Computational Implementation of Non-Concatenative Morphology

Yael Cohen-Sygal
Dept. of Computer Science
University of Haifa
yaelc@cs.haifa.ac.il

Dale Gerdemann
Seminar für Sprachwissenschaft
Universität Tübingen
dg@sfs.uni-tuebingen.de

Shuly Wintner
Dept. of Computer Science
University of Haifa
shuly@cs.haifa.ac.il

Abstract

We introduce *finite state registered automata*, a new computational model within the framework of finite state (FS) technology that accounts for non-concatenative morphological processes such as word formation in Semitic languages. It extends existing FS techniques, which are presently sub-optimal for describing such phenomena. We define the new model, prove its mathematical properties and exemplify its usability by describing some non-concatenative phenomena.

1 Introduction

While much of the inflectional morphology of Semitic languages can be rather straightforwardly described using concatenation as the primary operation, the main word formation process in such languages is inherently non-concatenative. The standard account describes words in Semitic languages as combinations of two morphemes: a root and a pattern.[1] The root consists of consonants only, by default three (although longer roots are known). The pattern is a combination of vowels and, possibly, consonants too, with 'slots' into which the root consonants can be inserted. Words are created by *interdigitating* roots into patterns: the first consonant of the root is inserted into the first consonantal slot of the pattern, the second root consonant fills the second slot and the third fills the last slot. As an example, consider the Hebrew roots g.d.l, k.t.b and r.$.m and the patterns haCCaCa, hitCaCCut and miCCaC, where the 'C's indicate the slots. When the roots combine with these patterns the resulting lexemes are *hagdala, hitgadlut, migdal, haktaba, hitkatbut, miktab, har$ama, hitra$mut, mir$am*, respectively. After the root combines with the pattern, some morpho-phonological alternations take place, which may be non-trivial but are mostly concatenative.

[1]An additional morpheme, *vocalization*, is used to abstract the pattern further; for the present purposes, this distinction is irrelevant.

Another non-concatenative process is *reduplication*: the process in which a morpheme or part of it is duplicated. Full reduplication is used as a pluralization process in Malay and Indonesian; partial reduplication is found in Chamorro to indicate intensivity. It can also be found in Hebrew as a diminutive formation of nouns and adjectives:

keleb	*klablab*	*$apan*	*$panpan*	*zaqan*	*zqanqan*
dog	puppy	rabbit	bunny	beard	goatee

$axor	*$xarxar*	*qatan*	*qtantan*
black	dark	little	tiny

Finite-state (FS) technology is considered adequate for describing the morphological processes of the world's languages since the pioneering works of Koskenniemi (1983) and Kaplan and Kay (1994). Several toolboxes provide extended regular expression description languages and compilers of the expressions to finite state automata (FSAs) and transducers (FSTs) (Karttunen et al., 1996; Mohri, 1996; van Noord and Gerdemann, 2001). While FS approaches for NLP 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 the non-concatenative word formation processes that Semitic languages exhibit (Lavie et al., 1988). The major problem that we tackle in this work is medium-distance dependencies, whereby some elements that are related to each other in some deep-level representation (e.g., the consonants of the root) are separated on the surface. These phenomena do not lie outside the descriptive power of FS systems: a naïve implementation will construct an FSA with an accepting path for each word. However, such an implementation can result in huge networks that are inefficient to process, as the following examples show.

Example 1. *Consider the three Hebrew patterns discussed above. In Hebrew, the patterns are written hCCCh, htCCCut and mCCC, respectively,*[2] *i.e., the consonants are inserted into the C slots as one unit. For simplicity we use this representation here. An FSA accepting all the possible combinations of roots and these three patterns is illustrated in Figure 1.*[3] *In the general case, for r roots and p patterns, the number of its states is $(r \times 2 + 2) \times p + 2$ or $O(r \times p)$ (in some cases where an affix appears in several patterns or roots the number will be somewhat smaller). Evidently, the three basic different paths that result from the three patterns have the same 'body', accounting for the roots. Attempting to avoid the duplication of paths, the FSA in Figure 2 accepts the language that is denoted by the regular expression $(ht + h + m)(roots)(ut + h + \epsilon)$. The number of states here is $2 \times r + 4$, i.e., independent of the number of patterns (and the same holds for the number of arcs). The problem of such an automaton is that it also accepts invalid words such as the pattern mCCCut. In other words, it ignores the dependencies which hold between prefixes and suffixes of the same pattern. Since FS devices have no memory, save for the states, there is no simple way to account for these dependencies.*

[2]Many of the vowels are not explicitly depicted in the Hebrew script.

