Research Article Open Access

Ján Kollár, Michal Sičák*, and Milan Spišiak

Abstraction of Meaningful Symbolized Objects

DOI 10.1515/comp-2016-0009
Received April 28, 2016; accepted June 8, 2016

Abstract: We describe the notion of abstraction and conceptualization of information obtained by symbolization of letters. We are able to recognize already observed information with those abstracted concepts. Further more, we are also able to recognize similar meaningful objects. Similarities identification is based on a repetition in the structure of information symbolized as a regular language string. We discuss approaches of repetition identification, i.e., longest repeating non-overlapping subsequence and longest common subsequence. Then we apply those approaches to the symbolized letters, thus obtaining abstracted information in the form of concepts.

Keywords: symbolization, conceptualization, automated abstraction, meaning inference

1 Introduction

Grammars are widely used formalisms. They allow us to grasp the structure of a language and help us with its comprehension. Many existing software systems do contain grammars, sometimes hidden and unformalized. Uncovering the underlying grammar can be useful feature in further development of such systems [1, 2]. Grammar can be utilized on many ways. For example, they may be used in language evolution [3] or in rapid domain specific language (DSL) development [4]. We can obtain grammars even from complex objects like graphical interfaces [5] and then use them for an automated development of DSLs.

The grammar inference is a grammar rule construction process that is achieved purely with the use of input strings. The strings belong either to positive (belonging to that language) or negative (not belonging to the language) sample. Gold in [6] has researched a process called regular inference in the limit. He argues, that in order to infer a grammar, one needs both positive and negative samples. The latter is needed to limit the regress of a regular grammar. Without those samples, an inference process might end up with the most general grammar, which can generate any sequence.

Meaningful information, such as written letters or spoken words can be symbolized i.e. transformed into a set of symbols that are processable by a computer. A grammar can be inferred from such a set and then used for the information recognition or reconstruction process. We describe a way to abstract strings obtained from a meaningful source by a process called symbolization. We also describe how to build a concept foundation that is based on obtained information.

Our approach leads to similar results as the regular inference. However, we do not want to find the perfect regular language for all input samples as in the case of the inference. There are no negative samples in our approach, since everything that we symbolize can be viewed as a positive sample. So the absence of negative samples could lead to inferring the most general regular language, as Gold [6] pointed out. Our approach is based on repetition pattern search within one or many samples. Therefore, we do not call that a language inference but rather a language abstraction.

Main contributions of this paper are:

- We present a problem of regular language abstraction in the section 4. This results into more compact grammar definition, as it is by means of a grammar inference. It also results into an identification of repeating parts.
- We present an algorithm to obtain longest repeating non-overlapping subsequence (LRNS) in the section 3 from an input sequence of symbols. Such a subsequence is an asset to repetition patterns identification and helps in the process of abstraction.
- Another subsequence problem, identification of a longest common subsequence (LCS) in multiple sequences is used for further improving abstraction across input data and inside identified repeated patterns. The process is described in the section 4.

Ján Kollár: Techical University of Košice, Department of Computers and Informatics, E-mail: jan.kollar@tuke.sk
***Corresponding Author: Michal Sičák:** Techical University of Košice, Department of Computers and Informatics, E-mail: michal.sicak@tuke.sk
Milan Spišiak: Techical University of Košice, Department of Computers and Informatics, E-mail: milan.spisiak@tuke.sk

– Those two approaches are researched and their application on an input data set are evaluated by an experiment reported in the section 7.

2 Information abstraction

Consider a grammar that encapsulates information as a regular language. Our approach would "consume" those grammars and perform abstraction on them. A greater grammar, still regular, is created. It recognizes all available pieces of information and another samples similar to the original one and their combinations. Since we are using regular grammars, a finite state automaton may be constructed for this recognition task, or we may use meta-execution tree forms of regular expressions [7].

