim of this paper is to propose a user-friendly syntax for SRE to be used in implementations of SRE-powered search algorithms. Our syntax has two levels: the first, aimed at the "advanced users", presents the extensions in a fully functional fashion available through the RE syntax, without the introduction of other complex constructs or the need of programming. Nevertheless, the full syntax can also be seen as a high-level programming language for algorithms that handle large quantities of structured data. In this way, SRE can help programmers to rapidly write the code to solve complex data manipulation problems.

On the other hand, we identified a set of processing tasks that represent, in our opinion, those that a beginner user may need, and in the second level syntax we present a set of macro-constructs that accomplish these tasks using very simplified, intuitive constructs.

The paper is organized as follows.

In Sect. 2 we define our Synchronized Regular Expressions and show how to compute the languages associated to them. The SRE, as we will see, introduce the concept of "synchronized elements" inside the regular expressions.

In Sect. 3 we study where such languages lie in the formal language hierarchy, showing that they are context sensitive and, under certain conditions, included in the Scattered Context Grammars [7] (a proper subclass of context sensitive grammars). Moreover, we prove that there are context free languages that cannot be generated by a SRE.

In Sect. 4 we address the membership problem in our context. We show that the full membership problem turns out to be NP-complete. Thus, we look for restrictions that can make the problem tractable. A first natural restriction is to limit the number of occurrences of each synchronized element. But, even when this number is bounded by 2, we prove that the problem remains NP-complete.

In Sect. 5 we consider the other natural restriction, namely bounding the number of synchronized elements. This time the membership problem turns out to be polynomial.

In Sect. 6 we address the problem of defining a user-friendly syntax for SRE, suitable to different kinds of users, and use this syntax to give a number of concrete examples.

Sect. 7 contains a comparison between our approach and others, with respect to both the theory of formal languages and the implementation of regular expressions.

2 Synchronized regular expressions

In this section we define our extension of the classical RE. We start with a brief introduction to the classical RE syntax and a description the pattern extension from [17]. Then we give the definition of SRE using the syntax for backreferences from [1] and introducing the new syntax for synchronized exponents. The semantics of the new language is given in subsection 2.4, followed by a collection of SRE examples.

2.1 Regular expressions

The canonical definition of RE is briefly recalled below. For a more detailed overview on this argument, the reader may refer e.g. to [12].

Definition 1. *The* Regular Expressions *on an alphabet A are the elements of the set RE defined as follows:*

- $\emptyset \in RE$ (empty language)
- $\epsilon \in RE$ (empty string)
- $\forall a \in A \quad a \in RE$ (letters)

if $e_1, e_2 \in RE$, then:

1. $e_1 \cdot e_2 \quad (\text{or} \quad e_1 e_2) \in RE$ (concatenation)
2. $e_1 + e_2 \in RE$ (union)
3. $e_1^* \in RE$ (star)
4. $(e_1) \in RE$ (parentheses)

Definition 2. *The* language associated *to a regular expression $r \in RE$ is $\mathcal{L}_{RE}(r)$ where the function $\mathcal{L}_{RE} : RE \to 2^{A^*}$ is defined as follows:*

1. $\mathcal{L}_{RE}(\emptyset) = \emptyset$
2. $\mathcal{L}_{RE}(\epsilon) = \{\epsilon\}$
3. $\mathcal{L}_{RE}(a) = \{a\} \quad \forall a \in A$
4. $\mathcal{L}_{RE}((e_1)) = \mathcal{L}_{RE}(e_1)$
5. $\mathcal{L}_{RE}(e_1 \cdot e_2) = \mathcal{L}_{RE}(e_1) \cdot \mathcal{L}_{RE}(e_2)$
6. $\mathcal{L}_{RE}(e_1 + e_2) = \mathcal{L}_{RE}(e_1) \cup \mathcal{L}_{RE}(e_2)$
7. $\mathcal{L}_{RE}(e_1^*) = \mathcal{L}_{RE}(e_1)^*$

2.2 Regular expressions with patterns

In this section we introduce the extension of RE from [2]. The pattern extension uses the same syntax we will adopt in SRE backreferences, so it seems useful to introduce patterns before defining the SRE.

Definition 3. *Let Σ be a finite alphabet containing at least two symbols and $X = \{x_1, x_2, \ldots x_k\}$ be a countable set disjoint from Σ of symbols called* variables. *A* pattern *is any finite non null string of symbols from $\Sigma \cup X$.*

The set of all patterns is denoted by P_*. Any function $\sigma : P_* \to P_*$ that is a none rasing homomorphism w.r.t. pattern concatenation (defined in the obvious way) and acts like the identity when restricted to Σ is called a *substitution*. In other words, a substitution takes a pattern and returns the same pattern with some variables replaced with different variables or letters (from Σ).

Definition 4. *If p is a pattern,* the language of *p, denoted by $\mathcal{L}_P(p)$, is $\{\alpha \in \Sigma^+ | \exists \sigma : \sigma(p) = \alpha\}$.*

That is, the language of a pattern p is the set of all strings in Σ^+ that can be obtained by applying a substitution σ to p.

For example, if $\Sigma = \{a, b, c, \ldots\}$ and $X = \{x_1\}$, the language of $x_1 a x_1$ is $\beta\, a\, \beta$ with $\beta \in \Sigma^+$.

2.3 Syntax of synchronized regular expressions

In this section we describe the syntax of Synchronized Regular Expressions. Compared with the two languages introduced in the previous subsections, SRE introduce the concept of *synchronization* between different parts of the expression, thus achieving more expressive power.

Remark 1. In the following, when referring to SRE we use the word *exponent* as a shorthand for *exponent variable*. That is, exponents are not numbers but literals that can be instantiated with numbers (with the exponent binding or assignment operation).

Definition 5. *The* Synchronized Regular Expressions *on an alphabet A, a set of* variables V *and a set of* exponents X *are the elements of the set* SRE *defined as follows:*

- $\emptyset \in SRE$ (empty language)
- $\epsilon \in SRE$ (empty string)
- $\forall a \in A \quad a \in SRE$ (letters)
- $\forall v \in V \quad v \in SRE$ (variables)

if $e_1, e_2 \in SRE$, then:

1. $e_1^* \in SRE$ (star)
2. $\forall x \in X \quad e_1^x \in SRE$ (exponentiation)
3. $\forall v \in V \quad (e_1)\,\%v \in SRE$ (variable binding)
4. $e_1 e_2 \in SRE$ (concatenation)
5. $e_1 + e_2 \in SRE$ (union)

Backreferences are created using variables. A variable is bound to a string from a given language using the binding operation (3). In this sense, SRE variables can be seen as variables with a prescribed domain. For a detailed study of this kind of patterns, see [6]. We will address variable occurrences that are not arguments of a binding operation as *backreferences*.

Example 1. Some examples of correct synchronized regular expressions include:

- $a\ (b+c)^*$ (as well as any other correct RE)
- $a^x\ (b+c)^y$ (exponents used without synchronization – equivalent to the star symbol of RE)
- $a^x\, b\, c^x$ (a SRE with exponent synchronization)
- $(b+c)\,\%v\, a\, v$ (a SRE using a backreference v bound to the subexpression $(b+c)$)

The reader may refer to subsection 2.5 for some more complex examples of SRE.

The backreferences syntax from [1] seems to be underspecified. Indeed, it allows different interpretations (compare, for example, [1] with the more end-user oriented [9]) since the intended meaning of some expressions implies several restrictions that are not expressed in the syntax. These restrictions may cause problems to the user. In fact, the above definition allows:

– Multiple bindings on the same variable. For example, consider the following SRE:
$$(a^*)\,\%v\,b\,v\ \ (b^*)\,\%v\,c\,v$$
If, as natural, we suppose that a binding replaces the previous one on the same variable, the language generation would rely on a not specified "expression ordering".

– Loops on variable bindings. These may cause *deadlocks*, as in the following example:
$$(v_1\,a)\,\%v_2\,d\ \ (v_2\,b)\,\%v_1$$
Expressions containing binding loops always express the empty language or do not generate any definite language, depending on the interpretation.

– Recursion on a variable binding. This is a special case of the previous problem. The following is a simple example of recursion:
$$(a\,v\,b)\,\%v$$

– Late binding. A variable could be used before being bound to an expression, for example:
$$a\,v\,b\ \ (c^*)\,\%v$$
This may lead to various problems, and it is unnecessary in practice.

– Unbound variables. A variable could be used while no binding for it occurs in the expression:
$$a\,v\,b\,v$$
A SRE with unbound variables does not express a definite language.

– Disjunction between an expression and its backreference. An expression can bind a variable on one side of a sum "+" and backreference it on the other side:
$$(a^*)\%v\ +\ v$$
Since the evaluation of the two subexpressions is mutually exclusive, we have an unbound variable in the right side.

To fix the syntax, we impose the following restrictions on valid SRE:

Definition 6. *A Synchronized Regular Expression is* valid *if, whenever the expression is read left-to-right, the following holds:*

1. *Bindings are always done on* fresh *(i.e. not previously used) variables.*
2. *Backreferences always refer to* bound *(i.e. not fresh) variables.*

Such restrictions solve all the above problems, with the exception of the disjunction problem that is treated by the semantic rules in Sect. 2.4.

Proposition 1. *A valid SRE does not allow:*

1. *multiple bindings on the same variable,*
2. *looped variable bindings,*
3. *recursive bindings,*
4. *late bindings,*
5. *unbound variables.*

Proof. 1. Since the binding operation can only be used on fresh variables, an already bound variable cannot be bound again.
2. To “loop” two variable bindings, each variable must appear in the expression that defines the other, but this would imply the use of a variable before its binding.
3. Recursion is impossible for the same reason, since a variable is bound only *after* the binding operation. This implies that *in* the binding expression the variable is fresh and cannot be used as backreference.
4. Late binding is directly disallowed by the second clause of Definition 6.
5. There cannot be unbound variables, since variables can only be used as backreferences *after* being bound.

Remark 2. Since we shall consider *valid* SRE only, we stipulate that, from now on, “SRE” stands for “valid SRE”.

2.4 Semantics of synchronized regular expressions

In this section we define the language generated by a SRE. Due to the synchronization issues, the language cannot be unambiguously described through a simple set of recursive rules, as done in Definition 2 for the regular expressions. Therefore, we introduce a function, $Eval$, that decides whether a string belongs or not to the language of a given SRE. Then, we define the language associated to a SRE in terms of $Eval$.

The function $Eval$ encodes the semantics of SRE as originally presented in [1], using induction on the structure of expressions.

