Software Tools for Understanding Grammatical Inference Algorithms: Part I – Tools for Regular Grammars and Finite-State Automata

Vaibhav Shah
Researcher
University of Minho
Portugal

Goran D. Putnik
Full Professor
University of Minho
Portugal

Software demonstrators are effective tools to show and understand scientific and engineering concepts in function, and they also allow rapid experiments. In the field of grammatical inference, there is a lack of "ready-to-use" grammar synthesis tools, with simple interfaces showing intermediate stages of the grammar inference process, and the presented work addresses this issue by giving tools for experimentation with regular grammar and finite-state automata, to help students and researchers understand their properties, underlying concepts and applications.

Keywords: Formal Grammars, Grammatical Inference, Finite-State Automata, Regular Grammars, Algorithms, Machine Learning, Demonstrator Software Tools, Industrial Engineering.

1. INTRODUCTION

There is a growing interest in the field of grammatical inference and it has received renewed attention especially in researh communities from various sub-fields of computer science and machine learning. Historically, there were elaborative works and applications of grammatical inference (GI) in the field of syntactic pattern recognition [1], [2]. Later on, the approach started to be widely used in other domains, including the machine learning for problem solving in industrial engineering [3], or in the areas of software engineering [4]. An example of application of GI for an industrial case is given in [5] and [6]. Also, as reported in [7], grammar has been used as a design tool in many disciplines including mechanical engineering [8], [9]. The work in the field of grammatical inference is also gaining momentum through dedicated research commu–nities such as International Community interested in Grammatical Inference (ICGI) [10].

Software demonstrators that showcase functioning of algorithms related to formal grammars can help in further rapid growth of the field of grammatical inference as a research subject. There have been implementations of several algorithms and these are available in the form of libraries, links to which can be found on the webpage of the ICGI [10]. Also, there are tools to visualize and simulate a finite-state automata [11], as well as tools for creating regular grammars and transforming from one model to another model of automata theory [12]. But for the purpose of explaining behavior and outputs of an algorithm, given an input set, in a simple and easy to grasp way, it may be userful to have ready-to-use tools of these algorithms with simple user interfaces. An algorithm implementation in the form of a software demonstrator is obviously an effective way to visualise the algorithm and understand its behaviour. The tasks of explaining and understanding algorithms become much more intuitive, interesting and simpler if there is an interactive way to see the algorithms in function. However, the authors of the presented work have not yet encountered any such software tools for grammatical inference algorithms, that allow the user to make easy correlation between input set and synthesised output grammars. A working implementation of an algorithm in the form of a software tool allows visualising the functioning of the algorithm in a much more effective intuitive manner than reading its description. Such tools can be very useful in explaining the concepts and functionality of the grammatical inference algorithms to students and researchers alike. In the presented work, software tools demonstrating algorithms of regular grammar inference are implemented and explained. The objective is to have easy-to-use tools for experimentation with regular grammar and finite-state automata and to understand their properties. They bring clarity to explanation as well as understanding of the core concepts. The presented tools can also be used to apply grammatical inference in many real-life cases, i.e. they are usable beyond the controlled laboratory use cases. In particular, these tools can be used for engineering applications, to synthesise grammars as models of industrial systems and architectures.

Three algorithms concerning regular grammars inference and minimizing finite-state automata were selected for implementation. These tools were developed as part of the work for a Doctoral thesis [13] and were primarily made to create better understanding of the algorithms. In the following sections, these tools are presented, with description of each tool following a brief outline of the corresponding algorithm. The software tools are explained with their respective graphical user interfaces (GUIs), functionalities, and

Received: October 2015, Accepted: December 2015
Correspondence to: Dr. Vaibhav Shah
Department of Production & Systems (DPS), School of Engineering, University of Minho, Guimarães, Portugal
E-mail: vaibhav.shah@dps.uminho.pt
doi:10.5937/fmet1601083S

examples of their functioning by giving output for a given set of input sentences.

Section 2, very briefly, describes the basic notions of regular grammar synthesis, i.e. definition of regular grammar and the symbolism applied in the tools for grammar synthesis. The authors have followed standard norms for input and output symbols, as globally accepted in formal languages and formal grammars. Section 3 describes Regular Expression Generator (REG) tool demonstrating an algorithm to generate regular expressions from input strings, where it is assumed that there are repetitions of patterns in each string and the algorithm finds these patterns and describes them in the form of a regular expression. Section 4 describes a software tool implemented to infer regular grammar through "successor" method, which is a simple regular grammar inference method. Section 5 describes *k-tails* method for state merging through *k*-eqivalence relations among states of a finite automaton, for arbitrary values of *k*. All the software tools were implemented using LabVIEW graphical programming environment.

2. REGULAR GRAMMAR SYNTHESIS BASIC NOTIONS

2.1 Grammar definition

Consider V, a set of all symbols and V^* a set of all possible strings made of symbols of V.

Definition [14]: A phrase structure grammar G is a 4-tuple

$$G = (T, N, R, \sigma)$$

where, T is a finite set of terminal symbols, N is a finite set of non-terminal symbols, $T \cap N = \Phi$, $T \cup N = V$, $\sigma \in N$ is the start symbol, and R is the finite set of rewriting rules or productions of the form $\alpha \to \beta$, $\alpha, \beta \in V^*$, $\alpha \neq \lambda$.

