Finite State Registered Automata and their uses in Natural languages

Yael Cohen-Sygal Shuly Wintner

Department of Computer Science
University of Haifa
yaelc@cs.haifa.ac.il shuly@cs.haifa.ac.il

Abstract. We extend *finite state registered automata* (FSRA) to account for medium-distance dependencies in natural languages. We provide an extended regular expression language whose expressions denote arbitrary FSRAs and use it to describe some morphological and phonological phenomena. We also define several dedicated operators which support an easy and efficient implementation of some non-trivial morphological phenomena. In addition, we extend FSRA to *finite-state registered transducers* and demonstrate their space efficiency.

1 Introduction

Finite-state (FS) technology is considered adequate for describing the morphological processes of natural languages since the pioneering works of [1] and [2]. Several toolboxes provide extended regular expression description languages and compilers of the expressions to finite state automata (FSAs) and transducers (FSTs) [3–5]. While FS approaches for natural languages processing have generally been very successful, it is widely recognized that they are less suitable for non-concatenative phenomena. In particular, FS techniques are assumed not to be able to efficiently account for medium-distance dependencies, whereby some elements that are related to each other in some deep-level representation are separated on the surface. These phenomena do not lie outside the descriptive power of FS systems, but their implementation can result in huge networks that are inefficient to process.

To constrain dependencies between separated morphemes in words, [6] propose *flag diacritics*, which add features to symbols in regular expressions to enforce dependencies between separated parts of a string. The dependencies are forced by different kinds of unification actions. In this way, a small amount of finite memory is added, keeping the total size of the network relatively small. The main disadvantage of this method is that it is not formally defined, and its mathematical and computational properties are not proved. Furthermore, flag diacritics are manipulated at the level of the extended regular expressions, although it is clear that they are compiled into additional memory and operators

in the networks themselves. The presentation of [6] and [7] does not explicate the implementation of such operators and does not provide an analysis of their complexity. Moreover, they do not present any dedicated regular expression operations for non-concatenative processes.

A related formalism is *vectorized finite state automata* (VFSA) [8], where both the states and the transitions are represented by vectors of elements of a partially ordered set. Two kinds of operations over vectors are defined: *unification* and *overwriting*. The vectors need not be fully determined, as some of the elements can be unknown (free). In this way information can be moved through the transitions by the overwriting operation and traversing these transitions can be sanctioned through the unification operation. The free symbols are also the source of the efficiency of this model, where a vector with k free symbols actually represents t^k vectors, t being the number of different values that can be stored in the free places. As one of the examples of the advantages of the model, [8] shows that it can efficiently solve the problem of 32-bit binary incrementor. The goal of this example is to construct a transducer over $\Sigma = \{0, 1\}$ whose input is a number in 32 bit binary representation and whose output is the result of adding 1 to the input. The naïve solution is a transducer with only 5 states and 12 arcs, but this transducer is neither sequential nor sequentiable. A sequential transducer for an n-bit binary incrementor would require 2^n states and a similar number of transitions. Using vectorized finite state automata, a 32-bit incrementor is constructed where first, using overwriting, the input is scanned and stored by the vectors, and then, using unification, the result is calculated where the carry can be computed from right to left. This allows a significant reduction in the network size. The main disadvantage of VFSA lies in the fact that it significantly deviates from the standard methodology of developing finite-state devices, and integration of vectorized automata with standard ones remains a challenge. Moreover, it is unclear how, for a given problem, the corresponding network should be constructed: programming with vectorized automata seems to be unnatural, and no regular expression language is provided for them.