Let $Eval\,(PS, V_B, V_F, E_B)$ be our evaluation function, where

- PS is a set of $(pattern, string)$ pairs;

- V_B is a set of $(variable, string)$ pairs, representing all the variables already associated to a string by a binding operation;
- V_F is a set of $(variable, string)$ pairs, representing the unsolved backreferences (variables used but not yet bound) and the strings they should match;
- E_B is a set of $(exponent, number)$ pairs, representing all the exponents already assigned.

The function is true iff, for every $(pattern, string)$ pair in PS, $pattern$ matches $string$ using the binding rules given in V_B, V_F and E_B.

The rules for $Eval$ are the following:

$$\begin{aligned}&Eval\left(\{(a,\alpha)\}\cup PS, V_B, V_F, E_B\right) =\\&\quad Eval\left(PS, V_B, V_F, E_B\right)\wedge(\alpha = a)\end{aligned}$$

$$\begin{aligned}&Eval\left(\{(e_1e_2,\alpha)\}\cup PS, V_B, V_F, E_B\right) =\\&\quad \bigvee\nolimits_{\alpha_1\alpha_2=\alpha} Eval\left(\{(e_1,\alpha_1),(e_2,\alpha_2)\}\cup PS, V_B, V_F, E_B\right)\end{aligned}$$

$$\begin{aligned}&Eval\left(\{(e_1+e_2,\alpha)\}\cup PS, V_B, V_F, E_B\right) =\\&\quad Eval\left(\{(e_1,\alpha)\}\cup PS, V_B, V_F, E_B\right)\vee\\&\quad Eval\left(\{(\overline{e_2},\alpha)\}\cup PS, V_B, V_F, E_B\right)\end{aligned}$$

where $\overline{e_2}$ is obtained from e_2 by substituting the leftmost backreference of each variable that is not bound in e_2 with the corresponding binding taken from e_1. This solves the "disjunction problem" introduced in Sect. 2.3.

$$\begin{aligned}&Eval\left(\{((e)\,\%v,\alpha)\}\cup PS, V_B, V_F, E_B\right) =\\&\quad Eval\left(\{(e,\alpha)\}\cup PS, V_B', V_F', E_B\right)\wedge\left(\forall\,(v,\alpha')\in V_F\ \ (\alpha'=\alpha)\right)\end{aligned}$$

where

$$\begin{aligned}&V_B' = \left(V_B\setminus\{(v,\alpha')\,|\alpha'\in A^*\}\right)\cup\{(v,\alpha)\}\\&V_F' = V_F\setminus\{(v,\alpha')\,|\alpha'\in A^*\}\end{aligned}$$

note that if $Eval$ is applied only to valid SRE (see Definition 6), the rule for V_B' can be simplified deleting the multiple-binding check, thus yielding:

$$V_B' = V_B\cup\{(v,\alpha)\}$$

$$\begin{aligned}&Eval\left(\{(v,\alpha)\}\cup PS, V_B, V_F, E_B\right) =\\&\quad\begin{cases}Eval\left(PS, V_B, V_F, E_B\right)\wedge(\alpha=\alpha') & \text{if } \exists\,(v,\alpha')\in V_B\\ Eval\left(PS, V_B, V_F\cup\{(v,\alpha)\}, E_B\right) & \text{otherwise}\end{cases}\end{aligned}$$

$$\begin{aligned}&Eval\left(\{(e^x,\alpha)\}\cup PS, V_B, V_F, E_B\right) =\\&\quad\begin{cases}Eval\left(\{(e^n,\alpha)\}\cup PS, V_B, V_F, E_B\right) & \text{if } (x,n)\in E_B\\ \bigvee\nolimits_{n\in\mathcal{N}} Eval\left(\{(e^n,\alpha)\}\cup PS, V_B, V_F, E_B\cup\{(x,n)\}\right) & \text{otherwise}\end{cases}\end{aligned}$$

$$\begin{aligned}&Eval\left(\{\}, V_B, V_F, E_B\right) =\\&\quad\begin{cases}true & \text{if } V_F=\emptyset\\ false & \text{otherwise}\end{cases}\end{aligned}$$

Given a string α and a SRE e, we say that α *is generated by* (i.e. *belongs to the language of*) e iff the expression $Eval\left(\{(e,\alpha)\},\emptyset,\emptyset,\emptyset\right)$ returns $true$.

Remark 3. From another point of view, we can also say that $Eval$ solves the pattern matching problem of e on α.

Given a string α and a pattern e, i.e. a regular expression (or a SRE in our case), the *pattern matching problem* of e on α consists in finding an instance of e that is identical to α (if it exists). Therefore, $Eval\left(\{(e,\alpha)\},\emptyset,\emptyset,\emptyset\right)$ also solves the pattern matching problem of e on α.

Note that the function $Eval$ is always defined on every input. In fact, the rules given above may extend the PS set, but always decrease the complexity of its elements. When an element reduces to a single letter or variable it is removed from the set without adding other elements, so finally the evaluation reaches the exit rule where $PS = \emptyset$.

However, $Eval$ is well defined only under the restrictions given in Definition 6. Indeed, without such restrictions, $Eval$ is always defined but nondeterministic (because of the "multiple-binding problem" explained in Sect. 2.3).

Definition 7. *The* language associated *to a synchronized regular expression $e \in SRE$ is $\mathcal{L}_{SRE}(e)$ where the function $\mathcal{L}_{SRE} : SRE \to 2^{A^*}$ is defined as $\mathcal{L}_{SRE}(e) = \{\alpha \in A^* | Eval\left(\{(e,\alpha)\},\emptyset,\emptyset,\emptyset\right)\}$*

2.5 Examples

To evaluate a SRE more easily, the reader can actually assign all the exponents at the beginning of the evaluation. Definition 8 and Proposition 2 below show the correctness of this approach.

Definition 8. *An* exponent assignment *for a set of exponents X is a function $\sigma_X : X \to \mathbb{N}$ that assigns a natural number to each exponent in the set X. The* SRE exponent expansion function $expand_{\sigma_X} : SRE \to SRE$ *expands the exponentiations in SRE by replacing each subexpression having exponent $x \in X$ with $n \in \mathbb{N}$ repetitions of the same expression, where $n = \sigma_X(x)$. The definition of $expand_{\sigma_X}$ is as follows:*

$$expand_{\sigma_X}(e) = \begin{cases} v & \text{if } e \equiv v \\ expand_{\sigma_X}(e_1)\%v & \text{if } e \equiv (e_1)\%v \\ a & \text{if } e \equiv a \\ expand_{\sigma_X}(e_1)^* & \text{if } e \equiv e_1{}^* \\ expand_{\sigma_X}(e_1)\, expand_{\sigma_X}(e_2) & \text{if } e \equiv e_1 e_2 \\ expand_{\sigma_X}(e_1) + expand_{\sigma_X}(e_2) & \text{if } e \equiv e_1 + e_2 \\ expand_{\sigma_X}(e_1)^n, \text{where } n = \sigma_X(x) & \text{if } e \equiv e_1{}^x \end{cases} \tag{1}$$

We have the following

Proposition 2. *For every* $e \in SRE$*,* $\mathcal{L}_{SRE}(e) = \bigcup_{\sigma_X} \mathcal{L}_{SRE}(expand_{\sigma_X}(e))$.

Proof.
- $\mathcal{L}_{SRE}(e) \subseteq \bigcup_{\sigma_X} \mathcal{L}_{SRE}(expand_{\sigma_X}(e))$
  For all $\alpha \in A^*$, $\alpha \in \mathcal{L}_{SRE}(e) \Rightarrow \alpha \in \mathcal{L}_{SRE}(expand_{\sigma_X}(e))$ where σ_X is built using the set E_B from the final step of the evaluation of $Eval(\{(e,\alpha)\}, \emptyset, \emptyset, \emptyset)$. Note that $Eval$ is defined and returns true in this case since $\alpha \in \mathcal{L}_{SRE}(e)$.
- $\bigcup_{\sigma_X} \mathcal{L}_{SRE}(expand_{\sigma_X}(e)) \subseteq \mathcal{L}_{SRE}(e)$
  For all $\alpha \in A^*, \alpha \in \mathcal{L}_{SRE}(expand_{\sigma_X}(e)) \Rightarrow \alpha \in \mathcal{L}_{SRE}(e)$ where, on the right side, every time $Eval$ has to evaluate an unassigned exponent we choose the assignment given by σ_X. □

Moreover, the reader may use a left-to-right version of $Eval$ to evaluate the SRE without exponents. An informal description of such left-to-right $Eval$ is the following:

- Since a backreference can only be done when the corresponding variable is already assigned, at least the first binding expression cannot contain backreferences. Thus, we begin assigning a value to each variable whose associated expression does not contain backreferences (i.e. is actually a RE).
- Now there is at least one new binding expression that is free from unbound variables (it backreferences the previous variables that have been assigned). So we can assign a value to it and bind another variable, and so on.

We have also formalized this version of $Eval$, but we omit it here to shorten the exposition.

Example 2. SRE can define the well known non context free language $a^n b^n c^n$. Indeed, it can be generated by the SRE $a^x b^x c^x$.

Example 3. The SRE $(A^*)\,\%vv$ defines the non context free language of squared words. Indeed, each of its instances is obtained by assigning a string $\alpha \in A^*$ to v and performing the variable substitution. So the language of this SRE is

$$\mathcal{L}_{SRE}((A^*)\,\%vv) = \{\alpha\alpha | \alpha \in A^*\}$$

that is the language of squared words.

Example 4. More complex SRE can be built, like the expression

$$a^x (c + d)\,\%vb^* va^x.$$

The language of this SRE is the following:

$$\mathcal{L}_{SRE}\left(a^{x}\left(c+d\right)\%vb^{*}va^{x}\right)=$$
$$\{a^{n}cb^{m}ca^{n}|n,m\in\mathbb{N}\}\cup$$
$$\{a^{n}db^{m}da^{n}|n,m\in\mathbb{N}\}$$

3 Synchronized regular expressions in formal languages

In this section we classify the languages defined by Synchronized Regular Expressions in the hierarchy of formal languages. In the following, we address with "Synchronized Regular Expression Languages", or briefly with $L\left(SRE\right)$ the set of languages defined by SRE and with $\mathcal{L}_{SRE}\left(e\right)$ the language defined by the SRE e. In the same way, we use the expressions $L\left(RE\right)$, $L\left(CF\right)$ and $L\left(CS\right)$ to denote the set of languages defined by RE, context free grammars and context sensitive grammars, respectively, and with $\mathcal{L}_{RE}\left(e\right)$, $\mathcal{L}_{CF}\left(G\right)$ and $\mathcal{L}_{CS}\left(G'\right)$ the language defined by the RE e, the context free grammar G and the context sensitive grammar G', respectively.

