View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Elsevier - Publisher Connector

Synchronized Regular Expressions ⋆

Giuseppe Della Penna [1,2] Benedetto Intrigila [1,3]
Enrico Tronci [1,4]

Area Informatica, Università di L'Aquila,
Coppito 67100, L'Aquila, Italy

Marisa Venturini Zilli [1,5]

Dip. di Scienze dell'Informazione, Università di Roma "La Sapienza",
Via Salaria 113, 00198 Roma, Italy

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 stresses the need of more powerful tools to interact with texts. Indeed, much work has been done to provide non–programming tools that can be useful for the most common text manipulation issues. Regular Expressions (RE), introduced by Kleene, are well–known in the formal language theory. RE received several extensions, depending on the application of interest. In almost all the implementations of RE search algorithms (e.g. the `egrep` [14] UNIX command, or the Perl [17] language pattern matching constructs) we find *backreferences* (as defined in [1]), i.e. expressions that make reference to the string matched by a previous subexpression. Generally speaking, it seems that all the kinds of synchronizations between subexpressions in a RE can be very useful when interacting with texts. Therefore, we introduce the Synchronized Regular Expressions (SRE) as a derivation 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 talking about formalisms that should have a practical utility and can be used in the real world, 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.

⋆ An extended version of this paper is considered for publication in Acta Informatica

[1] This research has been partially supported by MURST project TOSCA

[2] Email: gdellape@univaq.it
[3] Email: intrigila@univaq.it
[4] Email: tronci@univaq.it
[5] Email: zilli@dsi.uniroma1.it

©2002 Published by Elsevier Science B. V. Open access under CC BY-NC-ND license.

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 are usually converted to electronic format to save space) stresses the need of more powerful tools to interact with texts. This is true at every level – from the beginner to the advanced user – although very expert users may have tasks that can be accomplished only writing ad-hoc programs.

Indeed, much work has been done to provide non–programming tools that can be useful for the most common text manipulation issues.

Regular Expressions (RE from now on), introduced by Kleene, are well–known in the formal language theory. RE today 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 received several extensions, depending on the application of interest. As an example, wildcards like '?' or '*' are used as abbreviations of more complex RE in all the operating systems command shells.

Patterns introduced by Angluin [2] and also studied by other authors (see [16] for an overview), 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` [14], `sed` and `awk` UNIX commands, or the Perl [17] language pattern matching constructs) we find *backreferences* (as defined in [1], see also [8]) 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 the kinds of synchronizations between subexpressions in a RE (like backreferences) can be very useful when interacting with texts. Therefore, we introduce the Synchronized Regular Expressions (SRE) as a derivation of the well–known 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*. We focus on these kinds of synchronizations since they share very good properties:

- they seem 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 Section 6);
- they have an acceptable computational complexity, when used under reasonable constraints (see Section 5).

In this paper we give a formal syntax and 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 talking about formalisms that should have a practical utility and can be used in the real world, 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 low–level, non programmer 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 difficult.

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 to the "advanced users", presents the extensions in a fully functional fashion available through the well–known RE syntax, without the introduction of other complex constructs or the need of programming. Nevertheless, the full syntax can be also 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 manipulations 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 in a very simplified and intuitive way.

The paper is organized as follows.

In Section 2 we define our Synchronized Regular Expressions and show how to assign them a language.

In Section 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 [6]

In Section 4 we address the membership problem in our context. We show that the membership problem turns out to be NP-complete, even when the number of occurrences of each synchronized element is bound to two.

In Section 5 we consider a natural restriction of SRE, namely limiting the number of synchronized elements. This time the problem turns out to be polynomial.

In Section 6 we address the problem of defining an user–friendly syntax for SRE, suitable to different kinds of users.

The related work Section 7 contains a comparison between our approach and others, both in the theory of formal languages and in implementations of regular expressions.