2.2 Symbolism used in grammatical inference tools

The authors have followed globally accepted norms for the symbols of regular expressions and regular grammars throughout this work, i.e. in all the tools presented. Input strings are always formed either by numbers (0-9) or by lowercase letters (a-z) or by ASCII symbols (+, -, /, *, {, }, and so on…) or by a combination of any of these. The symbols generated by the tools are uppercase letters (A-Z). The tools also take care of these rules of symbols in case the user makes a mistake in writing symbols of input strings. For example, once the user introduces input strings, called input sample, these strings are analysed to extract individual symbols, and if there are any uppercase letters, they are automatically converted into their lowercase counterparts. These extracted symbols, used in the input strings are called the terminal symbols, and they are always in lowercase by global convention. And their set is called the Terminal Alphabet (T). The symbols generated by the tools are always uppercase letters, and are called non-terminal symbols (N).

The terminal symbols are essentially the abstraction of real life complex terms, or processes in case of engineering applications. When the grammatical inference algorithms are applied for engineering applications, or for any other application involving terms made of strings longer than a single character, the terms are abstracted into single character symbols, for simplicity in understanding and ease of processing. The system generates symbols to define the grammar rules, and these are the non-terminal symbols. These symbols are used for describing the synthesised grammar rules. In a real-life scenario, these symbols help re-write the synthesised model in shorter and cleaner format.

3. REGULAR EXPRESSION GENERATOR (REG)

This tool is an implementation of the uv^kw algorithm for regular expression generation from a given set of input strings. The algorithm was given in detail in [15], and a translated description in English was given in [16]. The algorithm's flowchart and outline in section 3.1 are based on the description from [16]. Sections 3.2 and 3.3 describe the REG tool and experiments, respectively.

3.1 The algorithm to generate regular expressions

Figure 1 shows the flowchart of the algorithm uv^kw algorithm as implemented in the REG.

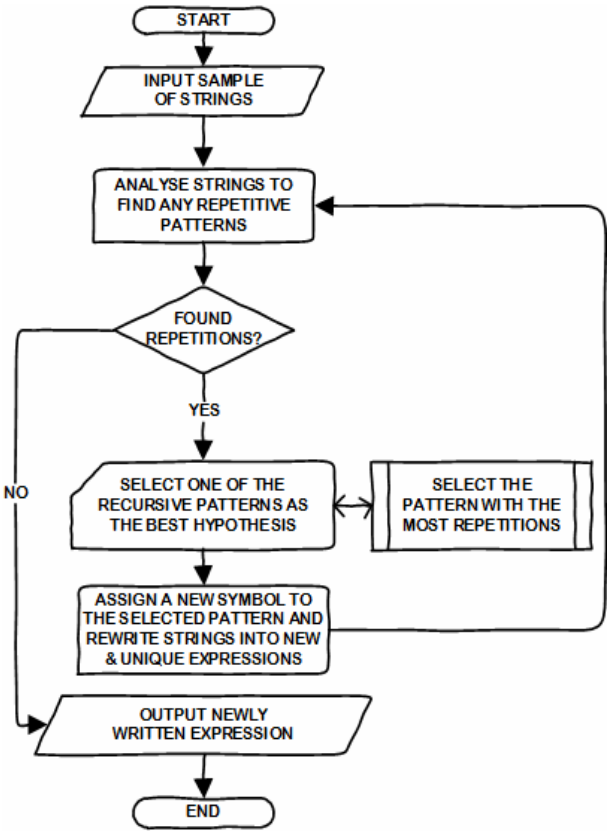

Figure 1. Flowchart of the implemented algorithm

In case when multiple patterns are found to be recursive, some selection criteria have to be applied to select a pattern as the best hypothesis for synthesis of regular expression. In our implementation, the criterion is to select the pattern that has the most number of repetitions in a given string. As stated by [17], in terms of the computation time, this method leads to a realistic program.

Below is outline for the uv^kw algorithm:

Input Sample I;
Do While Found (repetitions in the strings of I)
Build hypotheses for recursion;
Select the best hypothesis h_i based on some criteria;
Generate regular expression r_i for the substring h_i;
Rewrite I by replacing h_i with r_i;
End While
Output: Regular Expression.

The algorithm analyses the input strings to find recursive patterns v and rewrite the strings in the form of u-v^k-w, hence the name. In the output regular expressions, the recursion is denoted with an asterisk "*", and union of two expressions is denoted by a plus symbol "+". A pattern is closed within parentheses "(" and ")". Two symbols within a pattern are related by product operation which is denoted by writing two symbols together, e.g. *ab* is a pattern created as a product of two terminals *a* and *b*. The priorities for the operations in the regular expression notations are in the following order: Star > Product > Union (as followed from [17]).

3.2 The REG software tool implementation

This demonstrator has single graphical user interface. The interface is shown in figure 2. The areas of interest are marked with numbers.