Finite state registered automata (FSRA) ([9]) augment finite state automata with finite memory (registers) in a restricted way that saves space but does not add expressivity. The number of registers is finite, usually small, and eliminates the need to duplicate paths as it enables the automaton to 'remember' a finite number of symbols. Each FSRA defines an alphabet, Γ, whose members can be stored in registers. In this model, each arc is associated not only with an alphabet symbol, but also with a series of actions on the registers. There are two kinds of possible actions, *read* and *write*. The read action, denoted R, allows traversing an arc only if a designated register contains a specific symbol. The write action, denoted W, allows traversing an arc while writing a specific symbol into

a designated register. Then, the FSRA model is extended to allow up to k register operations on each transition, where k is determined for each automaton separately. The register operations are defined as a sequence (rather than a set), in order to allow more than one operation on the same register over one transition. [9] prove that FSRAs are equivalent to FSAs, and use them to efficiently describe some non-concatenative phenomena of natural languages, including interdigitation (root-and-pattern morphology) and limited reduplication.

In this work we extend the model of *FSRA* to account for medium-distance dependencies in natural languages. We provide an extended regular expression language whose expressions denote FSRAs in section 2. Section 3 defines several dedicated operators which support an easy and efficient implementation of some non-trivial morphological phenomena. We then extend FSRA to finite-state registered *transducers* in section 4 . Furthermore, the model is evaluated through an actual implementation in section 5. We conclude with a comparison with similar approaches and suggestions for future research.

2 A regular expression language for FSRAs

The first limitation of [9] is that no regular expression language is provided for constructing FSRAs. We begin by proposing such a language, the denotations of whose expressions are FSRAs. In the following discussion we assume the regular expression syntax of XFST ([7]) for basic expressions[1].

Definition 1. *Let* $Actions_n^\Gamma = \{R, W\} \times \{0, 1, 2, \ldots, n-1\} \times \Gamma$*, where* n *is the number of registers and* Γ *is the register alphabet. If* R *is a regular expression and* $\vec{a} \in \left(Actions_n^\Gamma\right)^+$ *is a series of register operations, then the following are also regular expressions:* $\vec{a} \triangleright R$*,* $\vec{a} \triangleright \triangleright R$*,* $\vec{a} \triangleleft R$ *and* $\vec{a} \triangleleft \triangleleft R$*.*

We now define the denotation of each of the above expressions. Let R be a regular expression whose denotation is the FSRA A, and let $\vec{a} \in \left(Actions_n^\Gamma\right)^+$. The denotation of $\vec{a} \triangleleft R$ is an FSRA A' obtained from A by adding a new node, q, which becomes the initial node of A', and an arc from q to the initial node of A; this arc is labeled by ϵ and associated with $\vec{a}$. Notice that in the regular expression $\vec{a} \triangleleft R$, R and $\vec{a}$ can contain operations on joint registers. In some cases, one would like to distinguish between the registers used in $\vec{a}$ and in R. Usually, it is up to the user to correctly manipulate the usage of registers, but in some cases automatic distinction seems desirable. For example, if R includes a circumfix operator (see below), its corresponding FSRA will contain register operations created automatically by the operator. Instead of remembering that circumfixation always uses register 1, one can simply distinguish between the

[1] In particular, concatenation is denoted by space and ϵ is denoted by 0.

registers of $\vec{a}$ and R via the $\vec{a} \triangleleft \triangleleft R$ operator. This operator has the same general effect as the previous one, but the transition relation in its FSRA uses fresh registers which are added to the machine.

In a similar way, the operators $\vec{a} \triangleright R$ and $\vec{a} \triangleright \triangleright R$ are translated into networks. The difference between these operators and the previous ones is that here, the register operations in $\vec{a}$ are executed *after* traversing all the arcs in the FSRA denoted by R. It is easy to show that every FSRA has a corresponding regular expression denoting it.