2 Synchronized Regular Expressions

2.1 Syntax of Synchronized Regular Expressions

In this section we define our extension of the classical RE. We give the standard definition of RE, enriched with the syntax for backreferences from [1], and the new syntax for synchronized exponents.

Definition 2.1 The *Synchronized Regular Expressions* on an alphabet A, a set of *variables* V and a set of *exponents* X are 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:

(i) $e_1^* \in SRE$ (star)
(ii) $\forall x \in X \quad e_1^x \in SRE$ (exponentiation)
(iii) $\forall v \in V \quad (e_1)\,\%v \in SRE$ (variable binding)
(iv) $e_1 e_2 \in SRE$ (concatenation)
(v) $e_1 + e_2 \in SRE$ (union)

We will address variable occurrences that are not arguments of a binding operation as *backreferences*.

The backreferences syntax from [1] seems to be underspecified. The definition allows different interpretations (compare, for example, [1] with the more end-user oriented [8]) since the intended meaning of some expressions implies several restrictions that are not expressed in the syntax, and these may cause problems to the user. In fact, the above definition allows:

- Multiple bindings on the same variable. For example, consider the SRE $(a^*)\,\%v\,b\,v\;(b^*)\,\%v\,c\,v$. If, as natural, we suppose that one binding replaces the previous, the language generation would rely on a not specified "expression ordering".
- Loops on variable bindings. These may cause *deadlocks*, as in the expression $(v_1\,a)\,\%v_2\,d\;(v_2\,b)\,\%v_1$. 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.
- 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 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. As an example, consider the SRE $(a^*)\%v + v$. 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 SREs:

Definition 2.2 A Synchronized Regular Expression is *valid* if, when the expression is analyzed in left–to–right order, the following holds:

(i) Bindings are always done on *fresh* (i.e. not used in the preceding subexpression) variables.

(ii) 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 Section 2.2.

Proposition 2.3 *A valid SRE does not allow:*

(i) *multiple bindings on the same variable,*

(ii) *looped variable bindings,*

(iii) *recursive bindings,*

(iv) *late bindings,*

(v) *unbound variables.*

Note 1 *In the present paper we omit the proofs for all the propositions. These proofs can be found in the full-length version of the paper, which is unavailable at present since it has been submitted to Acta Informatica.*

Remark 2.4 Since we shall consider *valid* SRE only, we stipulate that, from now on, "SRE" stands for "valid SRE".

2.2 Synchronized Regular Expressions Semantics

We present a semantics for SRE, corresponding to that originally presented in [1], using induction on the expression structure.

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 set to a number.

The function is true if, 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:

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

$$Eval\left(\{(e_1e_2,\alpha)\}\cup PS, V_B, V_F, E_B\right) = \\ \quad \bigvee_{\alpha_1\alpha_2=\alpha} Eval\left(\{(e_1,\alpha_1),(e_2,\alpha_2)\}\cup PS, V_B, V_F, E_B\right)$$

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

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 Section 2.1.

$$Eval\left(\{((e)\,\%v,\alpha)\}\cup PS, V_B, V_F, E_B\right) = \\ \quad Eval\left(\{(e,\alpha)\}\cup PS, V'_B, V'_F, E_B\right) \wedge (\forall\,(v,\alpha')\in V_F)\ (\alpha' = \alpha)$$

where

$$V'_B = (V_B \setminus \{(v,\alpha')\,|\alpha'\in A^*\})\cup\{(v,\alpha)\}$$
$$V'_F = V_F \setminus \{(v,\alpha')\,|\alpha'\in A^*\}$$

note that if $Eval$ is applied only to valid SREs (see Definition 2.2), the rule for V'_B can be simplified deleting the multiple-binding check:

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

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

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

note that this rule also applies to the star-expression e^*, which can be considered as an exponent-expression e^x where the exponent x is always fresh and not synchronized.

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

Given a string s and a SRE e, the pattern matching problem of e on s is evaluated by the expression $Eval\left(\{(s,e)\},\emptyset,\emptyset,\emptyset\right)$.