Symbolization is a step that is necessary when we want to process information automatically by a computer. We humans perceive information based on their informal meaning, where computers process information purely as formal symbols [7]. Formal meaning can be described by a grammar. Data produced by our symbolization is in a form of regular language with defined symbol alphabet. We use test samples created by symbolization of Latin alphabet letters. The process itself is presented in the section 6. The results of symbolization are strings of symbols. For explanatory purpose, we use arbitrary symbols. The terminal set of the regular grammar we use is: $\Sigma = a, b, c, d, e, \dots, z$. We restrain ourselves only to regular expressions, since symbolized strings obtained by the letter recognition are in a simple form. We use augmented definition of regular expressions, which use these four operations:

- **Concatenation** operation means direct continuation of symbols, i.e. $r_1 r_2$ accepts two regular expressions that form a sequence. The concatenation operator is invisible.
- **Alternative** operation represents selection of only one listed regular expression. It is represented by a "|" sign. Therefore, in $r_1|r_2|r_3$, only one expression can be selected and others are discarded in current computation.
- **Closure** operation, also known as Kleene star operation. It represents zero to n repetitions of a single regular expression. It is a unary operation, and its operator is represented as $(r)^{\star}$.
- **Option** is an operation that represents an optional selection of a regular expression. Either the expression is or isn't selected. It can be rewritten as $r|\varepsilon$, where ε is an empty expression. It is designated as $(r)^{+}$. We try to avoid empty expressions, therefore this operation is useful for us. We note that this is not a standard operation for minimal definition of regular expressions.

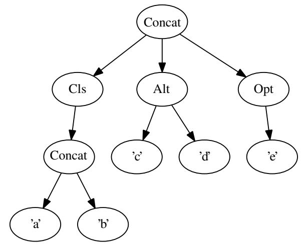

Figure 1: Tree form of $(ab)^{\star}(c|d)(e)^{+}$ regular expression

Regular expression in a textual form, like $(ab)^{\star}(c|d)(e)^{+}$, can be transformed into a tree form. We have expressed this notion in our previous work [7]. Example of this regular expression as a tree is in Fig. 1. Alternative and concatenation operators create n-ary nodes, where closure and option are unary. Leaf nodes represent symbols themselves.

Let's have a symbol string (1).

$$abcabcdedef \tag{1}$$

It represents a kind of information whose meaning is now irrelevant. With the abstraction, we should obtain more general form. We see a repetition of symbols in our example. By generalizing this fact, we get regular expression (2).

$$(abc)^{\star}(de)^{\star}f \tag{2}$$

It is more general representation of (1) since it can represent strings like: $abcf, def, abcdef, abcdededef, \dots$

Therefore the basic idea behind symbol string abstraction is a search for repetitive patterns and then putting them into the closures. This may be done multiple times over one input string. Take example (3) for instance. What we can see are two levels of abstracted regular expression, where the rightmost is the most abstract.

$$ababcdababcdgh \rightarrow (ababcd)^{\star}gh \rightarrow ((ab)^{\star}cd)^{\star}gh \tag{3}$$

So far we have discussed abstraction performed on the LRNS. But what if we have more than one input sequence, or there are sequences in alternatives? We can then determine the LCS of each string and proceed accordingly.

An abstraction of multiple input sequences can be performed. Consider sequences "abc" and "dbe". They contain the same symbol, "b". An automaton that accepts the expression $abc|dbe$ is depicted in Fig. 2. We can abstract both sequences into one regular expression "$(a|d)b(c|e)$".

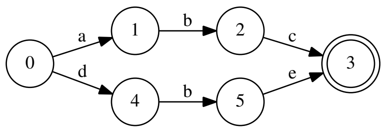

Figure 2: Finite state automaton for $abc|dbe$ expression

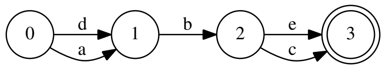

Figure 3: Finite state automaton for $(a|d)b(c|e)$ expression

Automaton constructed from that expression is depicted in Fig. 3. We can see that the count of states dropped from 6 to 4. The amount of transitions is lowered as well. We can claim that the automaton in Fig. 3 is more abstract than the automaton in Fig. 2, since the former can accept sequence of symbols like "abe" where the latter cannot.

3 Longest repeating non-overlapping subsequence

In order to perform an abstraction on one string, we need to extract repeating patterns from it. This presents a problem, commonly known as a longest repeating subsequence. What is more special in our case is the fact that the subsequence must be non-overlapping. Considering an example sequence:

$$ababa \tag{4}$$

the longest subsequence here is "aba". However, it's overlapping with itself. Therefore conducting an abstraction over it would yield erroneous results. We need to find the longest sequence that does not overlap. In this case, we have two such sequences: ab and ba. For now, it's unimportant which sequence is selected for an abstraction. It's notable that those sequences do overlap with themselves, so both cannot be chosen at the same time. The longest subsequence problem can be solved with suffix arrays.