Example 1 *Consider the case of vowel harmony in Warlpiri [10], where the vowel of suffixes agrees in certain aspects with the vowel of the stem to which it is attached. A simplified account of the phenomenon is that suffixes come in two varieties, one with 'i' vowels and one with 'u' vowels. Stems whose last vowel is 'i' take suffixes of the first variety, whereas stems whose last vowel is 'u' or 'a' take the other variety. The following examples are from [10] (citing [11]):*

1. *maliki+kiḷi+ḷi+lki+ji+li*
   (dog+PROP+ERG+then+me+they)
2. *kuḍu+kuḷu+ḷu+lku+ju+lu*
   (child+PROP+ERG+then+me+they)
3. *minija+kuḷu+ḷu+lku+ju+lu*
   (cat+PROP+ERG+then+me+they)

An FSRA that accepts the above three words is denoted by the following complex regular expression:

```
define LexI [m a l i k i]; % words ending in 'i'
define LexU [k u d u];     % words ending in 'u'
define LexA [m i n i j a]; % words ending in 'a'
! Join all the lexicons and write to register 1
! 'u' or 'i' according to the stem's last vowel.
define Stem [LexI ◁ <(W,1,i)>] |
            [ [LexU | LexA] ◁ <(W,1,u)>];
! Traverse the arc only if the scanned symbol is
! the content of register 1.
define V [<(R,1,i)> ▷ i] | [<(R,1,u)> ▷ u];
define PROP [+ k V l V]; % PROP suffix
define ERG  [+ l V];     % ERG suffix
define Then [+ l k V];   % suffix indicating 'then'
define Me   [+ j V];     % suffix indicating 'me'
define They [+ l V];     % suffix indicating 'they'
! define the whole network
define WarlpiriExample Stem PROP ERG Then Me They;
```

Register 1 stores the last vowel of the stem, eliminating the need to duplicate paths for each of the different cases. The lexicon is divided into three separate lexicons (LexI, LexU, LexA), one for each word ending ('i', 'u' or 'a' respectively). The separate lexicons are joined into one (the variable Stem) and when reading the last letter of the base word, its type is written into register 1. Then, when suffixing the lexicon base words, the variable V uses the the content of register 1 to determine which of the symbols 'i', 'u' should be scanned and allows traversing the arc only if the correct symbol is scanned. Note that this solution is applicable independently of the size of the lexicon, and can handle other suffixes in the same way.

Example 2 *Consider the following Arabic nouns:* qamar *(moon),* kitaab *(book),* $ams *(sun) and* daftar *(notebook). The definite article in Arabic is the prefix ''*al*', which is realized as ''al' when preceding most consonants; however, the 'l' of the prefix assimilates to the first consonant of the noun when the latter is 'd', '$', etc. Furthermore, Arabic distinguishes between definite and indefinite case markers. For example, nominative case is realized as the suffix '*u*' on definite nouns, '*un*' on indefinite nouns. Examples of the different forms of Arabic nouns are:*

word	*nominative definite*	*nominative indefinite*
qamar	*'alqamaru*	*qamarun*
kitaab	*'alkitaabu*	*kitaabun*
$ams	*'a$$amsu*	*$amsun*
daftar	*'addaftaru*	*daftarun*

The FSRA of Figure 1 accepts all the nominative definite and indefinite forms of these nouns. In order to account for the assimilation, register 2 stores information about the actual form of the definite article. Furthermore, to ensure that definite nouns occur with the correct case ending, register 1 stores information of whether or not a definite article was seen. This FSRA can be denoted by the following regular expression:

```
! Read the definite article (if present).
! Store in register 1 whether the noun is definite
! or indefinite.
! Store in register 2 the actual form of the
! definite article.
define Prefix [0 ◁ <(W,1,indef)>] |
              ['al ◁ <(W,1,def),(W,2,l)>] |
              ['a$ ◁ <(W,1,def),(W,2,$)>] |
```

```
              ['ad◁ <(W,1,def),(W,2,d)>];
! Normal base - definite and indefinite
define Base [ [0 ◁ <(R,2,l)>]|[0 ◁ <(R,1,indef)>] ]
            [ [k i t a a b]|[q a m a r] ];
! Bases beginning with $ - definite and indefinite
define $Base [ [0 ◁ <(R,2,$)>]|[0 ◁ <(R,1,indef)>] ]
             [$ a m s];
! Bases beginning with d - definite and indefinite
define dBase [ [0 ◁ <(R,2,d)>]|[0 ◁ <(R,1,indef)>] ]
             [d a f t a r];
! Read definite and indefinite suffixes.
define Suffix [<(R,1,def)> ▷ u]|[<(R,1,indef)> ▷ un];
! The complete network.
define ArabicExample Prefix [Base | $Base | dBase]
                     Suffix;
```