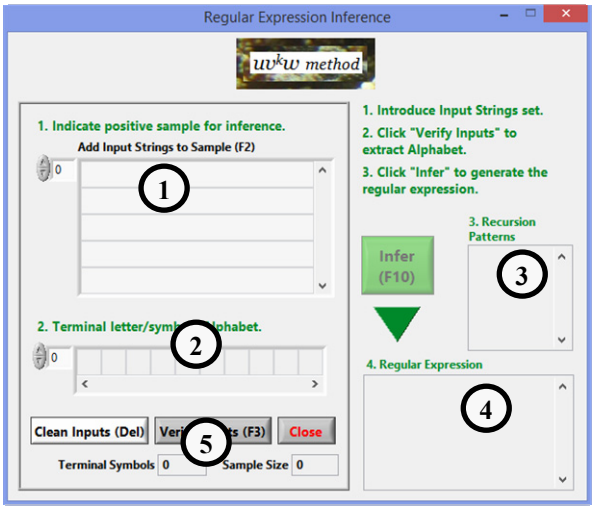

Figure 2. REG graphical user interface

Following is the description of each of the numbered parts of the interface:

(1) – this is where the user introduces input strings. These strings are believed to contain repetitions of certain patterns with one or more of the symbols. Each input string should be introduced in separate line in individual textbox of the list.

(2) – this indicator is to show the alphabet of terminal symbols, i.e. symbols/letters extracted from the input set of strings.

(3) – in this box the recursion patterns as found by the algorithm are enlisted, for the purpose of analysis of inputs and understanding of the algorithm.

(4) – this is where the final output of the algorithm is displayed, in the form of a regular expression.

(5) – these are the buttons for analysis of the input strings. By clicking the "analyse" button, the user tells the tool to extract the unique symbols from the input strings. This step is necessary before the actual inference step.

A program in LabVIEW can be viewed in two modes: front panel to edit user interface controls, and block diagram to write and edit the logical flow of the program. Figure 3 shows block diagram of the actual program written for the REG Tool.

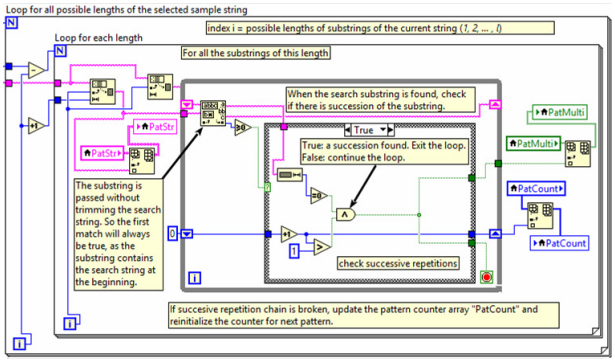

Figure 3. Implementation of the REG tool in LabVIEW

LabVIEW is a graphical programming language, where programs are drawn by "wiring" objects instead of "coding" as opposed to the text based programming languages.

3.3 Synthesis using the REG tool

Following are several experiments performed on the REG tool.

Experiment 1: As seen in figure 4, the REG is provided with 3 strings as input – *aaabc*, *abcabcabcbc*, *abababab* – where *a*, *b* and *c* are individual terminal symbols. In the first cycle, the REG finds repetitions of patterns *a*, *abc*, *bc* and *ab*. As per our criteria the pattern *a* has the most repetitions, hence it is replaced with a non-terminal symbol – in this case the capital letter A, and all the strings are re-written. In the next cycle, more such patterns are selected and re-written with new non-terminal symbols, following the convention. The REG synthesises the following regular expression after the inference process, as given in formula (1).

$$\left((a)^*(bc)^*\right)^*+((a)^*b)^* \qquad (1)$$

When the synthesis process terminated, as seen in figure 4, in total 4 recursion patterns were found. They are listed in table 1, in front of their respectively assigned non-terminal symbols. A brief analysis of the output regular expression from formula (1) is discussed here. As explained before, the priorities of operations in a regular expression are Star>Product>Union. When required, the parentheses are used to explicitly denote the operation, overpassing the priority rule. Hence, in the output expression above, the symbol *a* is a recurring pattern on its own, i.e. *a**, followed by the recursion of product of symbols *b* and *c*, i.e. *bc*, as a single pattern with recursion, i.e. *(bc)**. This forms one complex

pattern *(a)*(bc)**, which itself may appear in recursion, i.e. *((a)*(bc)*)**. Another pattern is the recursion of symbol *a*, i.e. *a** followed by symbol *b*, hence their product *(a)*b*. Also, it is "assumed" that this whole new pattern may also appear 0 or more times, hence a recursion *((a)*b)** which is also a complex pattern, created by multiple symbols and recursions within recursions. Finally, all the complex patterns thus generated are joined by union to create the final regular expression, so that the synthesised regular expression represents the entire input strings sample.