Suffix array is an array consisting of all suffixes of any string in sorted order. Construction of such an array is rather simple process and many efficient algorithms exist [8]. Basically the input sequence is split into all of its suffixes and then those suffixes will be sorted. The result is however in a form of an index array, which accompanies the original array. We claim it is a suffix array, where in fact, there are two arrays present. The index array points to the starting index of each suffix. In the case of our sequence (4), the suffix array would look like:

$$[a, b, a, b, a][4, 2, 0, 3, 1] \tag{5}$$

As we still call it a suffix array, we can view those two arrays as a single two dimensional array:

$$\begin{gathered}[a]\\ [aba]\\ [ababa]\\ [ba]\\ [baba]\end{gathered} \tag{6}$$

We propose an algorithm to find the LRNS with use of a suffix array. Since suffixes in an array are sorted, we need to compare the neighbouring pairs inside the array. While suffix symbols are equal, this comparison runs until the end of one suffix is reached or if we run into an overlapping part.

The overlapping part is indicated when the comparison of suffix with lower starting index reaches the starting index of a second suffix. In that case the comparison ends. But unlike the case of the end of a suffix or in the case of a character mismatch, in which case we just start comparing the next neighboring pair, when overlapping occurs the next suffix has to be compared with the first one. Taking following sequence as an example:

$$\begin{aligned} abababa = &[a], [aba], [ababa], [abababa],\\ &[ba], [baba], [bababa]\end{aligned} \tag{7}$$

the suffixes "aba" and "$ababa$" do overlap at the index 2, hence their part "ab" is LRNS. We need to compare "aba" with "$abababa$" since from those two suffixes a new, longer LRNS has been found: "aba" that does not overlap.

The algorithm pseudo-code for the LRNS search is depicted on pages 111 and 112. We need to have two original arrays in the memory, marked as *vector* for the sequence itself and *array* for an array of indices. The basic variables are i_1 and i_2, the indices pointing to the symbols inside our input sequence. We pick those indices from an index array provided by a suffix array. We need to keep the information about indices from that array as well. For the purpose of our algorithm, we mark them as j_1 and j_2. The first one points to the first actual string processed, the latter is used purely when sequences overlap and we are comparing pair of strings, which are not direct neighbours. The global variable of sequence length is needed, we shall mark it as i_{max}. Variable o contains the difference of lengths between searched sequences. It's used to mark the overlapping part. The result of our function is stored in the values f_l and f_i, where the former is the length and the latter is the starting index of the first LRNS found. Algorithm can be easily rewritten to find all LRNS. We would need to store all longest f_l and f_i pairs, sort them and chose the longest ones. Variables c_i and c_l store current LRNS in the memory. They are compared with the final ones and if c_l is longer than f_l, then their values are assigned to corresponding f variables.

The main part of this algorithm compares symbols indexed by i_1 and i_2. At the start of an execution, those numbers are found out at indices 0 and 1 of a suffix index array. In the case of a match, the current length (c_i) is incremented. We then perform a test to find out whether $i_1 + 1 \le i_{max}$. In case the test fails, we move to the next pair of indices. If the test passes, o variable is decremented. If o is zero, the sequences are going to overlap. We need to compare suffixes with starting indices j_1 and $j_2 + 1$ (this means that we are now comparing two non-neighbouring suffixes). In the case they do not overlap, the next symbols in both sequences are compared, thus we only increment both i_1 and i_2. The algorithm terminates when all neighbouring pairs have been compared.

```
$i_{max} \leftarrow$ LENGTH($array$)
$i_1 \leftarrow indices[0]$
$i_2 \leftarrow indices[1]$
$o \leftarrow |i_1 - i_2|$
$j_1 \leftarrow 0$
$j_2 \leftarrow 1$
$c_i \leftarrow i_1$
RESETCURRENT
SETFINAL
$f_l \leftarrow c_l$
$loop \leftarrow true$
while $loop$ do
    if $array[i_1] = array[i_2]$ then
        $c_l \leftarrow c_l + 1$
        if $i_{max} - i_1 > 1$ then
            $o \leftarrow o - 1$
            if $o > 0$ then
                SHIFTINDICES
            else
                OVERLAPSHIFT
            end if
        else
            NEXTPAIR
        end if
    else
        NEXTPAIR
    end if
end while

function NEXTPAIR
    if CANCONTINUE($j_1 + 2$) then
        $i_1 \leftarrow indices[j_1 + 1]$
        $i_2 \leftarrow indices[j_1 + 2]$
        $o \leftarrow |i_1 - i_2|$
        if $c_l > f_l$ then
            SETFINAL
        end if
        RESETCURRENT
    end if
end function

function OVERLAPSHIFT
    if CANCONTINUE($j_2 + 1$) then
        $i_1 \leftarrow indices[j]$
        $i_2 \leftarrow indices[j_2 + 1]$
        $j_2 \leftarrow j_2 + 1$
        $o \leftarrow |i_1 - i_2|$
        if $c_l > f_l$ then
            SETFINAL
        end if
        RESETCURRENT
    end if
end function

function SHIFTINDICES
    $i_1 \leftarrow i_1 + 1$
    $i_2 \leftarrow i_2 + 1$
end function
```