3.1 Synchronized regular expression languages are context sensitive

To show that the languages in $L\left(SRE\right)$ are Context Sensitive [12] and that the membership problem for SRE is in NP [10] (result that will be later used in Sect. 4) we need to prove the following

Proposition 3. *The language defined by a SRE can be accepted by a nondeterministic Turing Machine in linear space and polynomial time w.r.t. the input size.*

Proof. Here, for *input size* we intend the length in characters of the string to be accepted or rejected.

For a fixed SRE e, an instance of our problem contains a string α. We have to find a nondeterministic Turing Machine M that, given the string α, accepts it if $\alpha\in\mathcal{L}_{SRE}\left(e\right)$ or rejects it if $\alpha\notin\mathcal{L}_{SRE}\left(e\right)$, using polynomial time and linear space for its computation, w.r.t. $|\alpha|$.

We define some constants to be used in the rest of the proof: N_v is the number of variables used in e; N_x is the number of exponents used in e; N_s is the number of "$+$" operators that appear in e. Observe that these are constants since e is fixed.

As usual, we see all the nondeterminism of the machine included in a preliminary step, called *guess*, where a nondeterministic black box makes all the choices about e, i.e. it selects a value for every exponent (we also see asterisks as nonsynchronized, fresh exponents), and makes a choice for each group of “$+$” operators in the expression (such operators are grouped so that, e.g. the expression $a + b + c$ is handled by a single choice).

Then, a deterministic Turing Machine is loaded with input:

- the string to accept or reject, α ;
- an encoding of all the decisions taken by the nondeterministic step, that is:
  - the binary-compressed number n_i associated with each distinct exponent x_i;
  - a number s_i used to resolve each group of “$+$” operators (see below).

The length of the guess is linear w.r.t. $|\alpha|$, since:

- the length of the exponent section is

$$L_x = \left(\sum_{i=1}^{N_x} |n_i| + N_x \right) \leq N_x \cdot (1 + log\,(|\alpha|)) \tag{2}$$

  since every exponent cannot have a value greater than $|\alpha|$. Moreover, if r_i is the number of times that the exponent x_i occurs on e, we have $\sum_{i=1}^{N_x} (n_i \cdot r_i) \leq |\alpha|$.
- the total length of the choices for the “$+$” operators is

$$L_s \leq |\alpha| \cdot (1 + log\,(|e|)) \tag{3}$$

  This bound can be explained as follows. Since “$+$” can be nested in (one or more levels of) exponentiated subexpressions, the number of operators involved is not directly N_s, the number of operators that appear in e. Indeed, after the exponent expansion, more “$+$” can appear. To encode the choices for these operators, we use a compressed integer s_i that means ”the s_i-th argument of the union is chosen”. For example, in $a+b+c$ the number $s_i = 2$ would choose b. Since there cannot be more arguments in a single sum than the total length of e, each compressed number s_i will be long at most $log\,(|e|)$. Finally, since each sum produces at least one character of the final string (an empty sum result means that a subexpression of the e is useless and can be ignored), there cannot be more than $|\alpha|$ sums in the expanded expression e. This explains the bound given above.

The machine M starts matching e with α using the information stored by the guess. The machine states encode the position of e that M is trying to match with the current character(s) of α read by the machine head. The following table shows a summary of the possible states that the machine could enter while matching e with α, and the actions taken each time. In the table, K stands for a fixed constant that can be easily deduced by the reader.

State	Looking for a letter b
Action	Accept if reading b, reject otherwise
Space	–
Time	1
State	Looking for a sum $(\beta_1 + \ldots + \beta_k)$
Action	Read the next sum guess s_i from the tape, move back on α and change state to look for β_{s_i}. If the number of guesses is insufficient, reject.
Space	–
Time	$K \cdot \lvert\alpha\rvert$ (move to find s_i and back on α)
State	Looking for a variable binding $(e_i)\,\%v_i$
Action	Change state to match e_i with the current substring of α. Reject if match fails, otherwise save the substring that matched e_i in a separate area of the tape reserved for v_i, move back on α and continue with the next subexpression of e.
Space	The total length of the variable section is $L_v = \left(\sum_{N_v}^{i=1} \lvert v_i\rvert + N_v \right) \leq N_v \cdot (1 + \lvert\alpha\rvert) \quad (4)$ since each variable cannot be assigned to a string longer than α.
Time	$K \cdot \lvert\alpha\rvert$ (substring copy)
State	Looking for a variable v_i
Action	Make a copy on the tape of the guess for v_i, and try to match it with the current substring of α with a character-by-character comparison. Reject if match fails, otherwise move back on α and continue with the next subexpression of e.
Space	$\lvert\alpha\rvert$ (variable copy, reusable)
Time	$\lvert\alpha\rvert^2$ (copy and comparison)

State	Looking for an exponentiated subexpression $(e')^{x_i}$
Action	Make a copy of the guess n_i for x_i. While $n_i > 0$, change state to match e' with the current substring of α. If the match succeeds, change state, decrement n_i and repeat, otherwise reject. Since there can be nested exponents, M will possibly maintain a list of counters for each nested loop.
Space	$N_x \cdot log\left(\lvert\alpha\rvert\right)$ (copies of each exponent guess at the highest nesting level)
Time	$K \cdot \lvert\alpha\rvert \cdot T\left(e'\right)$ (copy/decrement the counters, move back on α and match e')

State	The matching of e has ended.
Action	If reading past the last character of α accept, otherwise reject.
Space	–
Time	1

As the rows *Space* and *Time* show in the table, each step of M does not require more than linear space and polynomial time w.r.t. $|\alpha|$.

To sum up, the rules given will make the nondeterministic Turing Machine M to accept the string α iff $\alpha \in \mathcal{L}_{SRE}\left(e\right)$, using linear space and polynomial time. This proves that $L\left(SRE\right) \subseteq L\left(CS\right)$ (SRE are context sensitive) and that the membership problem for SRE is in NP, respectively.
□

In Sect. 4.1 we use the following more general result to prove that the membership algorithm for SRE is NP-Complete:

Corollary 1. *There is a nondeterministic Turing Machine M' that, given a SRE e and a string α, accepts the string if $\alpha \in \mathcal{L}_{SRE}\left(e\right)$ or rejects it if $\alpha \notin \mathcal{L}_{SRE}\left(e\right)$, using polynomial time w.r.t $|\alpha| + |e|$ and linear space w.r.t $|\alpha| \cdot log|e|$.*

Proof. We denote with $|e|$ the length in characters of the SRE e. This time the machine is not "expression-specific", (i.e. it is not designed for a specific SRE), but completely general. For this reason, the numbers N_v, N_x and N_s are no longer constants, but can be bounded by the input size:

$$\begin{aligned} N_v &\leq |e| \\ N_x &\leq |e| \\ N_s &\leq |e| \end{aligned} \tag{5}$$

Given this, we have the following new inequalities for the guess tape length:

$$L_e \leq log\,(\alpha) + |e| \tag{6}$$

$$L_x \leq |\alpha| + |e| \tag{7}$$

$$L_s \leq |\alpha| \cdot (1 + log|e|) \tag{8}$$

The dominant term for space is $|\alpha| \cdot log|e|$. By substituting the new expressions for N_v, N_x and N_s in the action table for M, we can actually see that the space remains bounded by $|\alpha| \cdot log|e|$. Since the input size is this time $|\alpha| + |e|$, we can say that the space requirement is subquadratic.

The time needed to accept or reject increases since the machine has to read e from the tape step by step (and possibly make copies of its parts), but this obviously does not take more than polynomial time. □

3.2 *Synchronized regular expressions and scattered context grammars*

Scattered Context Grammars (SC grammars from now on) belong to the family of *grammars with controlled derivations* [7]. Let us briefly recall their definition:

A SC grammar is a quadruple $G = (N, A, P, S)$, where:
- N, A and S are specified as in a context free grammar (that is they are, respectively, the alphabet of non-terminal symbols, the alphabet of terminal symbols and the start symbol), and
- P is a finite set of matrices

  $$(\xi_1 \to \gamma_1, \xi_2 \to \gamma_2, \ldots, \xi_k \to \gamma_k)$$

  where $k \geq 1$, $\xi_i \in N$, and $\gamma_i \in (N \cup A)^*$, for $1 \leq i \leq k$ (the number k can differ from matrix to matrix).

Given $\delta, \eta \in (N \cup A)^*$ we say that δ *directly derives* η if and only if:

$$\delta = \delta_1\xi_1\delta_2\xi_2 \ldots \delta_k\xi_k\delta_{k+1}, \text{ for some } \delta_i \in (N \cup A)^*, 1 \leq i \leq k+1$$
$$\eta = \delta_1\gamma_1\delta_2\gamma_2 \ldots \delta_k\gamma_k\delta_{k+1}$$
$$(\xi_1 \to \gamma_1, \xi_2 \to \gamma_2, \ldots, \xi_k \to \gamma_k) \in P$$

Then we can define in the usual way the notion: δ *derives* η *in zero or more steps* ($\delta \Rightarrow^* \eta$).

The language generated by a SC grammar G, denoted by as $\mathcal{L}_{SC}(G)$, is defined as

$$\mathcal{L}_{SC}(G) = \{\alpha \in A^* | S \Rightarrow^* \alpha\}$$

It is known that (see [7]):

- the languages of SC grammars *without erasing productions* are included in context sensitive languages;
- the languages of SC grammars *with erasing productions* coincide with that of recursively enumerable languages.

In the following we reserve the name SC for the *scattered context grammars without erasing productions.*

We introduce a proper subclass of SRE, namely the 1-level or "flat" SRE. 1-SRE are a yet useful but much less complex subclass of SRE that can be naturally placed in the hierarchy of grammars with controlled derivations as a proper subclass of SC grammars. We were not able to prove whether the same holds for general SRE.

Definition 9. 1-level SRE *(1-SRE) are SRE where variables and exponents cannot be nested (i.e., variables and exponents cannot appear inside an exponentiated expression or in the expression that is bound to a variable)*

Proposition 4. *Every 1-SRE language that does not contain the null string ϵ is in SC.*

Proof. SC are proven in [7] to be an extension of context free (CF) grammars, SRE are an extension of regular expressions (RE), and we already know that $L\left(RE\right) \subset L\left(CF\right)$. To shorten the exposition, we only give a sketch of the proof that a SC grammar can cope with the extensions of RE introduced in 1-SRE. These extensions are *variables* and *exponents*, which we shall call *synchronization elements*, used at the outermost (not nested) level.