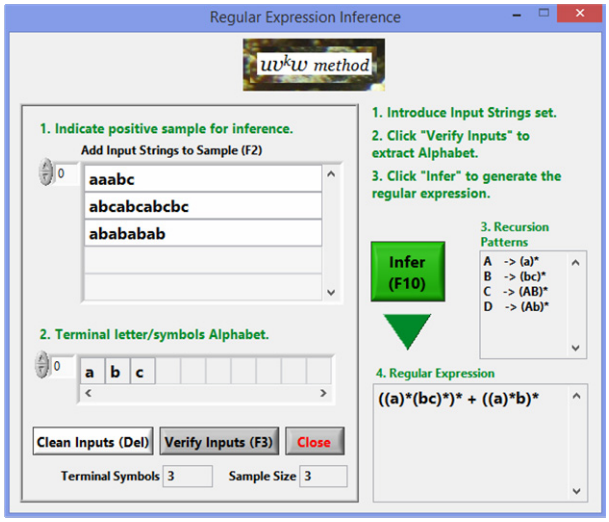

Figure 4. Inference of regular expression through REG

Table 1. Recursion patterns

Non-terminal symbols	Corresponding recursion patterns with stepwise substitutions
A	*a*
B	*bc*
C	AB >> A(*bc*) >> (*a*(*bc*))
D	A*b* >> (*a*)*b*

Table 2 shows more examples of regular expressions synthesised by REG for various input samples.

Table 2. Experiments using the REG

Exp. No.	Input samples	Regular expressions
1	*aabaaababcabc* *abcabaabcbc* *aaaaabc*	*(((a)*b)*c)* + (((a)*b)*c)*bc*
2	*aabaaababcabc* *abcabaabcabc* *aaaaabc*	*(((a)*b)*c)**
3	*sdsdsdsd* *sssssfffff* *sdfsdfsdf*	*(sd)* + (s)*(f)* + ((sd)*(f)*)**

Analysing the above experimental results, the experiments 1 and 2 are almost identical in terms of input strings, except that the second string in experiment 2 has an additional "*a*" before the last occurrence of "*bc*" and this eliminates a part of the regular expression. Implicating this in a manufacturing system application, introducing a single instance of an operation simplified the entire manufacturing process, and this effect was apparently relatively easily if translated into a regular expression in REG. Experiment 3 generated a regular expression as a union of three different expressions because the input strings are varied and show unique patterns, i.e. strings do not share common patterns among them.

4. REGULAR GRAMMAR INFERENCE THROUGH "SUCCESSOR" METHOD

This is one of the simplest methods to generate a regular grammar. The method was developed by two independent teams of researchers [18] & [19]. In section 4.1 a brief description and the algorithm outlines are given. In sections 4.2 and 4.3, the corresponding tool and experimental example, respectively, are presented.

4.1 The "successor" method algorithm

The "successor" method is a simple method to synthesise regular grammar from a given set of strings, by finding the immediate successor of each terminal, and then minimizing the automata to reduce the redundant states. The outline of the algorithm is following:

Input Sample I;
States Q = {};
Build alphabet A = {a_1, a_2 ...};
States counter r = 0;
Find "successors" of λ (empty string) in the sample, i.e. the first letters of any string : (a_i^0...): create state q_0;
For i = 1 to size(alphabet) i.e., for each symbol of the alphabet
Find "successors" of a_i in the sample, i.e. letters that follow a_i in any string : (a_{ji} ...): create state q_i. Where a_{ji} is the successor of a_i in j^{th} string (if exists);
EndFor
Merge states with same successors to build Q;
Output: Finite-State Automaton .

The finite-state automaton thus obtained is in fact the representation of corresponding regular grammar.

4.2 The tool for regular grammar synthesis

The demonstrator tool for regular grammar synthesis by "successor" method is given here, referred to as GISM tool in the remaining parts of the paper. The graphical user interface of the tool is given in figure 5. Following is a brief description of each numbered part on the interface.

(1) – this textbox is to introduce input strings, in order to populate the input sample. The set of input strings should be exhaustive, i.e., it should be positive sample containing all the symbols of the language represented. The input strings are automatically converted to lowercase characters if there is any character from A-Z.

(2) – this list shows the input strings as introduced by the user, in order to verify the input set before processing further.

(3) – the start symbol. By convention, S is used as the start symbol, but it is possible to mention a start symbol of the user's choice. Start symbol can be of multiple characters as well, such as "START" etc.

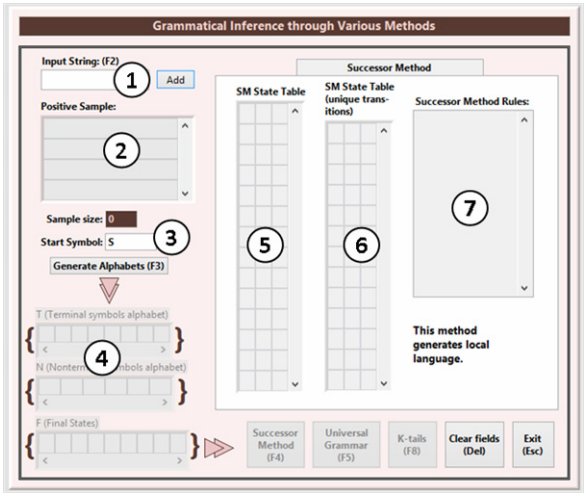

Figure 5. GUI of the successor method synthesis tool

(4) – These are the three sets T, N and F, each containing specific types of symbols. T is the set of terminal symbols. N is the set of non-terminal symbols, written in uppercase. F is the set of final states. Upon clicking "Generate Alphabets" these three sets are automatically filled. T is filled with lowercase unique symbols extracted from the input strings. N is filled with corresponding uppercase symbols for each symbol in T and the start symbol. F is filled with symbols considering the number of unique symbols at the end of each positive string, i.e. the final states.