4 Abstraction process

We have described the process of obtaining the LRNS of an input string of symbols. This part is then extracted from the original input, so we've got a substring that is repeated and a list of intermediate strings, "leftovers" from the separated string. Let us have an example:

$$abcdabceabcm = abc, [\varepsilon, d, e, m] \tag{8}$$

The repeated string abc is identified and then the list of rest pieces follows. The list length is $r + 1$, where r is the number of repeated substring occurrences.

For an input string s and obtained LRNS l we can separate any string as:

$$s \rightarrow a_1 l a_2 l \dots l a_n, \quad l = \text{LRNS}, 1 \le n \le r \tag{9}$$

Where all a_i's are the leftover substrings. Any of them can be empty, this is denoted by the ε symbol.

In an abstraction process, our identified LRNS is picked from the original string and put into a closure. The rest is then laid out according to its internal structure. From our example string (8), the regular expression $(abc(d|e|m))^*$ is constructed, as we see that "abc" always repeats itself. The rest in this example is straightforward, since each LRNS is continued by one different symbol. They are put in an alternative after our LRNS.

```
function setFinal
    $f_i \leftarrow c_i$
    $f_l \leftarrow c_l$
end function

function resetCurrent
    $c_i \leftarrow i_1$
    $c_l \leftarrow 0$
end function

function canContinue(x)
    $loop \leftarrow x < i_{max}$
    return $loop$
end function
```

The leftovers of LRNS might not be spread evenly. Considering separated string from (9), basic abstraction based on LRNS can be written as:

$$a_1 l a_2 l \dots l a_n \rightarrow \begin{cases} a_1(l(a_2|\dots|a_n))^* & \text{if } a_1 = \varepsilon \\ ((a_1|\dots|a_n)l)^* a_2 & \text{if } a_2 = \varepsilon \end{cases} \quad (10)$$

If both $a_1 \neq \varepsilon$ and $a_n \neq \varepsilon$, then selection of rule is determined by further abstraction of leftover part. Generally, the better abstraction of two possible options is selected. So we need to find out whether to put elements before or after LRNS in an alternative. In the case of the list $[a, b, c]$ and LRNS l, two possibilities arise: $a(l(b|c))^*$ or $((a|b)l)^* c$. Here in this selection we must use heuristic approach. Otherwise we would possibly get exponentially hard problem to solve. Therefore we always use the first option. The abstracted element always generates the original input in the both cases anyway, so there is no loss, only a danger of choosing a slightly worse option.

A conversion needs to be done in case that an element of an alternative is an empty string. Considering $(abc(d|e|m))^*$ as an example, and $e = \varepsilon, b \neq \varepsilon, m \neq \varepsilon$, we apply conversion rule (11), and get a result $(abc(d|m))^*$. Simply put, if any member of an alternative is an empty string, we need to put that alternative inside an option and remove every empty string from it.

$$\begin{aligned} &(s_{alt}) = (a_1|a_2|\dots|a_i|\dots|a_n) \rightarrow (s_{alt'})^+ \\ &a_i = \varepsilon, a_i \notin s_{alt'}, n \in \mathbb{N}, 1 \le i \le n \end{aligned} \quad (11)$$

So far, we have identified the LRNS and leftover parts. There is another option possible in order to perform an abstraction: identification if the LCS is either found in all input sequences or only in the portion of them. Those steps cannot be applied at once, but application in sequence may lead to the same results. Finding the LCS is an important step in this part of abstraction.

In contrast of finding the LRNS, where we are searching for the repetitions inside one string, the LCS is identified across two or more strings. The LCS identification is a well understood problem [9, 10], so we won't describe it here.