A generic 1-SRE with only variables and exponentiated subexpressions can be written as follows:

$$e_1 e_2 \ldots e_k \text{ with } e_i = \begin{cases} {e'}_i^{\,x_i} \\ v_i \\ \left(e'_i\right)\%v_i \end{cases} \tag{9}$$

We know that some variables could be erased, by setting $v_i = \epsilon$ if $\epsilon \in \mathcal{L}_{RE}\left(e_i\right)$, and that all exponentiated expressions can be erased by setting $x_i = 0$. Actually, we cannot have erasing effects in our SC grammars, so we first rewrite the expression (9) as the union of several 1-SRE

$$\sum e_{i_1} e_{i_2} \ldots e_{i_l} \tag{10}$$

where each expression in the sum is obtained from (9) by deleting zero or more elements e_i, taking into account the synchronizations, and the sum is over all the possible combinations of element deletions.

The equivalence between the two languages of (9) and (10) is straightforward. Now we can impose that every synchronized element in (10) must not be erased (i.e., variables cannot be assigned to ϵ and exponents cannot be assigned to 0), since for every combination of erasable elements in (9) we have a subexpression in the disjunction (10) that exactly does not present those elements.

Given a generic element of the union (10) we start building the corresponding SC grammar with the rule:

$$S \to \xi_1 \xi_2 \dots \xi_l \tag{11}$$

Then, for each nonterminal ξ_i, we write a set of rules depending on the kind of the synchronization element e_i.

1. if $e_i \equiv {e'_i}^{x_i}$, suppose that this element is synchronized with m other elements $e_{j_1}, \dots, e_{j_m}$, that is $e_{j_1} \equiv {e'_{j_1}}^{x_i}, \dots, e_{j_m} \equiv {e'_{j_m}}^{x_i}$. Then we write the following grammar for the nonterminals $\xi_i, \xi_{j_1}, \dots, \xi_{j_m}$:

$$\begin{array}{l} (\xi_i \to \xi_i \xi'_i, \xi_{j_1} \to \xi_{j_1} \xi'_{j_1}, \dots, \xi_{j_m} \to \xi_{j_m} \xi'_{j_m}) \\ (\xi_i \to \xi'_i, \xi_{j_1} \to \xi'_{j_1}, \dots, \xi_{j_m} \to \xi'_{j_m}) \end{array} \tag{12}$$

   where each ξ'_i is the start symbol for the grammar that produces the language of e'_i, etc.
   Note that in 1-SRE these subexpressions are always standard RE so we assume there is a CF grammar that produces their language. The parallelism in the grammar given in (12) forces the synchronization between the number of repetitions of $e'_i, e'_{j_1}, \dots, e'_{j_m}$, that must be greater than zero.
2. if $e_i \equiv (e'_i)\,\%v_i$, suppose that this element is synchronized with m other elements $e_{j_1}, \dots, e_{j_m}$, that is $e_{j_1} \equiv \dots \equiv e_{j_m} \equiv v_i$. Since in a 1-SRE e'_i must be a simple regular expression, we know that we have a CF grammar G that produces the language of this subexpression. Since, by our assumptions, the grammar cannot contain null productions (that are considered separately), we first rewrite the grammar to prevent it from producing the empty string.
   Suppose that the grammar G has start symbol S', is composed by a set of productions $P = \{R_0, \dots, R_k\}$ and that its nonterminals are fresh. Then we write the following grammar for the nonterminals $\xi_i, \xi_{j_1}, \dots, \xi_{j_m}$:

$$\begin{array}{l} (\xi_i \to S', \xi_{j_1} \to S', \dots, \xi_{j_m} \to S') \\ \underbrace{(R_0, R_0, \dots, R_0)}_{\text{m times}} \\ \vdots \\ \underbrace{(R_k, R_k, \dots, R_k)}_{\text{m times}} \end{array} \tag{13}$$

   The parallelism in the SC grammar forces the synchronization between the contents of the synchronized variables, since their productions can be applied only to all the variables at the same time. Note that this is true due to the fact that the language of each ${e'}_i$ is regular and so the grammar associated to each ξ_i is in turn regular.

Finally, we write the grammar for the entire union (10) as the (finite) union of all the languages of the corresponding subexpressions, built as explained above. □

3.3 Synchronized regular expressions do not generate all context free languages

We show that the languages generated by the Synchronized Regular Expressions on an alphabet *with more than one letter* do not contain all the context free languages on the same alphabet. To this aim, we make use of the language of palindromes. It is known that this language is context free: for example, it can be generated by the simple grammar:

$$S \rightarrow \epsilon\,,\ S \rightarrow aSa,\ \forall a \in A$$

In the proof of the following proposition we need a complexity measure on subexpressions of SRE.

Definition 10. *Let* e *be a SRE and* e' *be a subexpression of* e *(i.e.* $e = e_1e'e_2$*, where* e_1, e_2 *are possibly empty). The* weight *of* e' *is defined as the sum of the number of exponents and variable bindings it contains, plus the weight of all the subexpressions of* e *bound to variables backreferenced in* e'.

Observe that the previous recursive definition is correct due to the restriction on backreferences given in definition 6.

Proposition 5. *No Synchronized Regular Expression can generate the language of palindromes on an alphabet* A *with* $|A| > 1$.

Proof. It is known that, if A contains at least two letters, there exist words of any length in A^* that do not contain cubes [5]. This implies that there are also palindromes of any length that can be split in two halves each of which does not contain cubes.

In this proof we show that if a SRE e produces only *palindromes*, then there exists a length l s.t. *all* the palindromes produced by e longer than l contain cubes in one of their halves. This implies that there is no SRE that can generate the language of all palindrome strings.

Suppose there exists a SRE e that generates infinite palindromes without cubes. Thus, we can say that there exists a subexpression e' of e that generates infinite words without cubes (possibly $e' = e$).

Let $e'\,[v_1,\ldots,v_n]$ be the minimal (w.r.t. Definition 10) subexpression of e that generates infinite words without cubes. Let the bindings for the variables backreferenced in e' be $(e'_i)\,\%v_i$ with $i = 1 \ldots n$. Note that these bindings can appear both *inside* or *outside* e'.

Each e'_i has a weight smaller than e', since by Definition 10 an expression weight sums all the weights of the backreferenced variable bindings.

Thus none of the e'_i can produce infinite words without cubes, because this would contradict the hypothesis that e' is the minimal subexpression with this property. This implies that each e'_i can only produce a finite set B_i of words without cubes. Let $B_1,\ldots,B_n$ be an enumeration of all such B_i.

We make all the possible variable substitutions in e' with the values taken from the sets $B_1,\ldots,B_n$, thus obtaining a (finite) set of SRE of the form:

$$e'\left[{}^{\alpha_1}/_{v_1},\ldots,{}^{\alpha_n}/_{v_n}\right]\forall\alpha_i \in B_i \tag{14}$$

(where the notation ${}^{\alpha_i}/_{v_i}$ means that the variable v_i has been replaced by the word α_i) each of these SRE obviously maintains the property of generating infinite words without cubes.

Let e'' be one of the SRE obtained from the substitution above: the number of backreferences used in e'' is smaller than those used in e', and in particular e'' does not contain any backreference to the outer expression, so it may again be considered a "standalone" SRE that generates infinite words without cubes.

If e'' contains variables, we can reiterate the process above, find its minimal subexpression with the same property, instantiate all the backreferenced variables, etc. On every iteration we decrease the number of variables in the generated expressions, so finally we obtain a set of expressions without variables each of which can generate infinite words without cubes. Let us consider one of these expressions and call it $\bar{e}$.

The only way $\bar{e}$ has to generate words of any length is to use exponents (and stars, but we may consider them as unsynchronized exponents) to replicate one or more of its subexpressions.

Let $(\bar{\bar{e}})^x$ be the minimal exponentiated subexpression of $\bar{e}$ (w.r.t. the number of exponents that appear in the expression) that produces infinite words without cubes.

We know that $\bar{\bar{e}}$ alone cannot produce infinite words without cubes, because it has one exponent less than $(\bar{\bar{e}})^x$ and this would contradict the hypothesis that $(\bar{\bar{e}})^x$ is minimal. But this also implies that $\bar{\bar{e}}$ cannot contain exponents at all, because using exponents it would produce infinite words with cubes, and so would do $(\bar{\bar{e}})^x$ in contradiction with the hypothesis.

Thus we can say that $\bar{\bar{e}}$ only produces a finite number of different words, that is we can rewrite it as a finite sum of words. We have that:

$$(\bar{\bar{e}})^x = (\alpha_1 + \ldots + \alpha_n)^x \tag{15}$$

Now we go back to the initial expression e. We found that if we instantiate all the variables and all the exponents except one we end with an expression of the form:

$$\alpha\left(\alpha_1 + \ldots + \alpha_n\right)^x \gamma\beta \tag{16}$$

We suppose for generality that the subexpression $\left(\alpha_1, \ldots, \alpha_n\right)^x$ is backreferenced one time in e. If there is no backreference, the rest of the proof is trivial. If there is more than one backreference, the proof we show below can be applied as well. Moreover, to simplify the proof, we set $n = 2$ in the subexpression and call $\alpha_1 = X$ and $\alpha_2 = Z$. So the general expression we have is:

$$\alpha\left(\left(X + Z\right)^x\right)\%v\gamma v\beta \tag{17}$$

by hypothesis this expression must produce infinite palindromes without cubes. Before we give the final section of the proof, we need to prove the following property:

Claim. For every $X, Y, \alpha \in A^*$ with $X \neq Y$, $X\alpha Y = Y\alpha X \Rightarrow \exists\beta, \gamma \in A^*,\ n, m, p \in \mathbb{N}$ s.t. $X = (\beta\gamma)^n\beta, Y = (\beta\gamma)^m\beta, \alpha = (\gamma\beta)^p\gamma$.

Proof. We proceed using induction on the length of $X\alpha Y$. If one of the three strings is empty, then our thesis follows from the Defect Theorem [14]. Thus, the minimal length is 3 and the base of our induction is $X = a, Y = b, \alpha = c$ for some $a, b, c \in A$. But if $acb = bca$ then $a = b$, and the equations in our thesis are satisfied choosing $\beta = a, \gamma = c, n = m = p = 0$.

Now, for the inductive step, suppose $|Y| > |X| > 1$ (the case with $|X| > |Y| > 1$ is equivalent). Then from

$$X\alpha Y = Y\alpha X \tag{18}$$

follows that $Y = X\delta X$.