The variable Prefix denotes the arcs connecting the first two states of the FSRA, in which the definite article (if present) is scanned and information indicating whether the word is definite or not is saved into register 1. In addition, if the word is definite then register 2 stores the actual form of the definite article. The lexicon is divided into several parts: the Base variable denotes nouns that do not trigger assimilation. Other variables ($Base, dBase) denote nouns that trigger assimilation, where for each assimilitaion case, a different lexicon is constructed. Each part of the lexicon deals with both its definite and indefinite nouns by allowing traversing the arcs only if the register content is appropriate. The variable Suffix denotes the correct suffix, depending on whether the noun is definite or indefinite. This is possible using the information that was stored in register 1 by the variable Prefix.

3 Dedicated regular expressions for linguistic applications

3.1 Circumfixes

The usefulness of FSRAs for non-concatenative morphology is demonstrated by [9], who show a specific FSRA accounting for circumfixation in Hebrew. We introduce a dedicated regular expression operator for circumfixation and show how expressions using this operator are compiled into the appropriate FSRA. The operator accepts a regular expression, denoting a set of bases, and a set of circumfixes, each of which containing a prefix and a suffix regular expressions. It yields as a result an FSRA obtained by prefixing and suffixing the base with each of the circumfixes. The main purpose of this operator is to deal with

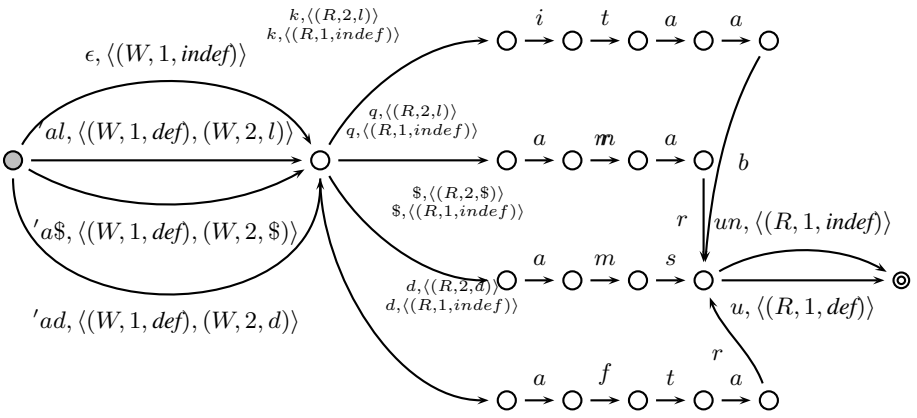

Fig. 1. FSRA-2 for Arabic nominative definite and indefinite nouns

cases in which the circumfixes are pairs of strings, but it is defined such that the circumfixes can be arbitrary regular expressions.

Definition 2. *Let Σ be a finite set such that $\Box, \{, \}, \langle, \rangle, \otimes \notin \Sigma$. We define the $\otimes$ operation to be of the form*

$$R \otimes \{\langle\beta_1 \Box \gamma_1\rangle\langle\beta_2 \Box \gamma_2\rangle \ldots \langle\beta_m \Box \gamma_m\rangle\}$$

where: $m \in \mathbb{N}$ is the number of circumfixes; R is a regular expression over Σ denoting the set of bases and β_i, γ_i for $1 \leq i \leq m$ are regular expressions over Σ denoting the prefix and suffix of the i-th circumfix, respectively.