[3]This is an over-simplified example; in practice, the process of combining roots with patterns is highly idiosyncratic, like other derivational morphological processes.

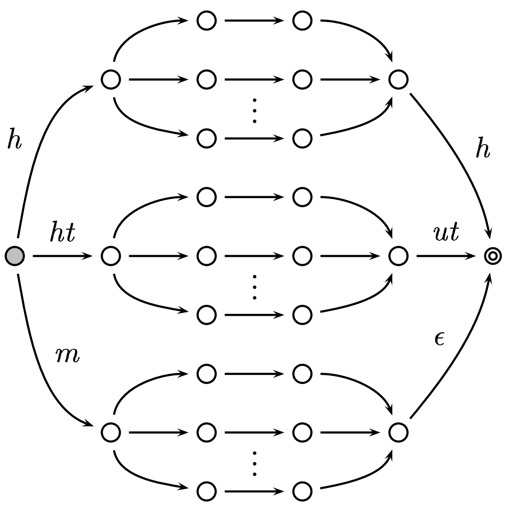

Figure 1: Naïve FSA with duplicated paths

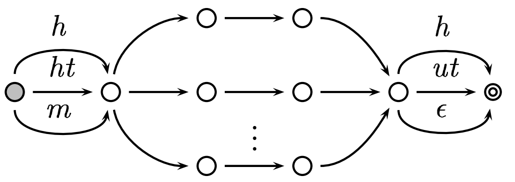

Figure 2: Over-generating FSA

Similar problems are caused by reduplication.

Example 2. *Let Σ be a finite alphabet. The language $L = \{ww \mid w \in \Sigma^*\}$ is known to be trans-regular; however, $L_n = \{ww \mid w \in \Sigma^*$ and $|w| = n\}$ for some constant n is regular. Recognizing L_n is a finite approximation of the general problem of recognizing L. As the amount of reduplication in natural languages is in practice limited, the descriptive power of L_n is sufficient for describing such phenomena (by constructing L_n for small number of different n's). An FSA for L_n can be constructed by listing a path for each accepted string: since Σ and n are finite, the number of words in L_n is finite. The main drawback of such an automaton is the growth in its number of states and arcs as $|\Sigma|$ and n increase: the number of strings in L_n is $|\Sigma|^n$. Thus, FS techniques can describe reduplication, but they do so inefficiently regarding their space complexity.*

In this work we create a novel FS model which facilitates the expression of medium-distance dependencies such as interdigitation and reduplication in an efficient way. Our main motivation is theoretical improvements, i.e., reducing the complexity of the number of states and arcs in the networks; we believe that these will result in practical improvements. We define the model formally, show that it is equivalent to FSAs and define many closure properties directly. For lack of space, some of the proofs and examples are suppressed here (see Cohen-Sygal (Forthcoming) for more details). We also provide a dedicated regular expression operator for interdigitation. We exemplify the usefulness of the model by efficiently accounting for the motivating examples (as this is still work in progress, some examples are not fully worked-out yet).

2 Related work

In spite of the common view that FS technology is in general inadequate for describing non-concatenative processes, several works deal with the above-mentioned problems in various ways.

Kiraz (2000) expands the traditional two-level model of Koskenniemi (1983) into n-tape automata, following the insight of Kay (1987) and Kataja and Koskenniemi (1988). The idea is to use more than two levels of expression: the surface level employs one representation, but the lexical form employs multiple representations (e.g., root, pattern, etc.) and therefore can be divided into different levels, one for each representation. Elements that are separated on the surface (such as the root's consonants) are adjacent on a particular lexical level. Kiraz (2000) does not discuss the space complexity of the resulting machine compared to the naïve one, but it seems that the number of states still increases with the number of roots and patterns. Moreover, the n-tape model requires specification of dependencies between symbols in different levels, which may be non-trivial.