By substituting the value for Y in (18) and simplifying the resulting expression we obtain:

$$\alpha X\delta = \delta X\alpha \tag{19}$$

since $|\delta| < |Y|$ this expression is shorter than (18), so we can apply the inductive hypothesis. The values for α, X, δ in (19) follow from our thesis:

$$\begin{aligned} \alpha &= (\beta\gamma)^n\beta \\ \delta &= (\beta\gamma)^m\beta \\ X &= (\gamma\beta)^p\gamma \end{aligned} \tag{20}$$

By substituting these values in the equation for Y we obtain that the values for X, α, Y in (18) are:

$$\begin{aligned} &X = (\gamma\beta)^p\gamma \\ &Y = X\delta X = (\gamma\beta)^p\gamma(\beta\gamma)^m\beta(\gamma\beta)^p\gamma = (\gamma\beta)^{2p+m+1}\gamma \\ &\alpha = (\beta\gamma)^n\beta \end{aligned} \tag{21}$$

which is what we wanted to prove. □

Now we are back in the main proof. We have the following expression:

$$\alpha\left((X+Z)^x\right)\%v\gamma v\delta \tag{22}$$

and we know that is must produce infinite palindromes. To further simplify the equations, suppose $|\alpha| < |\delta|$, so since the expression above is a palindrome we have that $\delta = \beta\alpha^{-1}$ (where α^{-1} is α reversed). In the other cases the proof is still the same or simpler.
We can rewrite the expression (22) as:

$$\left((X+Z)^x\right)\%v\gamma v\beta \tag{23}$$

Suppose that $|X| > \beta$, $|Z| > \beta$, $|\gamma| > \beta$ (a proof can be given also in the other cases):

- For $x = 1$, the expression $X\gamma X\beta$ must be a palindrome. So $X = \beta^{-1}Y$ and we obtain the palindrome $Y\gamma\beta^{-1}Y$. This implies that Y and $\gamma\beta^{-1}$ must also be palindromes. We can also write $\gamma = \beta\alpha$ with α palindrome.
- Since $Z\gamma Z\beta$ is a palindrome we have in the same way that $Z = \beta^{-1}W$ with W palindrome.
- Another palindrome is $XX\gamma XX\beta$. This can be rewritten using the substitutions obtained in the previous points as $\beta^{-1}Y\beta^{-1}Y\gamma\beta^{-1}Y\beta^{-1}Y\beta$. Since we know that Y and $\gamma\beta^{-1}$ are palindromes, we derive that also β is a palindrome.
- Finally, consider the palindrome $XZ\gamma XZ\beta$. By substituting the values found for X, Z and γ we have the palindrome $\beta^{-1}Y\beta^{-1}W\beta\alpha\beta^{-1}Y\beta^{-1}W\beta$. This implies that $Y\beta^{-1}W$ must be a palindrome, that is $Y\beta^{-1}W = \left(Y\beta^{-1}W\right)^{-1} = W^{-1}\beta Y^{-1} = W\beta^{-1}Y$.

By applying Claim 3.3 to the last expression we have that, for some $\bar{\beta}, \bar{\gamma} \in A^*$ and $n, m, p \in \mathbb{N}$:

$$\begin{array}{l} Y = \left(\bar{\beta}\bar{\gamma}\right)^n \bar{\beta} \\ W = \left(\bar{\beta}\bar{\gamma}\right)^m \bar{\beta} \\ \beta^{-1} = \left(\bar{\gamma}\bar{\beta}\right)^p \bar{\gamma} \end{array} \tag{24}$$

From these values we obtain the values for X and Z in (23).

$$\begin{array}{l} X = \beta^{-1}Y = \left(\bar{\gamma}\bar{\beta}\right)^p \bar{\gamma}\left(\bar{\beta}\bar{\gamma}\right)^n \bar{\beta} = (\gamma\beta)^{p+n+1} \\ Z = \beta^{-1}W = \left(\bar{\gamma}\bar{\beta}\right)^p \bar{\gamma}\left(\bar{\beta}\bar{\gamma}\right)^m \bar{\beta} = (\gamma\beta)^{p+m+1} \end{array} \tag{25}$$

We can finally rewrite the expression (23) as

$$\alpha\left(\left((\gamma\beta)^{p+n+1} + (\gamma\beta)^{p+m+1}\right)^x\right)\%v\gamma v\beta \tag{26}$$

Remark 4. If we supposed to have more than two expressions in the disjunction (22), then we could apply the method above using as X and Z all the possible pairs of expressions in the sum. For example, if $n = 3$ from the sum $(X + Z + Y)$ we obtain
$X = (\gamma_1\beta_1)^{p_1+n_1+1}$, $Z = (\gamma_1\beta_1)^{p_1+m_1+1}$ for the pair (X, Z) and
$Z = (\gamma_2\beta_2)^{p_2+n_2+1}$, $Y = (\gamma_2\beta_2)^{p_2+m_2+1}$ for the pair (Z, Y).

This implies that $Z = (\gamma_2\beta_2)^{p_2+n_2+1} = (\gamma_1\beta_1)^{p_1+m_1+1}$ and for the Defect Theorem [14] we have that $(\gamma_2\beta_2)^{p_2+n_2+1} = (\eta)^k = (\gamma_1\beta_1)^{p_1+m_1+1}$. By applying this method to all the pairs, we end with all the expressions reduced to powers of a same common subexpression, as in the previous case.

The expression (26) must produce infinite palindromes whose halves do not contain cubes. But the only part of the expression that can be used to produce longer words is the exponent x, and applying an exponent greater than three to x leads to words with squares and cubes. Since α is fixed, we know that there is a value l s.t. for all $x > l$ a cube appears in the first half of the produced palindrome. This contradicts our hypothesis.
Thus a SRE that produces the language of all palindromes does not exist. □

4 Complexity of membership algorithm for SRE

Here we show the complexity of the membership test for SRE, and look at some restrictions that may lower this complexity while leaving enough expressiveness to the language.

4.1 Membership for SRE is NP-complete

We already proved (in Sect. 3.1) that the membership problem for SRE is in NP. Moreover, the membership problem for Regular Expressions with backreferences has already been proved to be NP-Complete [1], and this obviously extends also to our SRE. By *membership problem* we mean the *uniform membership problem.* Such problem can be specified as follows: given a SRE e and a string α, decide whether $\alpha \in \mathcal{L}_{SRE}(e)$. On the other hand, the *non uniform membership problem* has the expression e fixed once for all and has always polynomial complexity (see also Sect. 5).

However, we want to prove that even for SRE *without backreferences*, that is only with synchronized exponents, the membership is NP-Complete. We proceed by reducing the well-known 3-CNF problem (satisfiability of boolean expressions in conjunctive normal form with three literals per clause) to the SRE membership problem through a polynomial transformation. To do this, we use Synchronized Regular Expressions with exponents synchronized several times.

It is natural to ask whether the complexity could be reduced if each exponent is repeated only twice, i.e. only one synchronization per exponent is allowed. We answered this question in the negative by showing that even if only one synchronization per exponent is allowed, the membership problem remains NP-Complete. This means, in a sense, that there is no complexity growth if the number of synchronizations per exponent in the expression increases.

Proposition 6. *The 3-CNF problem can be reduced to the membership problem for SRE without backreferences (with exponents only).*

Proof. Let F be a boolean formula in 3-conjunctive normal form.
Let $C_1, \dots, C_n$ be its clauses, and $l_1, \dots, l_m$ be its variables. Each clause C_i has the form $(L_{i_1} \vee L_{i_2} \vee L_{i_3})$ where L_{i_j} can be l_k or $\bar{l_k}$, for some $k \in \{1, \dots, m\}$.

We assign an exponent of our pattern x_i to each boolean variable l_i, so that

$$x_i = 1 \Leftrightarrow l_i = \text{false}$$
$$x_i = 0 \Leftrightarrow l_i = \text{true}$$

In the following we will build step-by-step a SRE, that we shall call *pattern*, and a *string*. We say that the pattern *matches* the string iff the string is in the language generated by the pattern.

First of all, we need the "negation" $\bar{x}_i$ of each exponent x_i, i.e. the exponent that corresponds to the negation of the associated variable. To get it, we use the following pattern - string pair:

Pattern: $NP_i = a^{x_i} a^{\bar{x}_i} b$
String: $NS = ab$

that also constrains our exponents in the range $\{0, 1\}$. If the pattern matches the string, then either x_i or $\bar{x}_i$ is 1 and the other exponent is 0.

For each clause $C_i = (L_{i_1} \vee L_{i_2} \vee L_{i_3})$, we write the corresponding pattern

$$P_i = a^{x_{i_1}} \, a^{x_{i_2}} \, a^{x_{i_3}} \, a^* \tag{27}$$

where

$$x_{i_j} = \begin{cases} x_k \text{ if } L_{i_1} = l_k \\ \bar{x}_k \text{ if } L_{i_1} = \bar{l_k} \end{cases} \tag{28}$$

and the string

$$S_i = aa \tag{29}$$

Now by concatenating all clause-patterns, using b as a separator, we obtain the following formula-pattern:

$$FP = P_1 \, b \, P_2 \, b \, \cdots \, b P_n \tag{30}$$

The corresponding string is

$$FS = \underbrace{aab\,aab\,\cdots\,aab}_{n-\text{times}} \tag{31}$$

It is easy to see that the pattern FP matches the string FS if and only if each clause-pattern P_i matches the string aa. This can happen if and only if at least one of the three exponents $x_{i_1}, x_{i_2}, x_{i_3}$ is zero, so that the pattern produces less than three a (the a^* that terminates each clause-pattern is used to clear all the remaining a in the string).

Let us suppose that $x_{i_j} = 0$. By hypothesis, this means that

- if $x_{i_j} = x_k$, then $C_i = (\ldots \vee l_k \vee \ldots)$. But $x_k = 0 \Rightarrow l_k = \text{true}$. Therefore $C_i = \text{true}$.
- if $x_{i_j} = \bar{x}_k$, then $C_i = \left(\ldots \vee \bar{l_k} \vee \ldots\right)$. But $\bar{x}_k = 0 \Rightarrow \bar{l_k} = \text{true}$. Therefore $C_i = \text{true}$.

If every pattern P_i matches the string aa, then there is at least one true boolean literal in each clause C_i, which is also true, so that the whole boolean formula is satisfied.

Finally, to obtain the complete pattern and target string, we append the negation-patterns and strings as follows :

$$\begin{aligned} P &= FP\,NP_1\,\ldots\,NP_m \\ S &= FS\,\underbrace{NS\,\ldots\,NS}_{m-\text{times}} \end{aligned} \tag{32}$$

Here a successful pattern matching ensures that

- every $\bar{x}_k$ is the negation of x_k (in the sense we described above), and both range in $\{0, 1\}$.
- the boolean variable assignment on $l_1, \ldots, l_m$ corresponding to the values of $x_1, \ldots, x_m$ makes true every clause C_i.

This implies that F is satisfied by the variable assignment determined by a pattern matching of P on S.

To measure the complexity of the transformation from the boolean formula F to the (pattern, string) pair (P, S), we consider both the output string and pattern as character sequences and recall here how many characters are written by each step of the above generation process. We also consider exponents on letters like extra characters (each exponent is one character long) in the sequence.