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 2.2. Indeed, without the restrictions, $Eval$ is always defined but nondeterministic (for the "multiple binding problem" explained in Section 2.1).

To evaluate more easily a SRE, the reader can actually assign all the exponents at the beginning of the evaluation. The following Definition 2.5 and Proposition 2.6 prove the correctness of this approach.

Definition 2.5 An *exponent assignment* is a function $\sigma_X : X \to \mathbb{N}$.

The action of an exponent assignment on a SRE is as follows:

$$(1) \quad \sigma_X(e) = \begin{cases} v & \text{if } e \equiv v \\ \sigma_X(e_1)\,\%v & \text{if } e \equiv (e_1)\,\%v \\ a & \text{if} e \equiv a \\ \sigma_X(e_1)^* & \text{if } e \equiv {e_1}^* \\ \sigma_X(e_1)\,\sigma_X(e_2) & \text{if } e \equiv e_1 e_2 \\ \sigma_X(e_1) + \sigma_X(e_2) & \text{if } e \equiv e_1 + e_2 \\ (e_1)^n = \underbrace{\sigma_X(e_1)\ldots\sigma_X(e_1)}_{n-\text{times}}, \text{where } n = \sigma_X(x) & \text{if } e \equiv {e_1}^x \end{cases}$$

We have the following

Proposition 2.6 *For every SRE e,* $\mathcal{L}(e) = \bigcup_{\sigma_X} \mathcal{L}(\sigma_X(e))$

3 Synchronized Regular Expressions in Formal Languages

Here we classify Synchronized Regular Expressions in the formal languages hierarchy.

3.1 Synchronized Regular Expressions are Context Sensitive

To show that SRE are Context Sensitive [11] and that the membership algorithm for SRE is in NP [9] (result that will be later used in Section 4) we proved the following

Proposition 3.1 *The language of a SRE can be accepted by a nondeterministic Turing Machine in linear space and polynomial time w.r.t the input size.*

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

Corollary 3.2 *There is a nondeterministic Turing Machine M' that, given a SRE e and a string α, accepts the string if $\alpha \in \mathcal{L}(e)$ or rejects it if $\alpha \notin \mathcal{L}(e)$, using polynomial time w.r.t $|\alpha| + |e|$ and linear space w.r.t $|\alpha| \cdot log|e|$.*

3.2 Synchronized Regular Expressions and Scattered Context Grammars

Scattered Context Grammars (SC grammars from now) belong to the family of *grammars with controlled derivations* [6]. 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 the alphabet of non-terminal symbols, the alphabet of terminal symbols and the start symbol, respectively), 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).

It is known that (see [6]):

- 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 deserve 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 3.3 *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 binds to a variable)

Proposition 3.4 *Every 1-SRE language that does not contain the null string ϵ is in SC.*

3.3 Synchronized Regular Expressions do not generate all Context Free Languages

We proved that the Synchronized Regular Expressions on an alphabet *with more than one letter* do not contain all the Context Free (CF) Languages on the same alphabet. To this aim, we make use of the language of palindromes, that is known to be context free.

Proposition 3.5 *No Synchronized Regular Expression can generate the language of palindromes on an alphabet A with $|A| > 1$.*

4 Complexity of Membership Algorithm on SRE

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

4.1 Membership on SRE is NP-Complete

We already proved (in Section 3.1) that the membership problem for SRE is in NP. Moreover, Regular Expressions with backreferences have already been proved to be NP-Complete [1], and this obviously extends also to our SRE.

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.

Proposition 4.1 *The 3-CNF problem can be reduced to the membership problem on SREs.*

In the proof of the above proposition we use Synchronized Regular Expressions with exponents synchronized several times, thus one may think that the complexity may be somehow lower if we use synchronization in its basic form: each exponent is repeated only twice, i.e. only one synchronization per exponent is allowed.