(5) – this is a table where each row corresponds to a transition from one state (first column) to another (third column), given the input symbol (second column). In other words, each row of this table corresponds to each execution cycle of the algorithm for the given positive sample strings. This table is mainly for analysis and educational purpose. It gives a clear understanding of the functioning of the algorithm.

(6) – this table is the actual state transition table, where each row represents a unique transition. This table is generated after eliminating all the redundant transitions from the table in item (5).

(7) – this box shows the state transition from (6) in the form of grammar productions, or rewriting rules, i.e. the regular grammar G representing the language L of the input strings I.

$$I \subseteq L(G) \qquad (2)$$

4.3 Example run of the tool

Below is an example run of the software tool demonstrating "successor" method in function. Consider the following input set of positive strings.

$$\mathrm{I} = \{\mathrm{ab}, \mathrm{baa}, \mathrm{bbb}, \mathrm{aaba}\}.$$

Figure 6 shows the state transition table, table of unique state transitions and grammar rules for the language representing the input strings. As seen in figure 6, the sets T, N and F are filled based on the input strings – extracting unique terminal symbols to fill T, their uppercase counterparts plus the start symbol to fill N, and a symbol for each of the unique ends of the input strings to fill F (in the given input set, the strings either end with a or b, hence two final states, A and B).

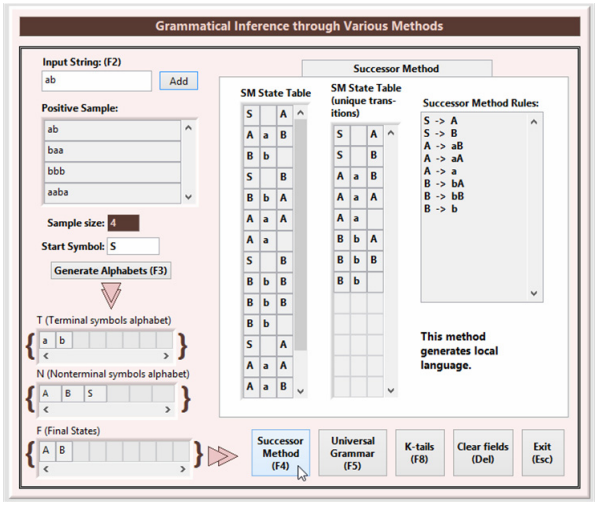

Figure 6. An example run of the GISM tool

The first state table generated is given in table 3.

Table 3. Initial state table with duplicate state transitions

Start state	Input symbol	New state
S		A
A	a	B
B	b	
S		B
B	b	A
A	a	A
A	a	
S		B
B	b	B
B	b	B
B	b	
S		A
A	a	A
A	a	B
B	b	A
A	a	

The execution of this algorithm is pretty straight–forward. For example, before reading each input string, the automaton is considered to be in the start state (S). In addition, each string has a prefix λ which is the symbol for an empty string. The purpose of this empty string is for it to be considered as the first input in order to consider the first actual symbol of the string as a "successor" of something (in this case the empty string).

Now, when the program picks the first string ab of the input sample, the first state transition occurs from the start state S to a new state A with the input string λ, i.e., $S \xrightarrow{\lambda} A$, the first row in table 3. Considering the "successor" of λ, the next input is a, by which the automaton makes a transition towards the next state,

calling it B, hence the state transition $A \xrightarrow{a} B$ is generated, and this transition is written in the second row of table 3. There are no more symbols in the first string, hence the state B becomes the final state, and is added in the transition table without output state. The third row of table 3 indicates this. In this way, one by one, all the strings are read new state transitions are calculated. On the GISM tool's GUI, this table is shown as "SM State table" (see figure 6) in the area numbered 5 as in figure 5.

As seen in table 3, the initial state table generated by the "successor" method, has multiple rows showing the same state transition, with the same (start state, input symbol, new state) triplets.

In the next step, these "duplicate" rows are eliminated, and only unique state transition rows are left in the state transition table, as in table 4.

Table 4. State table with unique state transitions

Start state	Input symbol	New state
S		A
S		B
A	a	A
A	a	B
A	a	
B	b	A
B	b	B
B	b	

To compare the algorithm's functioning against the GISM simulator tool's output, the algorithm's execution steps are given below. The method is to create a state for each symbol of the alphabet T. Recalling the input sample $I = \{ab, baa, bbb, aaba\}$.

The state set Q is created in the following steps:

"successors" of λ in I are (a,b): create state q_0.
"successors" of a in I are (a,b): create state q_1.
"successors" of b in I are (a,b): create state q_2.

Figure 7 shows the state diagram of the inferred grammar. This state diagram was created using JFLAP tool [11]. The states q_1 and q_2 are also the final states, hence marked with double outline. Rewriting the state transition table into the form of rules of a regular grammar, the rules are obtained as shown in table 5. Hence, the grammar synthesis process is completed.