Notice that R, β_i, γ_i may denote infinite sets. To define the denotation of this operator, let β_i, γ_i be regular expressions denoting the FSRAs A_i^β, A_i^γ, respectively. The operator yields an FSRA constructed by concatenating three FSRAs. The first is the FSRA constructed from the union of the FSRAs $A'^\beta_1, \ldots, A'^\beta_m$, where each A'^β_i is an FSRA obtained from A_i^β by adding a new node, q, which becomes the initial node of A'^β_i, and an arc from q to the initial node of A_i^β; this arc is labeled by ϵ and associated with $\langle(W, 1, \beta_i \Box \gamma_i)\rangle$. In addition, the register operations of the FSRA A_i^β are shifted by one register in order not to cause undesired effects by the use of register 1. The second FSRA is the FSRA denoted by the regular expression R (again, with one register shift) and the third is constructed in the same way as the first one, with the difference that the FSRAs are those denoted by $\gamma_1, \ldots, \gamma_m$ and the associated register operation is $\langle(R, 1, \beta_i \Box \gamma_i)\rangle$. Notice that the concatenation operation, defined by [9], adjusts

the register operations in the FSRAs to be concatenated, to avoid undesired effects caused by using joint registers. We use this operation to concatenate the three FSRAs, leaving register 1 unaffected (to handle the circumfix).

Example 3 *Consider the participle-forming combinations in German, e.g., the circumfix ge-t. A simplified account of the phenomenon is that German verbs in their present form take an 'n' suffix but in participle form they take the circumfix ge-t. The following examples are from [10]:*

säuseln 'rustle' gesäuselt 'rustled'
brüsten 'brag' gebrüstet 'bragged'

The FSRA of Figure2, which accepts the four forms, is yielded by the regular expression

$$[s\ \ddot{a}\ u\ s\ e\ l\ \mid\ b\ r\ \ddot{u}\ s\ t\ e] \otimes \{\langle \epsilon \Box n \rangle \langle g\ e \Box t \rangle\}$$

This regular expression can be easily extended to accept more German verbs in other forms. More circumfixation phenomena in other languages such as Indonesian, Arabic etc. can be modeled in the same way using this operator.

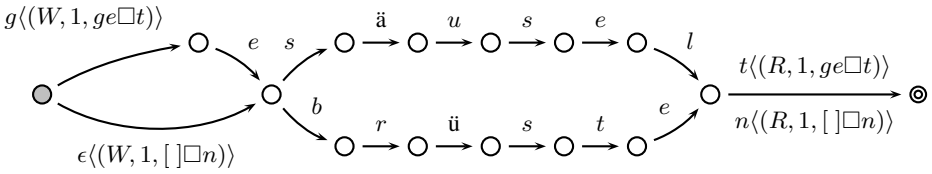

Fig. 2. Participle-forming combinations in German

3.2 Interdigitation

For interdigitation, [9] introduce a dedicated regular expression operator, *splice*, which accepts a set of strings of length n over Σ^*, representing a set of roots, and a list of patterns, each containing exactly n 'slots', and yields a set containing all the strings created by splicing the roots into the slots in the patterns. Formally, if Σ is such that $\Box, \{, \}, \langle, \rangle, \oplus \notin \Sigma$, then the *splice* operation is of the form

$$\{\langle \alpha_{1\ 1}, \alpha_{1\ 2}, ..., \alpha_{1\ n} \rangle, ..., \langle \alpha_{m\ 1}, \alpha_{m\ 2}, ..., \alpha_{m\ n} \rangle\}$$

$$\oplus$$

$$\{\langle \beta_{1\ 1} \Box \beta_{1\ 2} \Box ... \beta_{1\ n} \Box \beta_{1\ n+1} \rangle, ..., \langle \beta_{k\ 1} \Box \beta_{k\ 2} \Box ... \beta_{k\ n} \Box \beta_{k\ n+1} \rangle\}$$

