Random Generation of Unary Finite Automata over the Domain of the Regular Languages

Lynette van Zijl and Lesley Raitt,
Department of Computer Science, Stellenbosch University, South Africa
lynette@cs.sun.ac.za, lraitt@cs.sun.ac.za
This research was supported by NRF grant number 2053436.

June 5, 2008

Abstract

We show that the standard methods for the random generation of finite automata are inadequate if considered over the domain of the regular languages, for small n. We then present a consolidated, practical method for the random generation of unary finite automata over the domain of the regular languages.

1 Introduction

The random generation of finite automata has received renewed attention in the recent past [2, 9, 12]. However, and this is somewhat surprising, there has been no rigorous, general method proposed for the random generation of the finite automata *over the domain of the regular languages*. The random generation of finite automata has mostly been described in the context of the random generation of the graph representing the finite automaton [2, 4, 8, 12]. It is well-known that different finite automaton graphs can represent the same language (see Figure 3 for an example). Therefore, if one randomly generates finite automata based on their graph representation, one cannot assume that these automata are indeed random (that is, uniformly distributed) over the regular languages. This limits the scope of any experimental pattern analysis over the given set of randomly generated automata, as the pattern analysis can then only pertain to the finite automata and not to the languages represented by the automata. Our two main contributions in this paper are: (a) to show that existing methods for the random generation of n-state finite automata are not necessarily valid over the domain of the regular languages for small n, and (b) to give a method for the random generation of unary n-state finite automata over the domain of the regular languages, which is valid for all values of n.

We briefly describe the standard approach for the random generation of finite automata in Section 3. In Section 4, we then give a method for the random generation of unary deterministic finite automata over the domain of the regular languages. We also point out how to use our method to randomly generate only minimal finite automata. Finally, we conclude in Section 6 and discuss future work on this topic.

2 Background

We assume that the reader has a basic knowledge of automata theory, as for example in [11]. We also assume some statistical background as it pertains to random number generation – a good introduction can be found in [6].

2.1 Finite Automata

Definition 1 *A deterministic finite automaton (DFA) M is a 5-tuple, where $M = (Q, \Sigma, \delta, q_0, F)$, such that Q is a finite nonempty set of states, Σ is a*

finite nonempty alphabet, the transition function δ is the function $\delta : Q \times \Sigma \to Q$, $q_0 \in Q$ is a start state, and $F \subseteq Q$ is the set of final states.

The transition function δ can be extended to strings in the usual way:

$$\delta(q, aw) = \delta(\delta(q, a), w),$$

for any $a \in \Sigma$ and $w \in \Sigma^*$.

A unary DFA (UDFA) is a DFA with one alphabet symbol, and we assume without loss of generality that the alphabet of a UDFA is $\Sigma = \{a\}$. We briefly summarize the relevant properties of UDFAs here. The reader may also refer to [5, 9] for more details.

A UDFA is said to be complete if for every state $q \in Q$ it holds that $\delta(q, a) = q'$, for some $q' \in Q$. A UDFA is connected if for every state $q \in Q$ there is a string $w \in \Sigma^*$ such that $\delta(q_0, w) = q$. Throughout the rest of this paper, we assume that all UDFAs are complete and connected.

The states of any complete and connected n-state UDFA may be renumbered as $q_0, q_1, q_2, \ldots, q_{n-2}, q_{n-1}$, such that state q_0 is the start state and

$$\begin{array}{lcll} \delta(q_i, a) & = & q_{i+1}, & \text{where } 0 \leq i < n-1, \text{ and} \\ \delta(q_{n-1}, a) & = & q_k, & \text{where } 0 \leq k \leq n-1\ . \end{array}$$

The set of states $\{q_k, q_{k+1}, \ldots, q_{n-1}\}$ forms the loop of the UDFA. Clearly, a UDFA can be fully specified by its loop value k and its set of final states. In Figure 1, we illustrate a 4-state UDFA with loop value 1 and final state set $\{q_3\}$.