In case we have a set of sequences, we can abstract them by using transformation:

$$\begin{aligned} (a_1|a_2|\dots|a_n) &\rightarrow (b_1cd_1|b_2cd_2|\dots|b_ncd_n) \\ &\rightarrow (b_1|b_2|\dots|b_n)c(d_1|d_2|\dots|d_n) \end{aligned} \quad (12)$$

Where $n \in \mathbb{N}$ and $a_i, b_i, d_i, 1 \le i \le n$ are substrings, $c \neq \varepsilon$, $a_i \neq \varepsilon$, b_i and d_i can be empty. String c is ubiquitous LCS of input strings abstracted with alternative operator.

In the case that there is no common substring for all members, we can proceed to finding the LCS of at least two strings from alternative. This is described by transformation:

$$(a_1|a_2|\dots|a_n|b_1|b_2|\dots|b_j) \rightarrow (s_a|b_1|b_2|\dots|b_j) \quad (13)$$

Where a_i, $1 \le i \le n$ are strings with common sequence and b_i, $1 \le j \le n$ are not. The symbol s_a represents transformation (12) application on all a_i. At least two a_is are necessary and the set of b_i may not be empty, otherwise it would be just the transformation (12).

5 Conceptualization process

The grammar that is being created can be constructed using previously described processes in more than one way. As we mentioned in section 2, abstraction may be performed on an input set of sequences before each member is processed. This is done by using abstraction (13) recursively. On each created abstraction s_a, we can apply process (12), and if applicable, we identified shorter parts that can be brought back on the same level as the LCS identified earlier. This example illustrates the idea:

$$\begin{aligned} [abcdef, abcgef] &\xrightarrow{(13)} abc(def|gef) \\ &\xrightarrow{(12)} abc((d|g)ef) \rightarrow abc(d|g)ef \end{aligned} \quad (14)$$

The conceptualization is achieved by performed abstraction. The resulting grammar is the super-concept that encapsulates the whole input set.

6 Symbolization of Letters

We designed experiments to show the results of combining approaches mentioned in the previous sections. The abstraction is performed on meaningful data that were obtained by a symbolization of Latin alphabet letters. This

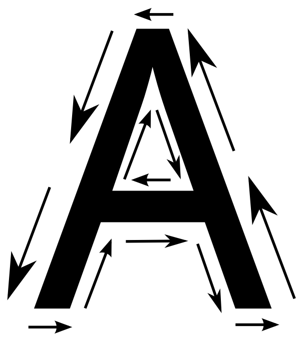
Figure 4: Chain code for the letter A

process deserves a bit of explanation, so before we show any experimental results, we present short but concise description of image object symbolization.

The basis of our abstraction is a chain code [11]. We use our own algorithm, which has been described in [12]. As an example, on Fig. 4 we see the arrows describing the shape of letter A. This process works on the level of pixels, where it searches for shape directions. It obtains direction vectors, which are subsequently transformed into a set of eight distinct symbols. Those symbols are numbers from 1 to 8, each representing different direction. The number 1 stands for the right (or east) direction, 2 for bottom-right (south-east) and so on.

At the end of such symbolization we get symbol strings. Here is an example of the symbolized form of the letter *E*:

$$[1, 2, 3, 4, 5, 4, 3, 2, 1, 2, 3, 4, 5, 3, 2, 1, 2, 3, 5, 7]$$

The repetition patterns are shown by repetition of numerals $1, 2, 3$ that roughly represents the three vertical lines of the *E* letter. We have used reduces sets, where each sequence repetition of the same number has been replaced by exactly one number only. We can process images with different sizes and obtain same results with that. And the repetitive patterns are easier to find.

7 Experimental Results

Using the symbolization explained in the previous section, we obtain a set of symbolic strings. They now can be abstracted with the use of abstraction process described in the section 4. The purpose of these experiments is to clarify, whether our abstraction process captures any meaningful patterns, i.e., concepts, and is able to abstract them. And the other important point to make is that in what order we should perform our two distinct abstractions, namely LCS and LRNS. A way to find out if our process captures any concepts, is to set a control group of randomly generated strings. They represent a noise, meaningless data. Comparing random data results with our abstractions will show us, if our process is any good in capturing meaningful concepts. The actual data we have processed come from two distinct sets of letters. Latin alphabet has 25 basic uppercase letters, hence with the use of two distinct fonts, we obtain 50 symbol strings in total. The length varies from 6 to 25 symbols for each string. The random data have been also selected from this interval. Hence 50 randomly generated strings of symbols with the length from the interval 6-25 have been used.