To answer this question, we also proved that exponents ranging in $\{0, 1\}$ (like those used in the proof of Proposition 4.1) can be synchronized without using the explicit synchronization of SRE more than two times on each exponent, so there is no complexity growth if the number of synchronizations in the expression increases.

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 of Proposition 4.1 is valid regardless of the number of synchronizations used. Therefore, we have the following:

Proposition 4.2 *The membership problem on SREs that*

- *do not contain backreferences,*
- *contain exponents synchronized at most two times,*

is NP–Complete.

In other words, the complexity is intrinsic in the problem of exponent synchronization. Making use of the same technique, we can also improve a result of [2] as follows:

Proposition 4.3 *The membership problem on SREs that*

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

is NP–Complete.

Observe that, by the third hypothesis, variables behave as those of [2].

5 Synchronized Regular Expressions with Limited Synchronization Elements

In Section 4 we proved that the general pattern matching problem with SREs 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 uses like those in Section 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 SRE membership complexity becomes polynomial. This has been already stated for backreferences only [1]. Here we expand the proof extending it to exponent synchronization.

Remark 5.1 When we state that the number of synchronized elements is fixed, we should take into account also 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

The polynomial algorithm for SRE pattern matching has some similarities with the dynamic programming technique, but is actually enumerative. Likewise dynamic programming algorithms, we have a notion of *state*. Our states are tuples like $(pattern, string, assignments)$ that show a pattern, the string it must match and the assignments done to variables and exponents so far. The initial state s_0 contains the pattern and string given to the algorithm by the user, and the assignments are empty. On each step, the algorithm looks at all the states in the current *state set* and performs a single *matching step* on each of these, generating the next state set. Note that a single matching step can often be performed in many ways (e.g. an exponent can be assigned to different values), producing more than one state. 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.

The complexity of this algorithm is simple to express: let n be the length of the target string and m be the length (in characters) of the pattern. If we fix the number of possible synchronizations (both with variables and exponents) to k, we have that the number of states generated by synchronization elements 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 degree equal to the limit of synchronizations we fixed.

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

Proposition 5.2 *The membership problem for SREs 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 fix a common syntax for SRE extensions to be used in implementations such as text editors, command line search utilities like `grep` [14], etc. In the rest of this section, we use the word *syntax* to denote the syntax used to write SRE on a computer terminal, that is obviously a little different from the formal syntax introduced in Section 2.

Our idea is to let the user access the power of SRE on 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 in SRE syntax).

Let us first introduce the full syntax for SRE. We inherit all the common syntax used for RE, adding 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 subexpression 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 \.

In this paper we limit to two examples of SRE applications. Many other examples, both using simplified syntax and full syntax, can be found on the full-length paper.

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 a substring repeated at two different positions, 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 low-level user to this syntax for very simple tasks like these. 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.

Therefore, we may use our synchronized variables as *synchronized asterisks*. 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 6.1 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 `rand()` function that returns a number in the range [0 . . . `MAXINT`]. Since this macro is not standard, many compilers don't 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 just to use this expression:

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

Example 6.2 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, then we may try to check this using a UNIX command like:

```
cat F1 Sep F2 |
   match (/p1/+/p2/+/p3/){n}
   'cat Sep'
   (/p1/+/p2/+/p3/){n}
```

we suppose to have a third file called `Sep`, which contains a separator text that does not appear in `F1` and `F2`. The meaning of this command is the following:

(i) concatenate the file `F1` with the file `Sep` and then append `F2` to the result;

(ii) pass the obtained 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 `(/p1/+/p2/+/p3/){n}` that means "the text is composed by (at most) three blocks, whose content is assigned to the variables `p1`,`p2` and `p3`, (mixed and) repeated n times". If this expression matches both files with synchronization between all the variables, then we know that the files 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. The number of backreference variables that we use in the expression increases the granularity of the comparison, so we may be able to discover more complex manipulations.