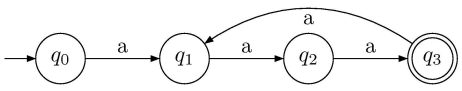

Figure 1: A 4-state UDFA with loop value $k = 1$ and $F = \{q_3\}$.

We say that a UDFA M is minimal if there is no other UDFA with less states which accepts the same language as M. Also, we say that the loop is minimal if encoding its final and nonfinal states as 1 and 0, respectively, results in a binary word w with no repeating pattern. That is, there is no word s and no value t such that $w = s^t$. Such words are known as primitive words. In Figure 1 above, the loop is encoded as 001, and hence forms a primitive word, so that the loop is indeed minimal.

We use regular expressions to describe regular languages in the standard way [11]. A regular expression is a string of symbols, formed from the given alphabet by concatenation, choice ('+'), brackets and the Kleene star. An example of a regular expression is $\epsilon + a + a(aa)^*$. We use the designation *term* for the substrings separated by the + symbol. Therefore, in the regular expression $\epsilon + a + a(aa)^*$, there are three terms: ϵ, a, and $a(aa)^*$. The length of a term in a regular expression is the number of alphabet symbols in that term. For example, the length of term $a(aa)^*$ is three. If two syntactically different regular expressions represent the same regular language, we say that the regular expressions are equivalent (otherwise, nonequivalent).

2.2 Random Number Generation

Random number generation (RNG) algorithms generate sequences of pseudo-random numbers [1], according to a certain algorithm. For an algorithm to be a good RNG algorithm, the numbers in its output sequence must behave as if they are uniformly distributed (that is, the numbers must evenly cover the possibilities over the domain under consideration). To randomly generate objects such as graphs, trees or finite automata [1], one typically uses a random sequence of numbers. Each number is then used to construct the elements of the object in a unique way. After constructing the (elements of the) objects from the sequence of random numbers, we are then assured that the sequence of these constructed objects is random over the domain of all such objects, given that the objects can be enumerated uniquely.

It is important to note here that, if the original random sequence of numbers is modified in any way (say, duplicate numbers are removed), then one cannot necessarily assume that the resulting sequence

[1]We take 'random' to mean 'pseudo-random' unless stated otherwise.

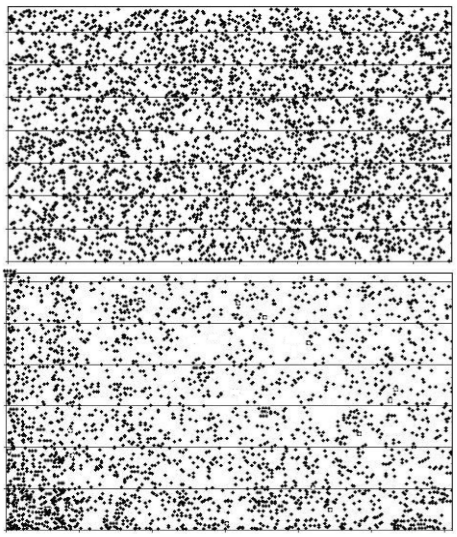

Figure 2: Uniform distribution (top) and non-uniform distribution (bottom).

is still random. Similarly, once a number has been mapped to an object, the objects may not be filtered (again, for example, by removing duplicate objects) without possibly compromising the uniform distribution of the set of objects.

Once an algorithm has been designed to generate a random sequence, one has to test whether its output is actually uniformly distributed. Usually, the algorithm is used to produce extremely long sequences of (supposedly) random output, and the output is then subjected to various statistical tests [7]. As a first measure of quality, one can test whether the output indeed seems to be uniformly distributed over the relevant domain. This is easily visualized in a two-dimensional scatterplot of points (see [6], pages 442–447 for more detail). We show an example of a uniform and non-uniform distribution in Figure 2 below. Note how the points in the uniform distribution evenly covers the area, while in the non-uniform distribution, there appear many clusters of points as well as gaps where there are no points at all.

Throughout this paper, we assume that we have a random number generator which produces evenly distributed random sequences of numbers.