We use three distinct approaches in our experiments. The first approach just evaluates data as they are given, no abstraction process has been performed upon them. We designate this in result charts as **No Process**. The second approach uses LCS as the starting point. Then, LRNS is applied on the parts of its result. We designate this process as **LCS** in charts. The third is reverse of the second approach, LRNS is used first, designated as **LRNS** in charts.

The first group of experimental results is depicted in Fig. 5, where the x axis of line chart denotes the current count of the input regular strings, while the y axis denotes the count of the symbols within the regular expression after the process application. We selected this property, as it shows the actual size of regular expressions. So in total, there are six different result sets, three for symbolized strings and three for random data. On the Fig. 5 the best results were obtained with the use of LCS first strategy. It deviates from its random counterpart rather significantly and even more from any other result. Note the strong linear growth on all three random data results. Interesting thing to point out is also the poor results achieved with LRNS first strategy. This is however easy to explain, since LRNS breaks each string into regular expressions separately, and then it is hard for LCS to capture similarities. The reverse order, the LCS first strategy, shows slightly non-linear grow, which might indicate that as the input set grows larger, the curve might resemble logarithmic grow.

In the second experiment we measured the state amount of finite state automata obtained from abstracted regular expressions. We used standard algorithm for conversion, with the use of miniaturization. Only three sets of data were selected for evaluation this time. The LCS first and LRNS first performed over actual symbolized data, and LCS first performed over randomly generated data.

The results are depicted in Fig. 6. Here we see the confirmation of the superiority of LCS first strategy. The LCS first random data strongly correlates with LRNS first strategy, which renders LRNS first strategy no better than a chance. On the other hand however, LCS first strategy per-

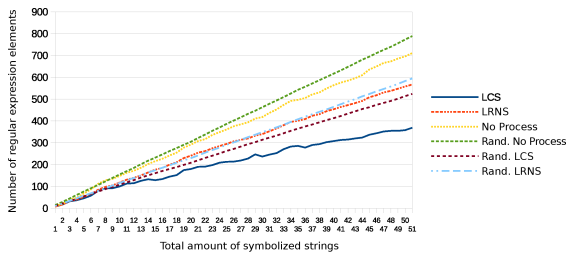

Figure 5: Comparison of symbol count in regular expressions

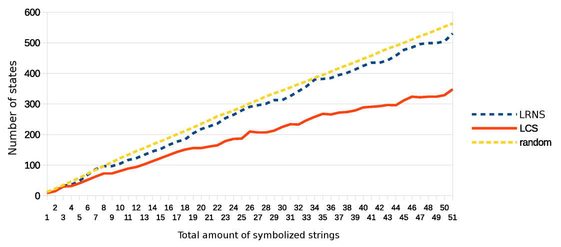

Figure 6: Comparison of state count in finite state automata

formed over symbolized strings presents better abstraction as it was in the previous experiment. We may conclude that this strategy achieves abstraction to a certain extent.

8 Discussion

We can see from the results in Fig. 5 and Fig. 6 that LCS first strategy does indeed abstract meaningful data. This abstraction is rather simple and surely can be done with more precision. For example, after the application of LRNS we could apply LCS once more, or apply it in different way. But, this is important to stress out, such a process would be either highly undeterministic, or highly computationally expensive. We could fall into an exponential complexity. This is the reason why we have chosen such a heuristic method. It simplifies the process while retains abstraction ability.

We can also see, that randomly generated data show strong linear characteristic. This strongly indicates that our abstraction process is capable of abstracting meaningful elements out of meaningful strings, while the noise remains meaningless even after abstraction. This opens the possibility of a meaning detection with the use of our LCS first approach.

9 Related works

Oncina and Garcia in [13] created an algorithm, called RPNI, for regular language inference in the limit. Their approach finds best regular grammar in theoretical search space of all regular grammars. Dupont, Miclet and Vidal improved on that work in [14]. Dupont in [15] extended RPNI algorithm so that it can infer grammars incrementally with ongoing stream of positive and negative samples. The inferred grammar was independent of samples order. Evolution algorithms may be used in inference process in order to obtain optimal results, even when inferring context-free grammars (CFG) [16].