Beesley (1998) suggests a way to constrain dependencies between separated morphemes. The method, called *flag diacritics*, adds 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 to networks, thus keeping the total size of the network relatively small. While our proposal is similar in spirit, we provide a complete mathematical and computational analysis of such extended networks, including a construction of the main closure properties and compilation of dedicated regular expression operators, whereas Beesley (1998) only provides usage examples. We also prove that our model is regular.

Beesley and Karttunen (2000) describe a technique, *compile-replace*, for constructing FSTs, which involves reapplying the regular-expression compiler to its own output. The compile-replace algorithm facilitates a compact definition of non-concatenative morphological processes. However, since such expressions compile to the naïve networks, no space is saved. Furthermore, this is a compile-time mechanism rather than a theoretical and mathematically founded solution.

Other works (Blank, 1985; Blank, 1989; Kornai, 1999) aim to extend the FS model by adding a limited amount of memory. As their main objective is not to account for medium-distance dependencies, they are not of immediate relevance here.

3 Finite state registered automata

We define a novel model which facilitates the expression of medium-distance dependencies. It is reminiscent of Kaminski and Francez (1994)[4] in the sense that it augments FSAs 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 FSA to 'remember' a finite number of symbols. In addition to being associated with an alphabet symbol, each arc is also associated with an action on the registers: *read* (R), which allows traversing an arc only if a designated register contains a specific symbol; and *write* (W), which writes a specific symbol into a designated register while traversing an arc.

3.1 Definitions and examples

Definition 1. *A finite state registered automaton (FSRA) is a tuple* $A = \langle Q, q_0, \Sigma, \Gamma, n, \delta, F\rangle$ *where*

[4]Kaminski and Francez (1994) define automata over infinite alphabets and use registers to store alphabet symbols.

Q is a finite set of states; $q_0 \in Q$ is the initial state; Σ is the finite language alphabet; Γ is a finite (registers) alphabet not including '$\#$'; $n \in \mathbb{N} \cup \{0\}$ is the number of registers; $\delta \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times \{R, W\} \times \{0, 1, \ldots, n\} \times (\Gamma \cup \{\#\}) \times Q$ is the transition relation and $F \subseteq Q$ is the set of final states. The initial content of the registers is $\#^{n+1}$, i.e., the initial value of all the registers is 'empty'.

The intuitive meaning of δ is as follows: when $i > 0$, $(s, \sigma, op, i, \gamma, t) \in \delta$ means that A can move from state s to state t upon reading the input symbol σ. If $op = R$, the move requires that the contents of register i be γ; if $op = W$, the move writes γ into register i. If $i = 0$, op and γ are ignored; we use the shorthand notation (s, σ, t) for such transitions. Note that FSRAs are non-deterministic.

A *configuration* of A is a pair (q, u), where $q \in Q$ and $u \in (\Gamma \cup \{\#\})^{n+1}$ (q is the current state and u represents the registers content). The set of all configurations of A is denoted by Q^c. The pair $q_0^c = (q_0, \#^{n+1})$ is called the *initial configuration*, and configurations with the first component in F are called *final configurations*. The set of final configurations is denoted by F^c.

Let $u = u_0 u_1 \ldots u_n$ and $v = v_0 v_1 \ldots v_n$. Given a symbol $\alpha \in \Sigma \cup \{\epsilon\}$ and an FSRA A, we say that a configuration (s, u) *entails* a configuration (t, v), denoted $(s, u) \vdash_{\alpha, A} (t, v)$, iff the automaton can move from (s, u) to (t, v) when scanning the input α (or without any input, when $\alpha = \epsilon$) in one step. That is, either $(s, \alpha, R, i, \gamma, t) \in \delta$ for some $0 \leq i \leq n$ and $\gamma \in \Gamma$ and $u = v$ and $u_i = v_i = \gamma$; or $(s, \alpha, W, i, \gamma, t) \in \delta$ for some $0 \leq i \leq n$ and $\gamma \in \Gamma$ and for all $k \in \{0, 1, ..., n\}$ such that $k \neq i$, $u_k = v_k$ and $v_i = \gamma$.