3 Standard Methods for the Random Generation of Finite Automata

The standard approach to the random generation of finite automata is to generate a random stream of numbers, and then use these numbers to uniquely generate the elements of the finite automata [2, 8]. In particular, if one randomly generates n-state UDFAs, then the numbers from the random number stream is used to construct the transition function and the set of final states [2]. Note that different numbers must map to different objects in the domain in order to maintain statistical randomness properties. Now, since two UDFAs with different graph representations may accept the same language (for an example, see Figure 3), it follows that the resulting stream of random UDFAs is random over the domain of the graphs of the UDFAs, but not *necessarily* random over the domain of the regular languages.

As an example, consider Figure 4. There, we used the bitstream method [2, 12] to randomly generate 6-state UDFAs, and plotted their graphs over the domain of all possible 6-state UDFA graph representations (note the uniform distribution in the uppermost scatterplot in Figure 4). For each UDFA, we then considered the regular language associated with it, and represented the languages on the bottom scatterplot. Here, notice how the dots cluster in the bottom left corner (this represents the empty language). Clearly, even though the UDFAs were randomly generated with respect to their graph representations, this randomness does not translate to randomness over the domain of the regular languages.

Recent results by Domaratzki *et al.* [5] on the number of distinct languages accepted by n-state finite automata indeed indicate that this standard approach is not valid if randomly generated automata are to be considered random over the domain of the

[2] In a UDFA, there is only one possible start state, and only one alphabet symbol, which means that it is not necessary to construct these elements.

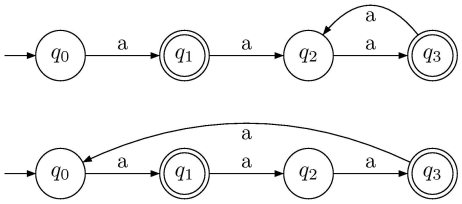

Figure 3: Two UDFAs with different graphs that both accept the language $a(aa)^*$.

n	5	10	20	50
L_n/G_n	78.75	86.54	93.09	97.24
n	100	400	800	
L_n/G_n	98.62	99.65	99.83	

Table 1: Significance of domain of regular languages as n grows.

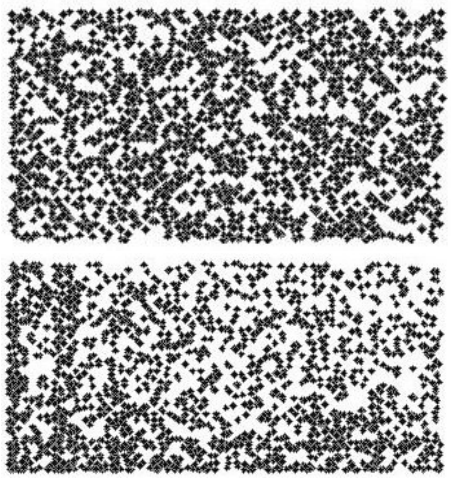

Figure 4: Bitstream method over graph domain for 6-state UDFAs (top) and over regular language domain (bottom).

regular languages, for smaller values of n. In Table 1, we list the percentage of the number L_n of distinct languages accepted by n-state UDFAs, divided by the number G_n of distinct graphs of n-state UDFAs. As n tends towards infinity, this value tends towards 100%. That is, as n grows, the difference between the number of distinct languages and the number of distinct graphs becomes insignificant. However, for smaller n, there is a significant difference. Similar results can be concluded for both nondeterministic unary finite automata and binary automata, from the results given by Domaratzki [5]. Hence, there is a need for a random generation method for finite automata over the domain of the regular languages, which is valid for small n.

4 Random Generation of UDFAs over the Domain of the Regular Languages

In this section, we present a method to randomly generate a sequence of n-state UDFAs over the domain of the regular languages. In order to do this, we require an enumeration of the unary regular languages accepted by n-state UDFAs, so that we can map a given random number to a specific regular language (as opposed to mapping it to a transition table or graph representation, as in most previous approaches).