Fernau [17] explored algorithms for regular ingerence basing only on a positive sample set. This approach bears greater similarity to ours because of that fact. The Prague Stringology Club members have been researching the aspects of approximate regular string matching for almost 20 years and their results relate to our work as well [18–20]. Their work relates heavily on automata however, where we try to use meta-executable trees instead.

10 Conclusion

We presented a method of symbolized information conceptualization basing on abstraction. We identified two approaches, both combinable, to perform abstraction of regular languages. LCS first strategy has shown significantly better results than reverse procedure, when LRNS were identified first.

In our results, we used fixed order of process applications. This probably lead to not optimal results. In our future research, we plan to concentrate on exploiting evolutionary algorithms in order to enhance abstraction. Genetic programming [21, 22] shows us a possible direction of our next steps. We may construct an evolutionary machine that performs abstraction with use of randomly mutated sets of processes and specific application rules, then selects the best performing ones to evolve another generation. This may lead to obtaining more precise abstractions, which are expected to precede performance of our current fixed best performing process, i.e., the LCS first strategy.

References

[1] Mernik M., Crepinsek M., Kosar T., Rebernak D., Zumer V., Grammar-based systems: Definition and examples. *Informatica*

(Slovenia), 28(3):245–255, 2004.

[2] Klint P., Lämmel R., Verhoef C., Toward an engineering discipline for grammarware. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 14(3):331–380, 2005.

[3] Kollár J., Pietriková E., Genetic evolution of programs. *Central European Journal of Computer Science*, 4(3):160–170, 2014.

[4] Chodarev S., Development of domain-specific languages based on generic syntax and functional composition. *Information Sciences and Technologies Bulletin of ACM Slovakia*, 4(3):47–53, 2012.

[5] Bačíková M., Porubän J., Lakatoš D., Defining domain language of graphical user interfaces. In *SLATE*, pages 187–202, 2013.

[6] Gold E. M., Language identification in the limit. *Information and control*, 10(5):447–474, 1967.

[7] Kollár J., Formal processing of informal meaning by abstract interpretation. *Smart Digital Futures 2014*, 262:122, 2014.

[8] Kärkkäinen J., Sanders P. Simple linear work suffix array construction. In *Automata, Languages and Programming*, pages 943–955. Springer, 2003.

[9] Hirschberg D. S., Algorithms for the longest common subsequence problem. *Journal of the ACM (JACM)*, 24(4):664–675, 1977.

[10] Bergroth L., Hakonen H., Raita T., A survey of longest common subsequence algorithms. In *String Processing and Information Retrieval, 2000. SPIRE 2000. Proceedings. Seventh International Symposium on*, pages 39–48. IEEE, 2000.

[11] Liu Y. K., Žalik B., An efficient chain code with huffman coding. *Pattern Recognition*, 38(4):553–557, 2005.

[12] Kollár J., Spišiak M., Direction Vector Grammar In *Scientific Conference on Informatics, 2015 IEEE 13th International* , pages 151–155. IEEE, 2015.

[13] Oncina J., Garcia P., Inferring regular languages in polynomial update time. *Pattern Recognition and Image Analysis*, 1:49–61, 1992.

[14] Dupont P., Miclet L., Vidal E., What is the search space of the regular inference? In *Grammatical Inference and Applications*, pages 25–37. Springer, 1994.

[15] Dupont P., Incremental regular inference. *Grammatical Interference: Learning Syntax from Sentences*, pages 222–237, 1996.

[16] Hrnčič D., Mernik M., Bryant B. R., Javed F., A memetic grammar inference algorithm for language learning. *Applied Soft Computing*, 12(3):1006–1020, 2012.

[17] Fernau H., Algorithms for learning regular expressions from positive data. *Information and Computation*, 207(4):521–541, 2009.

[18] Holub J., The finite automata approaches in stringology. *Kybernetika*, 48(3):386–401, 2012.

[19] Holub J., Melichar B., Approximate string matching using factor automata. *Theoretical Computer Science*, 249(2):305–311, 2000.

[20] Balík M., Dawg versus suffix array. In *Implementation and Application of Automata*, pages 233–238. Springer, 2003.

[21] Koza J. R., *Genetic programming: on the programming of computers by means of natural selection*, volume 1. MIT press, 1992.

[22] O'Neil M., Ryan C. Grammatical evolution. In *Grammatical Evolution*, pages 33–47. Springer, 2003.