A *run* of A on w is a sequence of configurations $c_0, ..., c_r$ such that $c_0 = q_0^c$, $c_r \in F^c$, for every k, $1 \leq k \leq r$, $c_{k-1} \vdash_{\alpha_k, A} c_k$ and $w = \alpha_1 ... \alpha_r$. Notice that $|w|$ might be less than r since some of the α_i might be ϵ. An FSRA A *accepts* a word w if there exists a run of A on w.

Example 3. *Consider again example 1. An efficient FSRA accepting all and only the possible combinations of r roots and $p = 3$ patterns is shown in Figure 3. The register alphabet is $\{p_1, p_2, p_3\}$, where p_i represents the i-th pattern. The number of states is $2r + 4$ (just like the FSA of Figure 2), that is, $O(r)$, and in particular independent of p. The number of arcs is also reduced from $O(r \times p)$ to $O(r + p)$.*

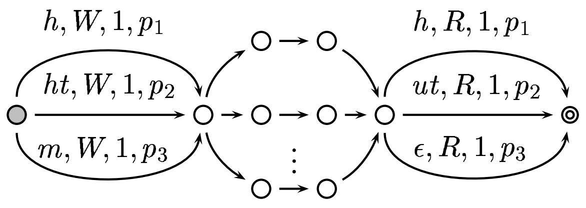

Figure 3: FSRA

Example 4. *Consider now a representation of Hebrew where all vowels are explicit, e.g., the pattern hitCaCeC. Consider also the roots g.d.l, k.t.b, r.$.m. A minimal FSA for the combinations of these roots and this pattern is given in Figure 4; it has fifteen states. If the number of roots is r, a general FSA accepting the combinations of roots with this pattern will have $4r + 3$ states and $5r + 1$ arcs. The FSRA of Figure 5 also accepts the same language (the register alphabet indicates the roots); it has 7 states and will have 7 states for any number of roots. The number of arcs is reduced to $3r + 3$.*

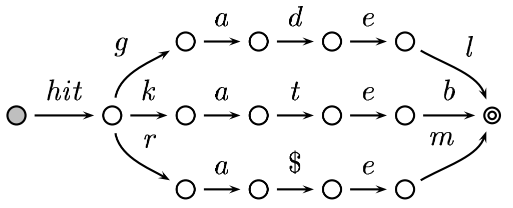

Figure 4: FSA for the pattern hitCaCeC

3.2 Equivalence to regular languages

FSRAs and FSAs recognize the same class of languages. Clearly, every FSA has an equivalent FSRA: given an FSA A, add to A one register to construct an equivalent FSRA A'. Since every transition (s, σ, t) in an FSRA is a shorthand notation for $(s, \sigma, R, 0, \#, t)$, every transition in A is also a transition in A'. Trivially $L(A') = L(A)$.

For the reverse direction, construct an FSA equivalent to a given FSRA. In FSRAs n, Γ and Q are finite, hence the number of configurations is finite. The FSA's states are the configurations of the FSRA and the transition function simulates the entailment relation. Notice that entailment between configurations in an FSRA is dependent on Σ only, similarly to the transition function in an

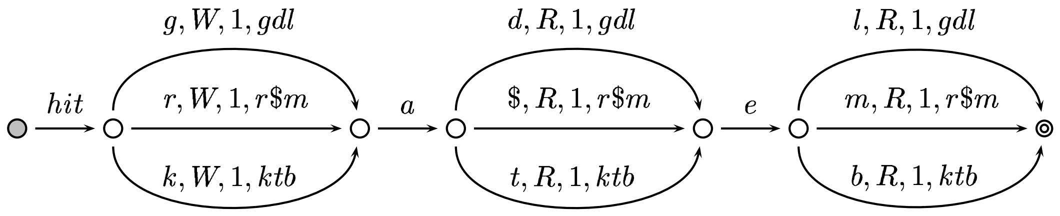

Figure 5: FSRA for the pattern hitCaCeC

FSA. The constructed FSA is non-deterministic, with possible ϵ-moves.