A regular language can be described either by the finite automaton accepting that language, or by a regular expression. We choose, without loss of generality, to describe regular languages with regular expressions. It is important to note again that there can be multiple syntactically different regular expressions that describe the same language – that means that an enumeration must count syntactically different regular expressions that accept the same language, only once. Thus, our enumeration of the unary regular languages is based on counting only the nonequivalent unary regular expressions.

Once we have an enumeration of the nonequivalent unary regular expressions, we can map our random number stream to nonequivalent regular expressions, and hence to nonequivalent regular languages – this solves the problem inherent to the previous approaches, since different random numbers cannot map to the same regular language anymore. Once the random number is mapped to a regular expression, we construct the UDFA that accepts the lan-

guage represented by that regular expression.

For the rest of this paper, we take regular expression to mean a regular expression that represents a unary language that is accepted by a complete and connected n-state UDFA.

Our method consists of the following steps:

(1) Generate a random sequence $S = s_0, s_1, \ldots$ of integers between 0 and $2^n - 1$.

(2) For each number s_i in S, associate s_i with a regular expression r, where r is the s_i-th regular expression in the enumeration of the unary regular languages accepted by n-state UDFAs.

(3) Construct an n-state UDFA associated with r. If so specified, construct the minimal UDFA.

Step (2) of the method above requires the mapping of a number to a regular expression. In order to achieve this mapping, we enumerate [3] the unary regular expressions algorithmically, so that for any random number s_i, we can find the s_i-th regular expression. For ease of exposition, we consider three separate classes in the algorithmic enumeration: (i) unary regular expressions for finite regular languages, (ii) unary regular expressions that contain terms of the form a^*, and (iii) unary regular expressions that contain terms of the form $(a^y)^*$, for $y > 1$.

This classification is based on the known combinatorial values for the number of unary regular languages, and we now consider the enumeration in detail in the next sections.

4.1 Regular Expressions for Unary Finite Language

We know that there are exactly 2^{n-1} distinct finite languages (see [5], Th. 14(b)) that are accepted by n-state UDFAs. Therefore, for a given n and given random number s_i, if s_i falls in the interval $0 \leq s_i < 2^{n-1}$, we map s_i to a regular expression which represents a finite regular language (if s_i falls outside this interval, we map s_i to a regular expression which represents an infinite regular language – see Section 4.2 and Section 4.3).

Within these 2^{n-1} regular expressions, we have to consider all the nonequivalent regular expressions in a certain order. Since the regular expressions represent finite languages, it is clear that no term may contain the Kleene star, and that no term can have length larger than $n-2$. Therefore, the regular expression must be of the form $a^{i_1} + a^{i_2} + \ldots + a^{i_j}$, for $0 < j \leq n-1$ and $0 \leq i_1, i_2, \ldots, i_j \leq n-2$. To exclude equivalent regular expressions, we simply have to stipulate that $0 \leq i_1 < i_2 < \ldots < i_j \leq n-2$. Generating all possible regular expressions of the form described above then uniquely gives all the nonequivalent regular expressions, as required.

An ordering on the regular expressions follows directly by the number of terms and the lengths of the terms in the regular expressions, in ascending order. For example, the regular expression $a + aaa$ would occur before $a + aa + aaa$. We partially list the ordering of these regular expressions in Table 2 for $n = 6$. Here i represents the i-th regular expression in the ordering.

Hence, given a random integer $0 \leq s_i < 2^{n-1}$, if s_i is the m-th number in this interval, we pinpoint the correct regular expression by taking the m-th regular expression in the ascending order of number and lengths of terms.