where $n \in \mathbb{N}$ is the number of slots (represented by '$\square$'); $m \in \mathbb{N}$ is the number of roots; $k \in \mathbb{N}$ is the number of patterns and $\alpha_{ij}, \beta_{ij} \in \Sigma^*$. This operator suffers from lack of generality as the set of roots and patterns must be strings; we generalize the operator in a way that supports *any* regular expression denoting a language for both the roots and the patterns. This extension is done by simply allowing $\alpha_{ij}, \beta ij$ to be arbitrary regular expressions (including regular expressions denoting FSRAs). The construction of the FSRA denoted by this generalized operation is done in the same way as in the case of circumfixes with two main adjustments. The first is that in this case the final FSRA is constructed by concatenating $2n + 1$ intermediate FSRAs (n FSRAs for the n parts of the roots and $n + 1$ FSRAs for the $n + 1$ parts of the patterns). The second is that here, 2 registers are used to remember both the root and the pattern. We suppress the detailed description of the construction. The circumfixation operator may seem redundant, being a special case of interdigitation. However, it results in a more compact network without any unnecessary register operations.

4 Finite state registered transducers

We extend the FSRA model to *finite-state registered transducers* (FSRT), denoting relations over two finite alphabets. The extension is done by adding to each transition an output symbol. This facilitates an elegant solution to the problem of binary incrementors which was introduced in section 1.

Example 4 *Consider again the 32-bit incrementor example mentioned in section 1. Recall that a sequential transducer for an n-bit binary incrementor would require 2^n states and a similar number of transitions. Using the FSRT model, a more efficient n-bit transducer can be constructed. A 4-bit FSRT incrementor is shown in Figure 3. The first four transitions copy the input string into the registers, then the input is scanned (using the registers) from right to left (as the carry moves), calculating the result, and the last four transitions output the result (in case the input is 1^n, an extra 1 is added in the beginning). Notice that this transducer guarantees linear recognition time, since from each state only one arc can be traversed in each step, even when there are ϵ-arcs. In the same way, an n-bit transducer can be constructed for all $n \in \mathbb{N}$. Such a transducer will have n registers, $3n + 1$ states and $6n$ arcs. The FSRT model solves the incrementor problem in much the same way it is solved by vectorized finite state automata, but the FSRT solution is more intuitive and is based on existing finite state techniques.*

It is easy to show that FSRTs, just like FSRAs, are equivalent to their non-registered counterparts. It immediately implies that FSRTs maintain the closure