Proposition 1. *FSRAs are equivalent to FSAs.*

Proof. Let $A = \langle Q, q_0, \Sigma, \Gamma, n, \delta, F\rangle$ be an FSRA. Construct a finite state automaton $A' = \langle Q', q'_0, \Sigma, \delta', F'\rangle$ where $Q' = Q^c$, $q'_0 = q^c_0$, $F' = F^c$ and $\delta' = \{(s', \alpha, t') \mid \alpha \in (\Sigma \cup \{\epsilon\}),\ s', t' \in Q'$, and $s' \vdash_{\alpha,A} t'\}$. The proof that $L(A) = L(A')$ is suppressed for lack of space. □

Notice that the number of configurations in A is $|Q| \times (|\Gamma| + 1)^{n+1}$, therefore the growth in the number of states when constructing A' from A might be in the worst case exponential in the number of registers. In other words, the move from FSAs to FSRAs can yield an exponential reduction in the size of the network.

3.3 Multiple register actions

The FSRA model defined above allows only one register operation on each transition. We extend it 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. FSRA-k allows further reduction of the net size for some automata as well as other advantages that are discussed below. Let $Actions_n = \{R, W\} \times \{0, 1, 2, \ldots, n\} \times (\Gamma \cup \{\#\})$, and a_i be meta-variables over elements of $Actions_n$.

Definition 2. *An order-k FSRA (FSRA-k) is a tuple $A = \langle Q, q_0, \Sigma, \Gamma, n, k, \delta, F\rangle$ where $Q, q_0, \Sigma, \Gamma, n, F$ and the initial contents of the registers are as before, $k \in \mathbb{N}$ is the maximum number of register operations allowed on each arc, and*

$$\delta \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times \bigcup_{j=1}^{k} \{\langle a_1, ..., a_j\rangle\} \times Q.$$

δ is extended to allow each transition to be associated with a series of up to k operations on the registers. Each operation has the same meaning as before. The register operations are done in the order in which they are specified. Thus, $(s, \sigma, \langle a_1, ..., a_i\rangle, t) \in \delta$ where $i \le k$ implies that if A is in state s, the input symbol is σ and all the register operations $a_1, ..., a_i$ are executed successfully, then A may enter state t.

Proposition 2. *FSRA-k and FSRAs recognize the same class of languages.*

Proof. Every FSRA is an FSRA-k for $k = 1$. For the reverse direction, construct an equivalent FSRA A' given an FSRA-k A. Each transition in A is replaced by a series of transitions in A', each of which performs one operation on the registers. The first transition in the series deals with the input symbol and the rest are ϵ-transitions. This construction requires additional states to enable the addition of transitions. Each transition in A that is replaced adds as many states as the number of register operations performed on this transition minus one. The formal construction and equivalence proof are omitted for lack of space. □

FSRA-k is a very space-efficient finite state device. The next proposition shows how ordinary finite state automata can be encoded efficiently by FSRA-2. Given an FSA A, an equivalent FSRA-2 A' is constructed. A' has three states, $\{q'_0, q_f, q_{nf}\}$, and one register (formally, it has 2 registers but register 0 is never used). State q_f functions as a representative for the final states in A, q_{nf} – as a representative for the non-final ones and q'_0 is the initial state. The registers alphabet consists of the states of A. Each arc in A has an equivalent arc in A' with two register operations. The first reads the current state of A from the register and the second writes the new state into the register. If the source state of a transition in A is a final state then the source state of the corresponding transition in A' is q_f; if the source state of a transition in A is a non-final state then the source state of the corresponding transition in A' is q_{nf}. The same holds also for the target states. The purpose of the initial state is to write the initial state of A into the register . In this way A' simulates the behavior of A. Notice that the number of arcs in A' equals the number of arcs in A plus one.

Proposition 3. *Every FSA has an equivalent FSRA-2 with three states and one register.*

Proof. Let $A = \langle Q, q_0, \Sigma, \delta, F\rangle$ be an FSA and $f : Q \rightarrow \{q_f, q_{nf}\}$ be a total function defined by $f(q) = q_f$ if $q \in F$, $f(q) = q_{nf}$ otherwise. Construct an FSRA-2 $A' = \langle Q', q'_0, \Sigma', \Gamma', 1, 2, \delta', F'\rangle$ where $Q' = \{q'_0, q_{nf}, q_f\}$, $\Sigma' = \Sigma$, $\Gamma = Q$, $F' = \{q_f\}$ and $\delta = \{(f(s), \sigma, \langle R, 1, s\rangle, \langle W, 1, t\rangle, f(t)) \mid (s, \sigma, t) \in \delta\} \cup \{(q'_0, \epsilon, \langle W, 1, q_0\rangle, f(q_0))\}$. The equivalence proof is suppressed. □