Given the regular expression associated with the random number s_i, we have to construct a UDFA accepting the language represented by the regular expression. That is, we have to specify the set of final states, and the loop value. Since this is a finite regular language, we know that the loop may not contain any final states. Moreover, it is easy to see that the regular expression contains a term a^t iff state q_t is a final state in the UDFA, and q_t is not in the loop of the UDFA. Therefore, the number of final states simply corresponds to the number of terms in the regular expression, and the set of final states are given by the length of each term. Note that there may be different possible loop values (see Table 2) for each final

[3] It is not difficult to enumerate the unary regular expressions [5]; however, the manner of enumeration in our case must exclude regular expressions which represent the same languages, and must ideally aid in an efficient implementation. Lee and Shallit [10] give a general grammar-based approach, but their enumeration is not easily adaptable for our purposes as they are concerned with the syntactic length of the regular expression, and do not exclude equivalent expressions.

i	Regular expressions	Final states	Possible loop values $\{q_0, \ldots, q_{n-1}\}$
0	$\emptyset$	000000	0,1,2,3,4,5
1	ϵ	100000	1,2,3,4,5
2	a	010000	2,3,4,5
3	ϵ+a	110000	2,3,4,5
10	a+aa	011000	3,4,5
11	ϵ+a+aa	111000	3,4,5
22	a+aa+aaa	011100	4,5
23	ϵ+a+aa+aaa	111100	4,5
30	a+aa+aaa+aaaa	011110	5
31	ϵ+a+aa+aaa+aaaa	111110	5

Table 2: Partial enumeration of regular expressions for finite languages accepted by UDFAs with 6 states.

state set – that is, there are different UDFAs corresponding to this regular expression. We then, from an independent random stream, choose one of these UDFAs (say the final state with the largest index is q_p, then there are $n-1-p$ possible loop values, and if the generated random number is $0 \leq t < n-1-p$, we let the loop value be $n-1-t$).

Example 1 *Suppose we have $n = 6$, and the random number 22. Then the associated regular expression is $a + aa + aaa$. The random UDFA then has the final state set $F = \{q_1, q_2, q_3\}$, and may have loop value $k = 4$, or $k = 5$ (see Table 2).*

4.2 Regular Expressions that include a^*

For this case, we enumerate all the nonequivalent regular expressions that contain the Kleene star applied to one alphabet symbol; that is, a^*.

We list all the regular expressions in this case as for the finite case, namely, ascending according to the number of terms and length of terms, excluding equivalent regular expressions.

We note the following equivalences:

- Any regular expression of the form a^*a^t is equivalent to a^ta^*, and hence we need to consider only regular expressions with terms that end on a^*.
- For $0 \leq s \leq t \leq n-1$, it holds that $a^sa^* + a^ta^*$ is equivalent to a^sa^*. Therefore, all regular expressions in this case can be simplified to the form $a^{i_1}+a^{i_2}+\ldots+a^{i_{j-1}}+a^{i_j}a^*$, for $0 < j \leq n-1$ and $0 \leq i_1 < i_2 < \ldots < i_j \leq n-1$.
- Finally, $a^s + a^sa^*$ simplifies to a^sa^*, so that the condition $i_j - i_{j-1} > 1$ excludes such equivalences.

A simple counting argument shows that there are 2^{n-1} non-equivalent different regular expressions to consider in this case. Therefore, if the original random number s_i falls in the range $2^{n-1} \leq s_i < 2^n$, we map s_i to this second case.

We give a partial listing of the ordering of these regular expressions in Table 3 for $n = 6$.

To find a UDFA from the given regular expression, we use the same approach as in the finite case. We note that there may be any combination of final states before the loop, representing the finite part of the language. Therefore, if the last term in the regular expression is a^ma^*, we take every term with length t less than m, and add the single final state q_t to the set of final states. The last term a^ma^* in the regular expression is then represented by setting states q_m through to q_{n-1} as final states, and choosing the loop value k randomly among states q_m to q_{n-1}.