- The creation of negated exponents takes 5 characters of pattern and 2 characters of string for each boolean variable in the formula.
- The creation of clause-patterns takes 8 characters of pattern and 2 characters of string for each clause.

Thus the full pattern (including separators) takes 9 characters for each clause and 5 for each variable, while the full string takes 3 characters for each clause and 2 for each variable.

We can roughly give an upper bound of the transformation complexity saying that the output is $20n$ characters long, where n is the number of characters the string representation of the input formula F (that is obviously greater than the sum $number\_of\_variables + number\_of\_clauses$).

So we can reduce the 3-CNF-SAT problem to a membership problem with synchronized exponents with a polynomial transformation. Since the 3-CNF-SAT problem is known to be NP-complete, our membership problem is also NP-complete. □

In the next subsection we show that two or more different exponents ranging in $\{0, 1\}$ can be synchronized without the need of *explicit synchronization* (i.e. not using the same exponent in all cases). Using this technique, we may rewrite the proof above using only pairs of explicit exponent synchronizations (i.e. the same exponent is used only twice). Since the transformation is polynomial, we ensure that the proof given above is valid regardless of the number of synchronizations used. Therefore, we have the following:

Corollary 2. *The membership problem for SRE without backreferences is NP-Complete.*

4.2 Synchronizing the value of different exponents

Here we show a simple method that forces the synchronization of *different* exponents ranging in $\{0, 1\}$. For a more detailed overview on the pattern synchronization techniques used in this section, the reader may refer to [16].

Let $x, x_1, \ldots, x_n$ be *distinct* exponents with $x \in \{0, 1\}$. We want to synchronize the values of $x_1, \ldots, x_n$ with that of x.

We can accomplish this with a pattern matching that uses these exponents but *makes no use of explicit synchronization* and an alphabet of three letters.

We build a pattern in the following way:

$$b^* c^* a^x [b^* a^{x_i}]_{i=1\ldots n} b^* c a^* \underbrace{[b^* a^*]}_{n-\text{times}} b^* c^* \tag{33}$$

The notation d^* stands for d^y where y is a fresh variable (that is, we use the star as a generic exponent *not synchronized*).

The string that the pattern should match is as follows:

$$bca \underbrace{[ba]}_{n-\text{times}} bc \tag{34}$$

For example, for $n = 2$ we have:

Pattern: $b^*c^*a^xb^*a^{x_1}b^*a^{x_2}b^*ca^*b^*a^*b^*a^*b^*c^*$
String: $bcabababc$

We state that the pattern matching between the given pattern and string synchronizes all the variables of the pattern to the same value. The proof is trivial and below we only sketch its basic steps.

First of all we note that the pattern contains a constant (without exponent) c. Such a letter must match one of the two c's in the target string.

1. If the constant c matches the first c in the string, then the preceding part of the pattern must produce a single b. This implies that $x, x_1, \ldots, x_n$ are all set to zero. It is then easy to see that the remaining part of the pattern matches the remaining part of the string:

$$\begin{array}{cccccccccccccccc} b^* & c^* & a^x & b^* & a^{x_1} & b^* & a^{x_2} & b^* & \mathbf{c} & a^* & b^* & a^* & b^* & a^* & b^* & c^* \\ & & & & & & & b & \mathbf{c} & a & b & a & b & a & b & c \end{array} \tag{35}$$

2. If the constant c matches the second c of the target string, then all the string must be produced by the pattern $b^*c^*a^x[b^*a^{x_i}]_{i=1\ldots n}b^*c$. This implies that the fragment b^*c^* must match bc, since it is the only part of the pattern that can produce another c, and the pattern $a^x[b^*a^{x_i}]_{i=1\ldots n}b^*$ must match the string $a \underbrace{[ba]}_{n-\text{times}} b$. This is true if and only if $x, x_1, \ldots, x_n$ are all set to one:

$$\begin{array}{cccccccccccccccc} b^* & c^* & a^x & b^* & a^{x_1} & b^* & a^{x_2} & b^* & \mathbf{c} & a^* & b^* & a^* & b^* & a^* & b^* & c^* \\ b & c & a & b & a & b & a & b & \mathbf{c} & & & & & & & \end{array} \tag{36}$$

So we know that the pattern can match the string in two ways only. In both ways the variables $x, x_1, \ldots, x_n$ have all the same value, zero or one. Once the value of x has been set, then there is only one possible match that sets all the other exponents $x_1, \ldots, x_n$ to the same value of x.

4.3 Synchronizing groups of different exponents in the same expression

Finally, we show how to extend the technique described in the previous section to synchronize several groups of different exponents in the same pattern. In Proposition 6, these groups are the (different) exponents used to represent the same boolean variable in each clause.

Let $\left(x_1, x_{1_1}, \ldots, x_{1_{n_1}}\right) \ldots \left(x_m, x_{m_1}, \ldots, x_{m_{n_m}}\right)$ with $m > 1$ be exponents, and $x_j \in \{0, 1\}$, $\forall j \leq m$. We want to synchronize all the exponents in each group $x_j, x_{j_1}, \ldots, x_{j_{n_j}}$ using a single SRE.

A simple approach is to introduce a new letter (d in the following examples) and use it as a separator between synchronization subpatterns created as explained in the previous section.

Let P_j, S_j be the pattern and the string used to synchronize the exponents $x_j, x_{j_1}, \ldots, x_{j_{n_j}}$ $\forall j \leq m$; we build the composite pattern:

$$P_1 d P_2 d \ldots P_m \tag{37}$$

In the same way we build the composite target string:

$$S_1 d S_2 d \ldots S_m \tag{38}$$

The correctness proof of this approach is simple. Since there are exactly $m-1$ occurrences of the constant d in the pattern, and only $m-1$ occurrences of d in the target string, the only possible match is the one that makes them correspond, forcing each sub-pattern P_j to match with the sub-string S_j and thus obtaining the requested synchronization.

A refinement of this approach allows us to avoid the introduction of a new letter in the pattern. Since in P_j letters a, b, c appear in a small number of configurations, we can use a combination that does not conflict with the synchronization patterns as a separator.

If we use the string bb as a separator, then the resulting pattern and string are:

Pattern: $P_1 bb P_2 bb \ldots P_m$
String: $S_1 bb S_2 bb \ldots S_m$

For example, for $m = 2, n_1 = n_2 = 2$ we have:

Pattern: $b^* c^* a^{x_1} b^* a^{x_{1_1}} b^* a^{x_{1_2}} b^* c a^* b^* a^* b^* a^* b^* c^*$
$bb\ b^* c^* a^{x_2} b^* a^{x_{2_1}} b^* a^{x_{2_2}} b^* c a^* b^* a^* b^* a^* b^* c^*$
String: $bcabababc\ bb\ bcabababc$

It is clear that the constant substring bb in the pattern must match one of the two identical (overlapping) sequences in the string. However, if the pattern bb matches the second sequence (40), then P_1 must match $bcabababc\ b$, and this is impossible since after producing the second c, P_1 cannot produce anything else, while the string has another b to match.

The only possible match must align with the separator bb in the same way it does with the separator d (39), so the synchronization is correct.

$$\begin{array}{ccccccccccccccc} \cdots & a^* & b^* & a^* & b^* & c^* & \mathbf{b} & \mathbf{b} & b^* & c^* & a^{x_2} & b^* & a^{x_{2_1}} & & \cdots \\ \cdots\ \cdots & b & a & b & c & \mathbf{b} & \mathbf{b} & \mathbf{b} & c & a & b & a & & b & \cdots \end{array} \tag{39}$$

$$\begin{array}{cccccccccccccc} \cdots & a^* & b^* & a^* & b^* & c^* & \mathbf{b} & \mathbf{b} & b^* & c^* & a^{x_2} & b^* & a^{x_{2_1}} & \cdots \\ \cdots\ b & a & b & c & \mathbf{b} & \mathbf{b} & \mathbf{b} & c & a & b & a & b & & \cdots \end{array} \tag{40}$$

The previous discussion shows that

Proposition 7. *The membership problem for SRE that*

- *do not contain backreferences, and*
- *contain exponents synchronized at most twice,*

is NP-Complete.

Using the same technique, we can also improve a result of [2] as follows:

Proposition 8. *The membership problem for SRE that*

- *do not contain exponents,*
- *contain each backreference at most twice (including the binding occurrence), and*
- *the SRE backreferenced is always A^*,*

is NP-Complete.

We omit the proof for brevity, since it is similar to the proof of Proposition 7. Observe that, by the third hypothesis, variables behave as those in [2].

5 Synchronized regular expressions with limited synchronization elements

In Sect. 4 we proved that the general pattern matching problem with SRE is NP-Complete even with an alphabet of only two symbols or when we limit the number of synchronizations for each variable or exponent. This is true for expressions containing variables, exponents or both.

However, examples of SRE like those in Sect. 6 show that, in real applications, the number of synchronization elements used in a single expression is often very small. Typical applications can safely fix a limit on the number of variables and/or exponents that the user can write in each expression. This limit is often suggested by the application domain itself.

When the number of synchronized variables and exponents in the expression is constrained (but not the number of times each of these can be used), the complexity of the SRE membership problem becomes polynomial. This has already been stated for backreferences only [1]. Here we expand the proof by extending it to exponent synchronization.

Remark 5. When we state that the number of synchronized elements is fixed, we should also take into account the nested expressions. That is, for example, an exponentiated variable $((v)^x)$ counts as two elements.

5.1 A simple polynomial algorithm for SRE pattern matching

This algorithm has some similarities with the dynamic programming, but is actually enumerative. Likewise dynamic programming algorithms, we have a notion of *state*. Our states are tuples of the form

$(pattern, string, assignments)$ that exhibit the pattern, the string it must match and the assignments done to variables and exponents so far, respectively. The initial state s_0 contains the pattern and string given to the algorithm by the user, and the assignments are empty.

For convenience, we view patterns split in *tokens*. Looking at a pattern from left to right, tokens are the longest subpatterns of the following kinds (the first rule has the highest precedence):

1. an exponentiated subpattern
2. a variable (binding or backreference)
3. a subpattern without any exponent or variable (a standard RE).

Let S_i be the set of states at the ith iteration of the algorithm. Initially $S_0 = \{s_0\}$. At every iteration i the algorithm considers the leftmost token in the pattern for each state in S_{i-1} and

1. if the token cannot match a prefix of the associated string, it does nothing;
2. if the token can match a prefix of the associated string in one or more ways, it tries all the possible matches; every match creates a new state in S_i where the used token and possibly the matched string prefix are changed, and all the assignments done to variables and exponents are updated.

Thus, the algorithm carries out "in parallel" all the possible matches.