6.2 Full Syntax for the Expert User

We show an example where we use the SRE syntax with its full power. The expressions used 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 6.3 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:

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

Replace with:
```
/2/ /3/ <//A> (/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) [7] 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 represent a grammatical extension of logic programs, namely the DCG (Definite Clause Grammars), used in several Prolog implementations. In them 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 the 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 known as AG (Attribute Grammars), RAG (Relational Attribute Grammars), FAG (Functional Attribute Grammars) or CAG (Conditional Attribute Grammars). Such extensions of context sensitive grammars are able to express also 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 allow to 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 half. 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 $(A^*)\,\%vv$.

Synchronized Regular Expression variables act exactly as backreferences. Moreover, they "name" the subexpressions, so references are given in a way more clear 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, 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 [10]. Thanks to this many GNU tools like `egrep` [14] allow the user to use them. 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 [8]. 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. That is, the algorithm is slower and it may potentially need much more memory to run.

Some attempts have been done to extend the implementation of the DFA model and obtain a regex engine which is fast as a DFA and has the power of a NFA. 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. But as long has NFAs are completely excluded (*if they can be excluded*) from these matching engines, the complexity problem using backreferences and similar matacharacters will remain.

Another implementation of backreferences is in the PERL language [17]. We know many books that discourage the use of backreferences in PERL because this could make the matching very complex and time consuming [8].

PERL also allows to use an indefinite number of backreferences (while the GNU code limit this number to nine), and this appears to be unsafe, since the unexperienced user may feel free to use them too many times, writing very inefficient programs.

References

[1] A.V. Aho, *Algorithms for Finding Patterns in Strings*, in J. van Leeuwen, editor *Handbook of Theoretical Computer Science* vol **1,** pp. 257-300 (Elsevier Science Publishers B.V., 1990)

[2] D. Angluin, *Finding Patterns Common to a Set of Strings*, in *Journal of Computer abd System Sciences* vol **21,** (1980) pp. 46-72

[3] D. Boneh, J. Shaw, *Collision–secure Fingerprinting for Digital Data*, in D. Coppersmith, editor *Proceedings CRYPTO 95* LNCS vol **963,** pp. 452-465 (Springer-Verlag, 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* (Springer, 1999)

[6] J. Dassow, G.Păun, A. Salomaa, *Grammars with Controlled Derivations* in G. Rozenberg, A. Salomaa, editors *Handbook of Formal Languages* (Springer-Verlag 1997)

[7] P. Deransart, J. Maluszyński, *A Grammatical View of Logic Programming* in *Journal of Symbolic Computation* (1993)

[8] J. E. F. Friedl *Mastering Regular Expressions* (O'Reilly, 1997)

[9] M.R. Garey, D.S. Johnson, *Computers and intractability : a guide to the theory of NP-completeness* (Freeman, 1979)

[10] *The GNU Project*: http://www.gnu.org/

[11] J.E. Hopcroft, J.D. Ullman, *Introduction to automata theory, languages, and computation* (Addison-Wesley, 1979)

[12] J. van Leeuwen, editor *Handbook of Theoretical Computer Science* vol **1,** (Elsevier Science Publishers B.V., 1990)

[13] M. Lothaire, *Combinatorics on Words*, in Gian-Carlo Rota, editor *Encyclopedia of Mathematics and its Applications* vol **17,** pp. 6-8 (Addison-Wesley, 1983)

[14] A. Magloire, *Grep: Searching for a Pattern* (iUniverse.com, 2000)

[15] F. A. P. Petitcolas, R. J. Anderson, M. G. Kuhn, *Information Hiding, a Survey*, in *Proceedings of the IEEE, special issue on protection of multimedia content* (IEEE, 1999)

[16] G. Rozenberg, A. Salomaa, editors *Handbook of Formal Languages* (Springer-Verlag 1997)

[17] L. Wall, T. Christiansen, J. Orwant *Programming Perl, 3rd Edition* (O'Reilly, 2000)