Example 2 *Suppose we have $n = 6$, and the random number 54. Then the associated regular ex-*

i	Regular expressions	Final states	Possible loop values $\{q_0, \ldots, q_{n-1}\}$
32	a^*	111111	0,1,2,3,4,5
33	aa^*	011111	1,2,3,4,5
34	aaa^*	001111	2,3,4,5
35	$\epsilon + aaa^*$	101111	2,3,4,5
48	$aa + aaaaa^*$	001001	5
49	$\epsilon + aa + aaaaa^*$	101001	5
54	$a + aa + aaaaa^*$	011011	4,5
55	$\epsilon + a + aa + aaaaa^*$	111011	4,5
62	$a + aa + aaa + aaaaaa^*$	011101	5
63	$\epsilon + a + aa + aaa + aaaaaa^*$	111101	5

Table 3: Partial enumeration of regular expressions containing a^* for 6-state UDFAs.

pression is $a + aa + aaaaa^*$*, the final state set is* $F = \{q_1, q_2, q_4, q_5\}$*, and the loop value may be either* $k = 4$ *or* $k = 5$*.*

4.3 Regular Expressions that include $(a^y)^*$

For this case, we have to enumerate all the nonequivalent regular expressions that contain a term $(a^y)^*$.

Each regular expression in this case may contain terms with no Kleene star; this is the finite part. The regular expression may also contain terms of the form $a^s(a^t)^*$. Since the regular expression represents a language accepted by an n-state UDFA, and a UDFA has only one loop value, it follows that the regular expression has the general form $a^{i_1}(a^{t_1})^* + a^{i_2}(a^{t_2})^* + \ldots + a^{i_{j-1}}(a^{t_{j-1}})^* + a^{i_j}(a^{t_j})^*$, for $0 < j \leq n-1$ and $0 \leq i_1 < i_2 < \ldots < i_j \leq n-1$, and $0 \leq t_m \leq n$ for $1 \leq m \leq j$, such that all the non-zero t_m are equal.

We generate all possible combinations of the general form given above, excluding equivalent regular expressions. Similar to the a^* case in Section 4.2, we note that $a^{t_1} + a^{t_2}(a^s)^*$ is equivalent to $a^{t_1}(a^s)^*$ if $t_2 - s = t_1$. For example, $a + a^3 + a^5(a^2)^*$ is equivalent to $a + a^3(a^2)^*$, which is again equivalent to $a(a^2)^*$. Other equivalences occur between two expressions of the form $a^{i_1}(a^{t_1})^*$ and $a^{i_1}(a^{t_2})^*$, where t_1 and t_2 are not relatively prime. For example, $a(a^6)^* + a^3(a^6)^* + a^5(a^6)^*$ is equivalent to both $a + a^3(a^4)^* + a^5(a^4)^*$ and $a + a^3 + a^5(a^2)^*$. In such cases, we always enumerate the regular expression with the smallest value of t in the term $(a^t)^*$.

The table for $n = 6$ is rather large in this case, and we refrain from listing it.

Given all the nonequivalent regular expressions, we can generate the UDFA in a similar manner as before. We generate a final state q_t if a term a^t occurs in the regular expression. For all terms $a^m(a^s)^*$, we generate final states q_{m+ps} for $p \geq 0$ and $0 < ps \leq n-1$. An initial loop value of $k = n - s$ is chosen. However, if either the resulting loop or the resulting UDFA forms a nonprimitive word $w = u^v$, then the loop value is chosen randomly from the set $\{n-s, n-2s, \ldots, n-vs\}$.

4.4 Analysis

Having presented a method which purportedly randomly generates UDFAs over the domain of the regular languages, we now have to test this claim.

In Figure 5 we give a simple distribution analysis for a stream of 2500 randomly generated 6-state UDFAs based on our method. One can clearly see that the distribution over the domain of the regular languages is uniform (compare this to Figure 4).

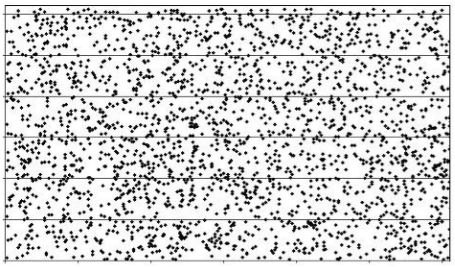

Figure 5: Randomly generated 6-state UDFAs over the domain of the regular languages.