Table 5. Regular grammar rules derived

Rules of synthesised grammar		
S	→	A
S	→	B
A	→	aB
A	→	aA
A	→	a
B	→	bA
B	→	bB
B	→	b

It is important to note that the three steps to grammar synthesis by "successor" method are clearly seen on the user interface of the presented tool. As claimed earlier, this kind of interface allows students to grasp the algorithm's functionality fairly clearly. The tool can be an effective educational tool as well as a tool for automatic grammar synthesis for practical applications.

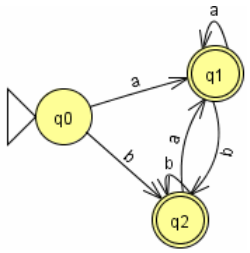

Figure 7. State diagram of inferred grammar

5. *K*-TAILS BASED REGULAR GRAMMAR INFERENCE

Of all the tools presented in this paper, the simulator of *k*-tails method is the most important. One reason is that the method itself is quite tricky and requires an understanding of various other concepts and definitions. The *k*-tails method is applied for minimization of finite automata for a constraint value defined as *k*. It was given in [20] and [21]. However, for the purpose of this work, the authors have adapted examples given in [22]. The method is very clearly explained in literatures [22] and [23], hence the details and the usage of *k*-tails are not discussed in the presented work. But for the purpose of clarity and context, some basic ideas are discussed in section 5.1.

5.1 The method in brief

The *k*-tails method follows procedure of finding *k*-equivalence relation among the states of a canonical grammar [16] through a heuristic state merging process. To understand the *k*-tails method, it is necessary to understand the idea of a formal derivative [23] of a set.

Definition [22, 23]: The formal derivative of a set I of strings with respect to a given symbol $u \in T$ is defined as $D_x I = \{u \,|\, xu \in I\}$. Considering the same input sample $I = \{ab, baa, bbb, aaba\}$ from section 4, following are some formal derivatives of the set I for strings λ, a and b:

$$D_\lambda I = I;\ D_a I = \{b, aba\};\ D_b I = \{aa, bb\}$$

Next, the *k*-tail of I with respect to a string x, formed by symbols of terminal alphabet T, is defined as:

$$g(x, I, k) = \{u \,|\, u \in D_x I, |u| \le k\} \tag{3}$$

Next, the simulator is presented.

5.2 Software simulator for *k*-tails based regular grammar inference

The figure 8 shows graphical user interface for the *k*-tails simulator tool. The parts numbered (1), (2) and (3)

are common from the GISM tool, hence not explained here. The other parts are as follows:

(4) – this table shows all possible distinct derivatives of the input sample. The purpose is to give the whole list at once, always present for reference.

(5) – this control is to specify an arbitrary value of *k*.

(6) – this is the list of *k*-tails of the sample, for the selected value of *k*.

(7) – this table shows the *k*-equivalence classes generated for the selected value of *k*.

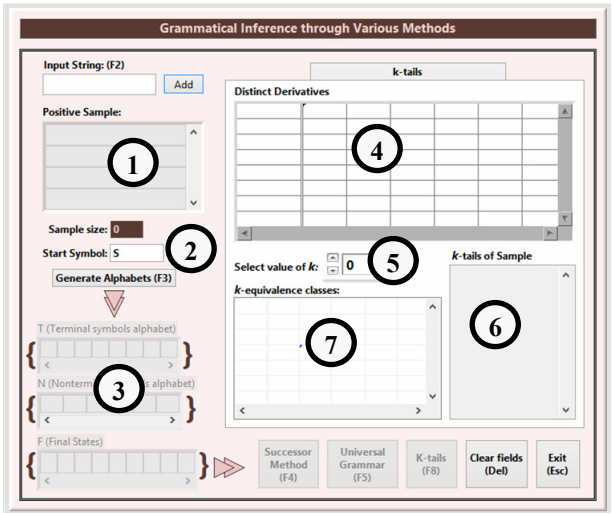

Figure 8. GUI of *k*-tails simulator tool

5.3 Experimenting with the simulator

Running the simulator with the same input sample $I = \{ab, baa, bbb, aaba\}$, the interface is initially filled with all possible distinct derivatives, which do not depend on any selection of value of *k*. These distinct derivatives are created for all possible strings, formed by the symbols of *T*, up to the length equal to the smallest string in input sample. In sample *I*, the smallest string is *ab*, hence the distinct derivatives of *I* are created for strings λ, *a*, *b*, *aa*, *ba*, *bb* (*ab* is omitted deliberately because there are no other string in *I* starting with *ab*, except the string *ab* itself).

As soon as the interface is activated, the user is able to change the values of *k*, and the interface looks as seen in figure 10. With each changing value of *k*, the parts *k*-equivalence classes and *k*-tails of Sample show corresponding values. Shown in Figures 10 and 11 show the whole interface for values of *k*=1 and 2, respectively.

Table 6 shows *k*-equivalence classes for different values of *k*. It has been proven in [16] that the class merging is effective for values of *k* less than or equal to the number of states – 2.