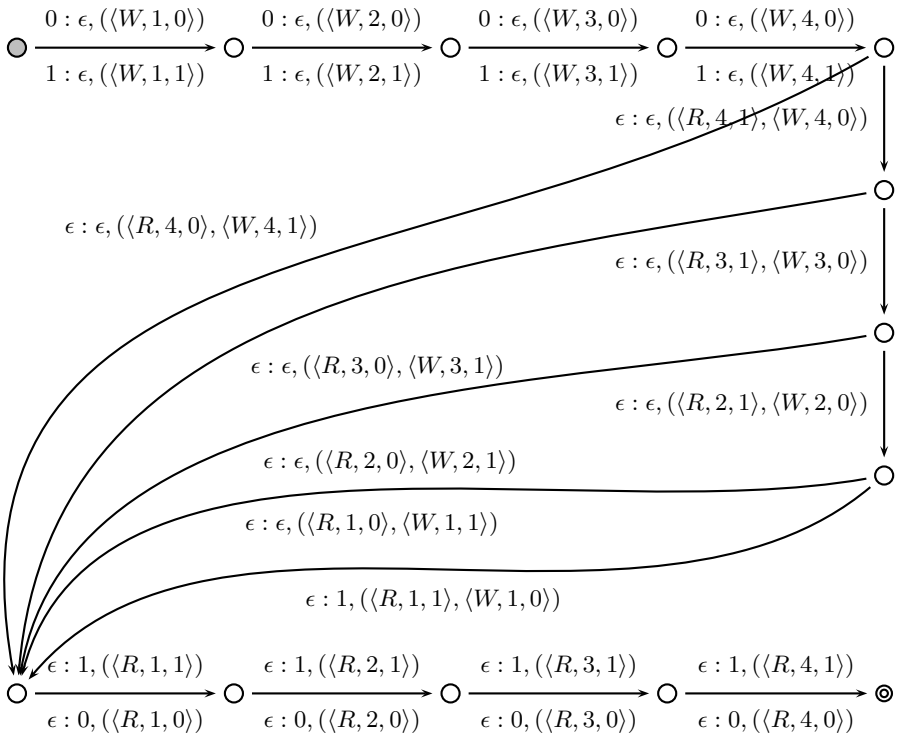

Fig. 3. 4-bit incrementor using FSRT

properties of regular relations. Thus, performing the regular operations on FSRTs can be easily done by converting them first into finite state transducers. However, such a conversion may result in an exponential increase in the size of the network, invalidating the advantages of FSRTs. Therefore, as in FSRAs, implementing the closure properties directly on FSRTs is essential for benefiting from their space efficiency. Implementing the common operators such as union, concatenation etc. is done in the same ways as in FSRAs ([9]). Direct implementation on FSRTs of composition is a naïve extension of ordinary transducers composition, based on the intersection construction of FSRAs ([9]). We explicitly define these operations in [12].

5 Implementation and evaluation

In order to practically compare the space and time performance of FSRAs and FSAs, we have implemented the special operators introduced in section sec: regular expression for nl for circumfixation and interdigitation, as well as direct construction of FSRAs. We have compared FSRAs with ordinary FSAs by

building corresponding networks for circumfixation, interdigitation and n-bit incrementation. For circumfixation, we constructed networks for the circumfixation of 1043 Hebrew roots and 4 circumfixes. For interdigitation we constructed a network accepting the splicing of 1043 roots into 20 patterns. For n-bit incrementation we constructed networks for 10-bit, 50-bit and 100-bit incrementors. Figure 4 displays the size of each of the networks in terms of states, arcs and actual file size.

Clearly, FSRAs provide a significant reduction in the network size. In particular, we could not construct an n-bit incrementor FSA for any n greater than 100 as a result of memory problems, whereas using FSRAs we had no problem constructing networks even for $n = 50,000$.

In addition, we compared the recognition times of the two models. For that purpose, we used the circumfixation, interdigitation, 10-bit incrementation and 50-bit incrementation networks to analyze 200, 1000 and 5,000 words. As can be seen in Figure 5, time performance is comparable for the two models, except for interdigitation, where FSAs outperform FSRAs by a constant factor. The reason is that in this network the usage of registers is massive and thereby, there is a higher cost to the reduction of the network size, in terms of analysis time. This is an instance of the common tradeoff of time versus space: FSRAs improve the network size at the cost of slower analysis time in some cases. When using finite state devices for natural language processing, often the generated networks become too large to be practical. In such cases, using FSRAs can make network size manageable. Using the closure constructions one can build desired networks of reasonable size, and at the end decide whether to convert them to ordinary FSAs, if time performance is an issue.

Operation	Network type	States	Arcs	Registers	File size
Circumfixation	FSA	811	3824	–	47kb
(4 circumfixes, 1043 roots)	FSRA	356	360	1	16kb
Interdigitation	FSA	12,527	31,077	–	451kb
(20 patterns, 1043 roots)	FSRA	58	3259	2	67kb
10-bit incrementor	Sequential FST	268	322	–	7kb
	FSRT	31	60	10	2kb
50-bit incrementor	Sequential FST	23,328	24,602	–	600kb
	FSRT	151	300	50	8kb
100-bit incrementor	Sequential FST	176,653	181,702	–	4.73Mb
	FSRT	301	600	100	17kb