3.4 Closure properties

The equivalence of FSAs and FSRAs immediately implies that FSRAs maintain the closure properties of regular languages. Thus, performing the regular operations on FSRAs can be easily done by converting them first into FSAs. However, as shown above, such a conversion may result in an exponential increase in the size of the automaton, invalidating the advantages of this new model. Therefore, we show how these operations can be done directly on FSRAs. The constructions are mostly based on the standard ones with some essential modifications. Let $A_1 = \langle Q_1, q^1_0, \Sigma_1, \Gamma_1, n_1, \delta_1, F_1\rangle$ and $A_2 = \langle Q_2, q^2_0, \Sigma_2, \Gamma_2, n_2, \delta_2, F_2\rangle$ be FSRAs.

To recognize $L(A_1) \cup L(A_2)$, construct an FSRA $A = \langle Q, q_0, \Sigma, \Gamma, n, \delta, F\rangle$ where $q_0 \notin Q_1 \cup Q_2$ and $Q = \{q_0\} \cup Q_1 \cup Q_2$; $\Sigma = \Sigma_1 \cup \Sigma_2$; $\Gamma = \Gamma_1 \cup \Gamma_2$; $n = max\{n_1, n_2\}$; $F = F_1 \cup F_2$; and $\delta = \delta_1 \cup \delta_2 \cup \{(q_0, \epsilon, q^1_0), (q_0, \epsilon, q^2_0)\}$. Notice that in any specific run of A, the computation goes through just one of the original automata, so the set of registers can be used for strings of $L(A_1)$ or $L(A_2)$, as needed.

For concatenation, construct an FSRA $A = \langle Q, q_0, \Sigma, \Gamma, n, \delta, F\rangle$ to recognize $L(A_1) \cdot L(A_2)$ where: $Q = Q_1 \cup Q_2$; $q_0 = q^1_0$; $\Sigma = \Sigma_1 \cup \Sigma_2$; $\Gamma = \Gamma_1 \cup \Gamma_2$; $n = n_1 + n_2$; $F = F_2$; and $\delta = \delta_1 \cup \{(f, \epsilon, q^2_0) \mid f \in F_1\} \cup \{(s, \sigma, op, i + n_1, \gamma, t) \mid (s, \sigma, op, i, \gamma, t) \in \delta_2\}$. The first n_1 registers (after register 0) are used for recognizing strings of $L(A_1)$, and the rest are used for $L(A_2)$.

For Kleene closure, add register operations to clear the contents of the registers between two subsequent traversals of the original automaton. Construct an FSRA-n $A = \langle Q, q_0, \Sigma, \Gamma, n, n, \delta, F\rangle$ to recognize $L(A_1)^*$, where $q_0 = q^1_0$; $\Sigma = \Sigma_1$; $\Gamma = \Gamma_1$; $n = n_1$; $F = F_1 \cup \{q_0\}$; and $\delta = \delta_1 \cup \{(f, \epsilon, \langle W, 1, \#\rangle, \ldots, \langle W, n, \#\rangle, q_0) \mid f \in F\}$. The intuitive meaning of δ is that δ_1 handles a substring from $L(A_1)$, then the added arcs delete the contents of the registers, leaving them ready to handle the next substring from $L(A_1)$.

For intersection, construct an FSRA-2 $A = \langle Q, q_0, \Sigma, \Gamma, n, 2, \delta, F\rangle$ to recognize $L(A_1) \cap L(A_2)$. The construction simulates the runs of A_1 and A_2 simultaneously. Each transition is associated with two register operations, one for each automaton. The number of the registers is the sum of the registers in the two automata. In A, the first n_1 registers after register 0 simulate the behavior of the registers of A_1 and the next n_2 registers simulate the behavior of the registers of A_2. In this way a word is accepted by the intersection automaton iff it is accepted by each one of the automata separately. Thus, $Q = Q_1 \times Q_2$; $q_0 = (q^1_0, q^2_0)$; $\Sigma = \Sigma_1 \cap \Sigma_2$; $\Gamma = \Gamma_1 \cup \Gamma_2$; $n = n_1 + n_2$; $F = F_1 \times F_2$; and $\delta = \{((s_1, s_2), \sigma, \langle op_1, i_1, \gamma_1\rangle, \langle op_2, i_2 + n_1, \gamma_2\rangle, (t_1, t_2)) \mid (s_1, \sigma, \langle op_1, i_1, \gamma_1\rangle, t_1) \in \delta_1$ and $(s_2, \sigma, \langle op_2, i_2, \gamma_2\rangle, t_2) \in \delta_2\}$.