5 Random Generation of Minimal UDFAs over the Domain of the Regular Languages

Our method allows for an almost trivial adaptation to the generation of minimal UDFAs. We know from Domaratzki [5] that an n-state UDFA with loop value k can be minimal only if its loop is minimal, and states q_{k-1} and q_{n-1} have opposite finality. Hence, for all three cases, we simply have to ensure that we select a minimal loop, together with the correct finalities for states q_{n-1} and q_{k-1}.

For finite languages, the loop value must be $n-1$ in order to ensure loop minimality (otherwise, the loop can be encoded as 0^t for some $t > 1$). Therefore, all minimal UDFAs in this case have loop value $n-1$, with q_{n-1} a nonfinal state and therefore q_{n-2} a final state. We observe from Table 2 that there are 2^{n-2} such cases. Therefore, to generate a minimal n-state UDFA, we generate a random number $0 \leq s_i < 2^{n-2}$, and follow the same procedure as for non-minimal UDFAs, but make our selection only from these minimal UDFAs.

The case where the regular expression contains a^* follows the same argument as for the finite case: there are 2^{n-2} regular expressions that contain a term $a^{n-1}a^*$, and we can choose from those cases a UDFA with state q_{n-1} as one of the final states, and state q_{n-2} as a nonfinal state. Again, note that the loop in the a^* case must consist of all final states, and therefore the only possibility for a minimal loop is the one where the loop value is $n-1$.

For the last case, we choose a loop value such that the states in the loop form a primitive word w, in order to ensure a minimal loop. Then we choose the states before the loop such that state q_{k-1} has the opposite finality to state q_{n-1}.

6 Conclusion and Future Work

We pointed out the problem associated with existing methods for the random generation of finite automata over the domain of the regular languages, for small n. We then presented a method for the random generation of UDFAs over the domain of the regular languages, which is valid for any value of n.

It may be possible to extend our method to the case of nondeterministic unary finite automata. However, for binary automata, the enumeration of the regular expressions become quite complex, and another method is needed in this case for small values of n.

References

[1] Barcucci, E., Del Lungo, A., Pergola, E., Random Generation of Trees and other Combinatorial Objects. Theoretical Computer Science **218** (1999), 219–232.

[2] Champarnaud, J.-M., Hansel, G., Paranthoën, T., Ziadi, D., NFAs Bitstream-based Random Generation. *Proceedings of the 4th Conference on Descriptional Complexity of Formal Systems*, August 2002, London, Ontario, Canada.

[3] Champarnaud, J.-M., Paranthoën, T., Random Generation of DFAs. Theoretical Computer Science **330** (2005), 221–235.

[4] De Beijer, N., Watson, B.W., Kourie, D.G., Stretching and Jamming of Automata. *Proceedings of SAICSIT 2003*, Johannesburg, South Africa, September 2003.

[5] Domaratzki, M., Kisman, D., Shallit, J., On the Number of Distinct Languages accepted by Finite Automata with n States. Journal of Au-

tomata, Languages and Combinatorics **7** (2002), 469–486.

[6] Law, A.M., Kelton, D., *Simulation Modeling and Analysis.* McGraw-Hill, 2000.

[7] L'Ecuyer, P., Testing Random Number Generators. *Proceedings of the 1992 Winter Simulation Conference*, IEEE Press, Dec. 1992, 305–313.

[8] Leslie, T.K.S., Efficient Approaches to Subset Construction. MSc thesis, University of Waterloo, Waterloo, Canada, 1994.

[9] Nicaud, C., Average State Complexity of Operations on Unary Automata. *Proceedings of MFCS'99*, Lecture Notes in Computer Science **1672** (1999), 231–240.

[10] Shallit, J., Lee, J., Enumerating Regular Expressions and their Languages. *Proceedings of CIAA2004*, Lecture Notes in Computer Science **3317** (2005), 2–22.

[11] Sipser, M., *Introduction to the Theory of Computation.* PWS Publishing Company, 1997.

[12] Van Zijl, L., Random Number Generation with Symmetric Difference NFAs. *Proceedings of CIAA2001*, Lecture Notes in Computer Science **2494** (2002) 263–273.