Fig. 4. Space comparison between FSAs and FSRAs

		200 words	1000 words	5000 words
Circumfixation	FSA	0.01s	0.02s	0.08s
(4 circumfixes, 1043 roots)	FSRA	0.01s	0.02s	0.09s
Interdigitation	FSA	0.01s	0.02s	1s
(20 patterns, 1043 roots)	FSRA	0.35s	1.42s	10.11s
10-bit incrementor	Sequential FST	0.01s	0.05s	0.17s
	FSRT	0.01s	0.06s	0.23s
50-bit incrementor	Sequential FST	0.13s	0.2s	0.59s
	FSRT	0.08s	0.4s	1.6s

Fig. 5. Time comparison between FSAs and FSRAs

6 Conclusions

We have shown how FSRAs can be used to model non-trivial morphological processes in natural languages, including vowel-harmony, circumfixation and interdigitation. We also provided a regular expression language to denote arbitrary FSRAs. In addition, we extended FSRAs to transducers and demonstrated their efficiency. Moreover, we evaluated FSRAs through an actual implementation.

While our approach is similar in spirit to Flag Diacritics ([6]), we provide a complete and accurate description of the FSRAs constructed from our extended regular expressions. The transparency of the construction details allows further insight into the computational efficiency of the model and provides an evidence to its regularity. Moreover, the presentation of dedicated regular expression operations for non-concatenative processes allows easier construction of complex registered networks, especially for more complicated processes such as interdigitation and circumfixes.

In section 5 we discuss an implementation of FSRAs. Although we have used this system to construct networks for several phenomena, we are interested in constructing a network for describing the complete morphology of a natural language containing many non-concatenative phenomena, e.g., Hebrew. A morphological analyzer for Hebrew, based on finite state calculi, already exists [13], but is very space-inefficient and, therefore, hard to maintain. It would be beneficial to compact such a network using FSRTs, and to inspect the time versus space tradeoff on such a comprehensive network.

Acknowledgments

This research was supported by The Israel Science Foundation (grant number 136/01). We are grateful to Dale Gerdemann for his help.

References

1. Koskenniemi, K.: Two-Level Morphology: a General Computational Model for Word-Form Recognition and Production. The Department of General Linguistics, University of Helsinki (1983)
2. Kaplan, R.M., Kay, M.: Regular models of phonological rule systems. Computational Linguistics **20** (1994) 331–378
3. Karttunen, L., Chanod, J.P., Grefenstette, G., Schiller, A.: Regular expressions for language engineering. Natural Language Engineering **2** (1996) 305–328
4. Mohri, M.: On some applications of finite-state automata theory to natural language processing. Natural Language Engineering **2** (1996) 61–80
5. van Noord, G., Gerdemann, D.: An extendible regular expression compiler for finite-state approaches in natural language processing. In Boldt, O., Jürgensen, H., eds.: Automata Implementation. Number 2214 in Lecture Notes in Computer Science. Springer (2001)
6. Beesley, K.R.: Constraining separated morphotactic dependencies in finite-state grammars. In: FSMNLP-98., Bilkent, Turkey (1998) 118–127
7. Beesley, K.R., Karttunen, L.: Finite-State Morphology: Xerox Tools and Techniques. Cambridge University Press (Forthcoming)
8. Kornai, A.: Vectorized finite state automata. In: Proceedings of the workshop on extended finite state models of languages in the 12th European Conference on Artificial Intelligence, Budapest (1996) 36–41
9. Cohen-Sygal, Y., Gerdemann, D., Wintner, S.: Computational implementation of non-concatenative morphology. In: Proceedings of the Workshop on Finite-State Methods in Natural Language Processing, an EACL'03 Workshop. (2003) 59–66
10. Sproat, R.W.: Morphology and Computation. MIT Press, Cambridge, MA (1992)
11. Nash, D.: Topics in Warlpiri Grammar. PhD thesis, Massachusetts Institue of Technlogy (1980)
12. Cohen-Sygal, Y.: Computational implementation of non-concatenative morphology. Master's thesis, University of Haifa (2004)
13. Yona, S., Wintner, S.: A finite-state morphological grammar of hebrew. In: Proceedings of the ACL-2005 Workshop on Computational Approaches to Semitic Languages. (2005)