Table 6. *k*-equivalence classes derived

k=1	*k*=2	*k*=3
(S_0, S_2, S_5), (S_1, S_4), (S_3)	(S_0), (S_1, S_4), (S_2), (S_3), (S_5)	(S_0), (S_1), (S_2), (S_3), (S_4), (S_5)
$S_0 = S_2 = S_5 = \{\}$; $S_1 = \{b\}$; $S_3 = \{a\}$; $S_4 = \{b\}$	$S_0 = \{ab\}$; $S_3 = \{a\}$; $S_1 = S_4 = \{b\}$; $S_2 = \{aa,bb\}$; $S_5 = \{ba\}$	$S_0 = \{ab, baa, bbb\}$; $S_1 = \{aba, b\}$; $S_3 = \{a\}$; $S_2 = \{aa, bb\}$; $S_4 = \{b\}$; $S_5 = \{ba\}$

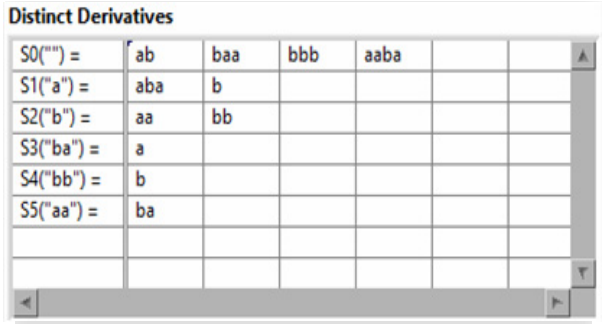

Figure 9. Distinct derivatives of input sample

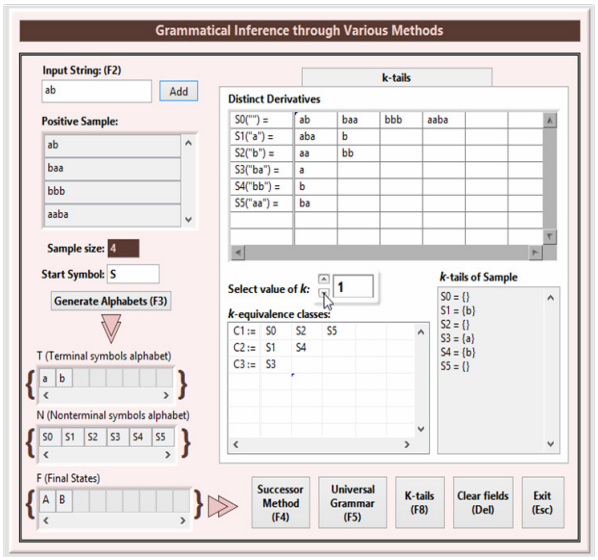

Figure 10. GUI of the k-tails simulator showing values for k=1

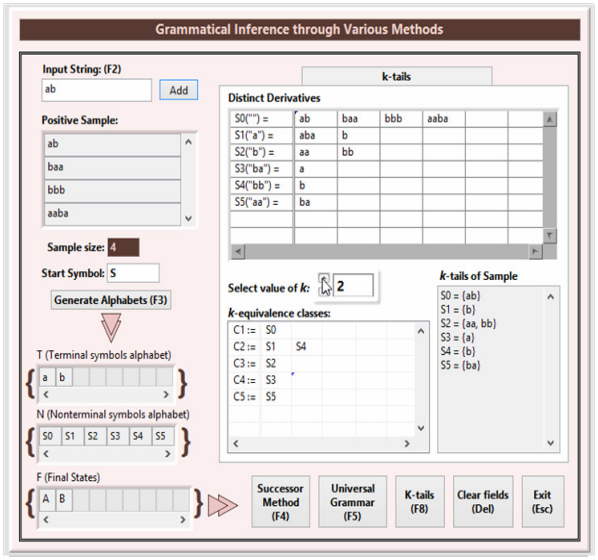

Figure 11. GUI of the *k*-tails simulator showing values for *k*=2

In order to understand the *k*-tails method, and to apply it for a given set of strings, it is necessary to calculate all the derivatives of the set of strings, and then apply the *k*-tails method on those derivatives, for values of *k*. This requires careful calculations, which can be tricky if there are a large number of strings and/or symbols and may lead to frequent errors if calculated manually. Furthermore, for a student, trying to understand the

method and the concept behind the method, it is extremely helpful if there is a simulator which allows visualising all the distinct derivatives as well as different outcomes of *k*-equivalence classes for different values of *k*.

The presented *k*-tails simulator tool addresses both of these issues with a user-friendly interface that highlights all the intermediate information required to understand the final outcome of the *k*-equivalence classes.

6. CONCLUSION

Three algorithms concerning the regular grammar inference were implemented and presented in this paper. The tools can be further enhanced for inclusion of more functions, such as handling more complex symbols. Moreover, the authors expect that these tools can be considered as starting point for further development of user friendly tools for inference of formal grammars to help the growing research and academic community working in the field of grammatical inference, as well as for helping its application in engineering problems.

ACKNOWLEDGMENT

This work has been supported by COMPETE: POCI-01-0145-FEDER-007043 and FCT – Fundação para a Ciência e Tecnologia within the Project Scope: UID/CEC/00319/2013, and by Doctoral Grant (FCT), referenced SFRH/BD/62313/2009.