When we say that we try every possible match between a token and a string we mean the following:

1. A variable binding can match every string prefix that its associated SRE can. The algorithm continues matching the SRE subexpression recursively. When the end of the subexpression is reached, the assignments of each reached state are updated by adding a new binding for the variable to the matched prefix.
2. An exponentiated expression whose exponent is not bound can match the string for every value of the exponent in $[0 \ldots n]$, where n is the length of the string. This rule will generate a new set of states where the exponentiated expression is substituted by n repetitions of the expression itself (without exponent), for $n \in [0 \ldots n]$. The assignments of each state are updated by adding a new binding for the exponent to n.
3. A standard RE can also have different matches, when the star and the '+' character are used. This rule will generate a new set of states where the RE subexpression is deleted from the pattern and the matched prefix is deleted from the string.
4. A backreference acts like a constant RE.
5. An exponentiated expression whose exponent is already bound to a number acts like the specified number of repetitions of the expression (without exponent).

The algorithm stops when it generates the empty state (string and pattern are empty), meaning that the match is successful, or when the new state set is empty, i.e. the match cannot be carried out in any way.

Example 5. We want to find if the SRE $ab(bc)^x(d+b)\%v(b+c)va^x$ can generate the string $abbcdbda$.

The initial state set is $S_0 = \left\{ \begin{bmatrix} ab(bc)^x(d+b)\%v(b+c)va^x \\ abbcdbda \\ \{\} \end{bmatrix} \right\}$

The pattern tokens are

$$ab, \quad (bc)^x, \quad (d+b)\%v, \quad (b+c), \quad v, \quad a^x$$

Step 1. The leftmost token in the first state is ab, that can match the string prefix in only one way. So, the new state set is

$$S_1 = \left\{ \begin{bmatrix} (bc)^x(d+b)\%v(b+c)va^x \\ bcdbda \\ \{\} \end{bmatrix} \right\}$$

Step 2. The leftmost token now is $(bc)^x$. It can match the string prefix in two ways, that is for $x=0$ and $x=1$ (the general algorithm says that we should try with every x from zero to six, but we cut the search tree to have a smaller example). So our new state set contains two possible states:

$$S_2 = \left\{ \begin{bmatrix} (d+b)\%v(b+c)va^x \\ bcdbda \\ \{x=0\} \end{bmatrix}, \begin{bmatrix} bc(d+b)\%v(b+c)va^x \\ bcdbda \\ \{x=1\} \end{bmatrix} \right\}$$

Step 3. The leftmost token in the first state of S_2 is the variable binding $(d+b)\%v$. The algorithm continues matching the SRE $d+b$. This produces only one state, where the binding for v is updated:

$$S_{3_1} = \left\{ \begin{bmatrix} (b+c)va^x \\ cdbda \\ \{x=0, v=b\} \end{bmatrix} \right\}$$

the second state of S_2 has a constant RE as its leftmost token, so it can have only one match

$$S_{3_2} = \left\{ \begin{bmatrix} (d+b)\%v(b+c)va^x \\ dbda \\ \{x=1\} \end{bmatrix} \right\}$$

$S_3 = S_{3_1} \cup S_{3_2}$ contains 2 states.

Step 4. The first state of S_3, whose leftmost token is the RE $(b + c)$ can continue the match and produce a new state:

$$S_{4_1} = \left\{ \begin{bmatrix} va^x \\ dbda \\ \{x = 0, v = b\} \end{bmatrix} \right\}$$

the second state of S_3 has a variable binding $(d+b)\%v$ as its leftmost token. Again, the algorithm proceeds by matching $d+b$, and this produces one new state only, where the binding of v is also updated:

$$S_{4_2} = \left\{ \begin{bmatrix} (b + c)va^x \\ bda \\ \{x = 1, v = d\} \end{bmatrix} \right\}$$

$S_4 = S_{4_1} \cup S_{4_2}$ contains 2 states.

Step 5. The first state in S_4, whose leftmost token is the already assigned variable v, cannot continue the match, since its string has a prefix that does not match with the fixed value of v.

The remaining state of S_4 with leftmost token $(b + c)$ can continue the match and produce the new state:

$$S_5 = \left\{ \begin{bmatrix} va^x \\ da \\ \{x = 1, v = d\} \end{bmatrix} \right\}$$

Step 6. Now the only state in S_5 has leftmost token v and can continue the match. Since $v = d$ we obtain the new state:

$$\begin{bmatrix} a^x \\ a \\ \{x = 1, v = d\} \end{bmatrix}$$

Step 7. Finally, since $x = 1$ we obtain:

$$\begin{bmatrix} a \\ a \\ \{x = 1, v = d\} \end{bmatrix}$$

which, in the final step, produces the empty state. So our match was successful.

5.2 Complexity of the algorithm

The complexity of this algorithm is simple to express: if n is the length of the target string and m is the length (in characters) of the pattern, we can make a worst-case evaluation in the following way:

- a variable binding is matched as a normal SRE, and a variable backreference is matched as a constant string.
- an exponentiated subexpression can have n different matches (the exponent ranges in $[0 \ldots n]$ since a non empty subpattern, repeated more than n times, would exceed the length of the string).

  Note 1. The exponentiated and variable-bound subexpressions are not requested to be simple RE: they may be another SRE, i.e. they may contain other variables and/or exponents. However, since we stated that the *overall* number of synchronized elements is fixed, also these variables and/or exponents were counted.
  Moreover, it is correct to count nested synchronization elements only once, even if they are nested in an exponentiated expression: actually, expanding such an expression will introduce only one new synchronization element, repeated for a certain number of times.

- all the other RE subexpressions can be matched in linear time, using well-known methods.

If we fix the number of possible synchronizations (both with variables and exponents), say k, we have that the number of states generated by synchronization tokens is at most n^k. All the other RE matches can be considered linear w.r.t. $n \cdot m$, so the complexity of our algorithm is $O\left(m \cdot n^k\right)$, a polynomial with a degree equal to the limit k of synchronizations.

We can summarize the results of this section in the following proposition:

Proposition 9. *The membership problem for SRE with a limited number of synchronization elements (i.e. less or equal to a fixed number k) can be solved in polynomial time $O\left(m \cdot n^k\right)$, where n is the size, in characters, of the target string and m is the size, in characters, of the pattern to match.*

6 A user-friendly syntax for synchronized regular expressions

The second aim of this paper is to provide a common syntax for SRE extensions to be used in implementations such as text editors, command line search utilities like `grep` [15], etc. In the rest of this section, the word *syntax* will denote the syntax used to write SRE on a computer terminal, that is obviously a little different from the formal syntax introduced in Sect. 2.

Our idea is to let the user access the power of SRE at various levels, since we observed that, even if backreferences and exponents are useful to all classes of users, the beginner user usually applies them to solve a limited number of common problems. In these cases the fully-general syntax may be excessive and useless. Instead, we introduce other constructs that accomplish these common tasks acting as macros (i.e., shortcuts, that can be expanded into SRE syntax).

Let us first introduce the full syntax for SRE. We inherit the common syntax used for RE and add the following constructs:

- `/(e)var/` is a binding for variable named var to the SRE e.
- `/var/` is a backreference for variable called var.
- $\{$`id`$\}$ on the right of any expression binds the exponent called `id` to that expression.

Of course in our syntax the character `/` is reserved, and can be accessed literally using the expression `//`. This is a very common technique, actually used for other metacharacters like $\backslash$.

6.1 Simplified syntax for the non expert user

A very common use of backreferences is to *group* a substring and later use its value in the same or another expression. For example, we may want to find if a string contains two occurrences of the same substring, or we may get a substring from an expression and use it in another one, that is mostly common is search/replace or data extraction functions. In both cases, we are usually interested in a generic substring that can be bound to a SRE variable with expressions like `/(.*)v/` and then referenced with `/v/`.

It seems quite unnatural to force the non expert user to this syntax for such tasks. We may reintroduce the concept of *asterisk wildcard*, known by all users as a part of the *filename globbing* utility of almost every UNIX command shell. In these programs, the star has not the semantics of RE, but stands for a generic string.

We say that, if a certain name is never bound to a SRE in an expression, then it can be bound to any string in A^*. Thus, `/v/` acts both as binding to the `(.*)` expression and as backreference, where the first use of a particular variable is its binding, and all the following are backreferences. This is what the user usually expects.

Example 6. In some releases of the standard C header `math.h`, there is a predefined macro called `random(n)` used to get a random number in the range $[0 \ldots n]$. This macro is expanded to the expression `rand() % n` using the standard function `rand()` that returns a number in the range

[0...MAXINT]. Since this macro is not standard, many compilers do not recognize it and generate an error.

The best way to solve the problem is using a synchronized regular expression over our source files in a search and replace function. We have simply to use these expressions:

Search: `random(/arg/)`

Replace with: `rand() % /arg/`

Example 7. Let us suppose that, in a UNIX filesystem, we have a set of files with filenames like:
`invoice_103_1994, receipt_183_1994, invoice_151_1995, receipt_1263_1995, ...`

The two numbers in each filename are the invoice/receipt serial number and its year, respectively. If we want to move all the invoices to the directory `invoices/` and all the receipts to the directory `receipts/`, renaming all of them in the format `year_number`, we may simply issue the command:

```
mv /type/_/number/_/year/ /type/s///year/_/number/
```

So far we have shown the utility of variables, but exponents are also a very interesting extension introduced in SRE.

Example 8. If we have two texts in files `F1` and `F2`, and suspect that one has been obtained from the other by simply "shuffling" its paragraphs and phrases and possibly deleting some of them (a very common technique!), then we may check this using a UNIX-like command as the following:

```
cat F1 Sep F2 |
   match (/*1/+/*2/+/*3/){n}
   'cat Sep'
   (/*1/+/*2/+/*3/){n}
```

The meaning of this command is the following:

1. concatenate the file `F1` with the file `Sep`, which contains only a separator text that does not appear in `F1` and `F2`, and then append `F2` to the result;
2. pass the resulting file to the `match` command, that is supposed to return true if its input matches the given SRE. Note that in the middle of the SRE we used a standard UNIX shell substitution command (`'cat Sep'`) that expands to the contents of the file `Sep`.

The SRE is forced by the presence of the separator text to match the contents of both files with the expression `(/*1/+/*2/+/*3/){n}` that means "the text is composed by three blocks, whose content is assigned to the asterisk-variables, (mixed and) repeated n times". If this expression matches both files with synchronization between all the variables, then we know that they are both composed by n parts with the same contents. This means that only the order is changed and possibly some parts have been substituted by others. Note that the number of asterisks that we use in the expression increases the granularity of the comparison.