REFERENCES

[1] Torokhti, A., and Howlett, P.: *Syntactic Methods in Pattern Recognition*, Elsevier, 1974.

[2] Fu, K.S.: *Syntactic Pattern Recognition and Applications*, Prentice-Hall, 1982.

[3] Tanaka, T., Ohsuga, S., and Ali, M. (Eds.): *Industrial and Engineering Applications of Artificial Intelligence and Expert Systems*, Gordon and Breach Publishers, 1996.

[4] Stevenson, A. and Cordy, J.R.: A survey of grammatical inference in software engineering, Science of Computer Programming, Vol. 96, pp. 444-459, 2014.

[5] Putnik, G.D., and Rosas, J.A.: Manufacturing System Simulation Model Synthesis: Towards Application of Inductive Inference, in: *Reengineering for Sustainable Industrial Production*, Chapman & Hall, pp. 259-272, 1997.

[6] Putnik, G.D., and Sousa, R.M.: On formal theories- and formalisms for virtual enterprises. In: *Information Technology for Balanced Manufacturing Systems*, pp. 223-232, Springer US, 2006.

[7] Lee, J., Wyner, G.M., and Pentland, B.T.: Process grammar as a tool for business process design, MIS Quarterly, Vol. 32, No. 4, pp. 757-778, 2008.

[8] Rinderle, J.R.: Grammatical Approaches to Engineering Design, Part II: Modeling Configuration and Parametric Design Using Attributed Grammars, in: *Research in Engineering Design*, Vol. 2, pp. 137-146, 1991.

[9] Schmidt, L.C., and Cagan, J.: GGREADA: A Graph Grammar-Based Machine Design Algorithm, in: *Research in Engineering Design* Vol. 9, No. 4, pp. 195-213, 1997.

[10] ICGI: International Community interested in Grammatical Inference, official webpage of the community: http://www.grammarlearning.org/

[11] Rodger, S.H., and Finley, T.W.: *JFLAP: An Interactive Formal Languages and Automata Package*, Jones & Bartlett Publishers, Sudbury, MA, 2006.

[12] Chesnevar, I.C., and Cobo, M.L.: *Simulators for Teaching Formal Languages and Automata Theory: A comparative Survey*, SeDiCI, 2002.

[13] Shah, V.: *Contribution to Automatic Synthesis of Formal Theories of Production Systems and Virtual Enterprises*, PhD thesis, School of Engineering, University of Minho, Guimarães, 2014.

[14] Hopcroft, J.E., and Ullman, J.D.: Formal Languages and Their Relation to Automata, Addison-Wesley, Reading, Mass., 1969.

[15] Miclet, L.: *Inference de grammaires régulières*, Thése de Docteur-Ingénieur – PhD thesis, Ecole Nationale Superieure des Telecommunications, Paris, France, 1979.

[16] Miclet, L.: Grammatical Inference in: Bunke, H., and Sanfeliu, A. (Eds.) *Syntactic and Structural Pattern Recognition – Theory and Applications*, Series in Computer Science – Vol. 7. World Scientific, 1987.

[17] Miclet, L.: *Structural Methods in Pattern Recognition*, Chapman and Hall, 1986.

[18] Richetin, M., and Vernadat, F.: Efficient regular grammatical inference for pattern recognition, Pattern Recognition, Vol. 17, No. 2, pp. 245-250, 1984.

[19] Rodger R.S., and Rosebrugh, R.D.: Computing a grammar for sequences of behavioural acts, Animal Behaviour, Vol. 27, pp. 737-749, 1979.

[20] Biermann, A.W., and Feldman, J.A.: On the syntheses of finite-state machines from samples of their behaviour, IEEE Trans. On Computers, Vol. 21, pp. 592-597, 1971.

[21] Biermann, A.W., and Feldman, J.A.: A survey of results in grammatical inference, in: Watanabe, S. (Ed.): *Frontiers of Pattern Recognition*, Academic Press, 1972.

[22] Fu, K.S., and Booth, T.L.: Grammatical Inference: Introduction and Survey – Part I, IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. PAMI-8, No. 3, May 1986.

[23] Booth, T.L.: Sequential Machines and Automata Theory, Wiley, New York, 1967.

СОФТВЕРСКИ АЛАТИ ЗА РАЗУМЕВАЊЕ АЛГОРИТАМА ЗА ЗАКЉУЧИВАЊЕ О ГРАМАТИКАМА: ДЕО 1 – АЛАТИ ЗА РЕГУЛАРНЕ ГРАМАТИКЕ И АУТОМАТЕ КОНАЧНИХ СТАЊА

Ваибхав Шах, Горан Д. Путник

Софтверски демонстратори су ефикасно средство за показивање и разумевање функцио–нисања научних и техничких појмова, а омогућавају и брзо експериментисање. У области закључивања о граматикама постоји недостатак готових алата за синтезу граматике са једноставним интерфејсом који приказује непосредне фазе процеса закључивања о граматикама. Приказани рад се бави проблемом обезбеђења алата за експериментисање регуларним граматикама и аутоматима коначних стања у циљу пружања помоћи студентима и истраживачима у разумевању каракатеристика, апликација и појмова који се налазе у основи регуларних граматика и аутомата коначних стања.