6.2 Full syntax for the expert user

We show two examples where we use SRE syntax with its full power. The expressions under consideration are of course very complex but, as we stated in the introduction of this section, the general SRE syntax is reserved to expert users.

Example 9. Another possible use of our SRE is in the field of data security. One method to check if a data block (i.e., a document) has not been modified consists in the creation of a *document fingerprint* [3]. This fingerprint is obtained from the data by applying a hash function, and is stored separately from the document itself for later checking.

Another method to secure data consists in changing the data themselves, in a way that does not interfere with its contents, by adding a *signature* that can be later checked to ensure that the original data was not changed. This method is called *watermarking* [18] and is commonly used, e.g. to mark digital pictures.

We suggest a simple variant of this procedure. Our aim is to protect the data from malicious manipulations during transfer, that is we want to ensure that nobody changed parts of the data, or simply mixed its contents "on the fly" during the transfer (this is commonly called *data integrity preservation*). Our method is valid whenever the attacker has at his/her disposal only a time- and space- limited "view" on the data, as it is common when the attack is done on a data stream being transferred over a packet network.

A random key is generated, composed by m segments. For example, let $m = 5$. A key could be:

```
abc def ghi jkl mno
```

The key is then split in two parts, where the first part, that we may call *data key*, contains the first m' segments, and the second part is called *auxiliary key*. The data key is inserted in the source data k times: its segments are always in the given order, but may not be contiguous, that is there may be data between the key segments.

In our example $m' = 3$ and $k = 2$. After the key insertion, the data may look like the following:

```
xxyxy abc y def xyy ghi xy abc xyy def yy ghi x
```

Note that the key generation should be data-driven to ensure that the inserted segments do not overlap with the source data (for example, in a C source file the segments might be embedded in the code as comments).

To build the key file, we first write a random file and then insert both the data key and the auxiliary key in it, following the same method used for the data. The total number of key insertions must be exactly k.

In our example, the key file is the following:

```
23123 jkl 13 mno 145676 abc 4 def 37783 ghi 468
```

Then we send both data and key files. The numbers m, m' are fixed *a priori* as part of the communication protocol. Note that we do not explicitly send the data key, or the number of times it can be found among the data. This information will be retrieved directly by comparing the two files with a SRE.

Given the hypothesis that the key segments can overlap with neither the data nor the random file, and supposing that `A` is a set containing the alphabet of both the key and the data files, we can apply the following synchronized patterns to both files:

key file:	`((A* /(A*)1/ A* /(A*)2/ A* /(A*)3/) +`
	`(A* /(A*)4/ A* /(A*)5/)){n}`
data file:	`(A* /1/ A* /2/ A* /3/){n}`

If the match succeeds, then the data and the key files were not modified. One may also use the matching algorithm to locate and remove the key segments from the data in order to restore the source data.

Example 10. Web documents written in HTML usually contain links to other resources. These links have a displayed form, that usually identifies the link target, and an internal form, which contains other useful information and, most interesting of all, the link address. For example, a line of HTML like

```
Follow this
    <A HREF="http://www.website.com/file.html">
link
    </A>
```

would display in any browser as "Follow this link".

If we print the page, we loose all the link addresses. It may be very useful to automatically process the page and write these addresses near their description text. We may do this with SRE in a search/replace command:

Search:	`</(A[^>]*HREF="/([^"]*)1/"[^>]*)2/>`
	`/(([^<]|<[^A])*)3/<//A>`
Replace with:	`/2/ /3/ <//A> (/1/)`

Example 11. Many of today's documents contain internet addresses. Suppose we want to highlight these links to be more visible on a page produced, say, with LaTeX. We may use the SRE syntax in a search/replace command like this:

```
Search:         /((http|ftp)://///[^ \t\n]*)1/
Replace with:   \textbf{/1/}
```

7 Related work

We already considered several related notions in the body of the paper; here we limit ourselves to a few other significant related works. In the ECFG (Extended Context Free Grammars) [8] parameters have been added to nonterminal characters of a context free production rule in order to control the number of applications of the rule. So an instantiated value of the parameter acts in this case as a counter. For instance, $\{A(N) \longrightarrow aA(N-1),\ A(1) \longrightarrow a\}$, with $\{N \leftarrow 3\}$ gives rise to $A(3) \longrightarrow aA(2) \longrightarrow aaA(1) \longrightarrow aaa$. A class of such grammars with an infinite set of terminal characters represents a grammatical extension of logic programs, namely the DCG (Definite Clause Grammars), used in several Prolog implementations. In these programs strings are atoms or terms of a first order language and a production rule can handle a sequence of them.

Indeed, parameters occurrence within production rules dates back to WG (van Wijngaarden Grammars), designed to define the syntax of contextual programming languages and whose variants are widely used for compiler constructions. A WG rule is a rule schema of an infinite set of context free production rules. So the alphabet of nonterminal characters can be infinite, whilst the alphabet of terminal ones is finite. The values a parameter can assume are dealt with by a context free grammar. So WG grammars have been considered as two level grammars.

Instead of pointing out the exact relationships among the previous grammars, we notice the known fact that all of them are included in the contextual grammars called AG (Attribute Grammars), RAG (Relational Attribute Grammars), FAG (Functional Attribute Grammars) or CAG (Conditional Attribute Grammars). Such extensions of context sensitive grammars are also able to express the semantics of programming languages (the so-called Knuth semantics). However all these extensions go far beyond our goals. The same holds for the other extension of the DCG grammars, namely the well-known λHHG (Higher Harrop Grammars). They represent the grammatical view of λ-Prolog as the DCG of Prolog.

We end with some words about the actual implementations of backreferences in common tools and languages. As in our SRE, backreferences in text editing tools can be synchronized with an arbitrary preceding subexpression of their expression, which has been marked and grouped with parentheses. For example, the expression `(.*)\1` would match any string composed by two identical halves. Here `(.*)` stands for "any sequence of any character", and `\1` is a backreference to the value assigned to `(.*)`. In our SRE syntax, this would be expressed with the pattern $(A^*)\%vv$.

SRE variables act exactly as backreferences. Moreover, they "name" the subexpressions, so references are given in a clearer way than the implicitly-indexed one. The rule "backreferences can only be done after binding" is forced by the syntax in the indexed backreference method, and is given as a semantic rule in our definitions. Our SRE also allow a different kind of synchronization, i.e. exponentiation, where the content of two subexpressions may change but their repetitions must be the same.

A well known implementation of backreferences is in the GNU `regex` library [11]. Thanks to this library, many GNU tools like `egrep` [15] support backreferences in RE. The GNU matching engine demonstrates our claims about complexity: when backreferences are present in the regular expression, it switches from a very fast DFA algorithm to a NFA [9]. Actually the implementation is *very similar* to a (mathematically defined) NFA, but diverges from it in some points.

The price to pay for this extended recognition power is that the NFA implementation is an exponential algorithm with very extensive use of recursion. In other words, the algorithm is slower and it may potentially need much more memory to run.

Some attempts have been done to realize a faster, still powerful extended DFA model. To our knowledge, the best done so far is to slightly extend the boundary between the DFA and NFA domain, that is the regex engine can match more patterns without switching to NFA.

Another implementation of backreferences is in the PERL language [20]. We know many books that discourage the use of backreferences in PERL because this could make the matching very complex and time consuming [9]. PERL also allows one to use an indefinite number of backreferences (while the GNU code limits this number to nine), and this appears to be unsafe, since the unexperienced user may feel free to use them too many times, thus writing very inefficient programs.

Acknowledgements. We wish to thank Stefano Varricchio for his helpful suggestions on the topics of the paper. Our gratitude goes to the referees for suggesting many important corrections to the paper. Finally, we are grateful to Monica Nesi and Igor Melatti for their help in reviewing the preliminary version of this paper.

References

1. A.V. Aho, *Algorithms for Finding Patterns in Strings*. In: J. van Leeuwen, (ed.) *Handbook of Theoretical Computer Science* **1**, 257–300 Amsterdam: Elsevier 1990
2. D. Angluin, *Finding Patterns Common to a Set of Strings*. *Journal of Computer and System Sciences* **21**, 46–72 (1980)
3. D. Boneh, J. Shaw, *Collision-secure Fingerprinting for Digital Data*. In: D. Coppersmith, (ed.) *Proceedings CRYPTO 95* LNCS **963**, 452–465, Berlin Heidelberg New York: Springer 1995
4. M. Crochemore, W. Rytter *Text Algorithms* Oxford University Press 1994
5. A. De Luca, S. Varricchio, *Finiteness and regularity in semigroups and formal languages*. Berlin Heidelberg New York: Springer 1999
6. S. Dumitrescu, G. Păun, A. Salomaa, *Languages associated to finite and infinite sets of patterns*. *Revue Roumaine de Mathématiques Pures et Appliquées* **41**(9–10), 613–631 (1996)
7. J. Dassow, G. Păun, A. Salomaa, *Grammars with Controlled Derivations* in [19]
8. P. Deransart, J. Maluszyński, *A Grammatical View of Logic Programming*. In: *Journal of Symbolic Computation* 1993
9. J. E. F. Friedl, *Mastering Regular Expressions*. Cambridge: O'Reilly, 1997
10. M.R. Garey, D.S. Johnson, *Computers and intractability: a guide to the theory of NP-completeness*. New York: Freeman 1979
11. *The GNU Project*: http://www.gnu.org/
12. J.E. Hopcroft, J.D. Ullman, *Introduction to automata theory, languages, and computation*. Reading, MA: Addison-Wesley 1979
13. J. van Leeuwen, editor *Handbook of Theoretical Computer Science* Vol **1**, Amsterdam: Elsevier 1990
14. M. Lothaire, *Combinatorics on Words*. In: Gian-Carlo Rota, (ed.) *Encyclopedia of Mathematics and its Applications* Vol **17**, pp. 6–8. Reading, MA: Addison-Wesley 1983
15. A. Magloire, *Grep: Searching for a Pattern* (iUniverse.com, 2000)
16. A. Mateescu, A. Salomaa, *Finite Degrees of Ambiguity in Pattern Languages*, *RAIRO. Th. Inform. and Appl.* **28**, 233–253 (1994)
17. V. Mitrana, *Patterns and Languages. An Overview*. *Grammars* **2**(2), 149–173 (1999)
18. F. A. P. Petitcolas, R. J. Anderson, M. G. Kuhn, *Information Hiding, a Survey*. *Proceedings of the IEEE, Special Issue on Protection of Multimedia Content* (IEEE, 1999)
19. G. Rozenberg, A. Salomaa (eds), *Handbook of Formal Languages*. Berlin Heidelberg New York: Springer 1997
20. L. Wall, T. Christiansen, J. Orwant *Programming Perl, 3rd Edition.* Cambridge: O'Reilly, 2000