Notice that each transition in δ is associated with exactly two register operations. Moreover, each register operation is associated with a different register, thus there cannot be two W operations on the same register. This guarantees that no information is lost during the simulation of the two intersected automata.

3.5 Regular expression expansion

We introduce a new regular expression operator, *splice*, for interdigitation, and show how expressions using it are compiled into FSRAs. The operator 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.

Definition 3. *Let Σ be such that $\star, \{, \}, \langle, \rangle, \oplus \notin \Sigma$. Define the splice operation to be of the form*

$$\{\langle \alpha_{1,1}, \ldots, \alpha_{1,n} \rangle, \ldots, \langle \alpha_{m,1}, \ldots, \alpha_{m,n} \rangle\} \oplus$$
$$\{\langle \beta_{1,1} \star \ldots \star \beta_{1,n+1} \rangle, \ldots, \langle \beta_{k,1} \star \ldots \star \beta_{k,n+1} \rangle\}$$

where $n \in \mathbb{N}$ is the number of slots (represented by '$\star$'); $m \in \mathbb{N}$ is the number of roots; $k \in \mathbb{N}$ is the number of patterns; $\alpha_{ij} \in \Sigma^*$ and $\beta_{ij} \in \Sigma^*$.

The first operand is a set of roots to be inserted into the slots of the second operand, a set of patterns. The slots are represented by the symbol '$\star$'. Such expressions are compiled into the FSRA of Figure 6, where β_i represents the i-th pattern and α_i – the i-th root. Only moves from q_0 to q_1 and from q_1 to q_2 write into the registers; other arcs are associated with R operations only. This FSRA has 3 registers where registers 1 and 2 'remember' the pattern and the root, respectively. It will have 3 registers and $2n+2$ states for any number of roots, patterns and slots.

Consider now the general case where α_{ij} and β_{ij} can be strings of letters. In this case, arcs of the FSRA defined above are simply replaced by paths: where an arc in the above definition is labeled by the letter α_{ij} it is replaced by a sequence of arcs labeled by the word α_{ij}, and similarly for β_{ij}.

Example 5. *Consider the Hebrew roots g.d.l, k.t.b, r.$.m and the patterns hitCaCeC, miCCaC and haCCaCa. The following expression yields the set of all lexemes obtained by splicing the roots into the patterns: $\{\langle g, d, l \rangle, \langle k, t, b \rangle, \langle r, \$, m \rangle\} \oplus \{\langle hit \star a \star e\star, \rangle \langle mi \star \star a\star, \rangle \langle ha \star \star a \star a \rangle\}$. We suppress the FSRA this expression compiles into.*

3.6 Reduplication

To account for reduplication we extend FSRAs to efficiently accept L_n of example 2. FSRAs have to be extended in order to be able to identify a pattern without actually distinguishing between different symbols in that pattern. The extended model, FSRA*, is created from FSRAs by introducing a new symbol, '*', assumed not to belong to Σ, and by forcing Γ to be equal to Σ. The '*' indicates equality between the input symbol and the designated register content, eliminating the need to duplicate paths for different symbols.

Definition 4. *Let $* \notin \Sigma$. An FSRA* is an FSRA where $\Sigma = \Gamma$ and the transition relation is extended to be $\delta \subseteq Q \times (\Sigma \cup \{\epsilon, *\}) \times \{R, W\} \times \{0, 1, \ldots, n\} \times (\Sigma \cup \{\#, *\}) \times Q$.*

The extended meaning of δ is as follows: let $(s, \sigma, op, i, \gamma, t) \in \delta$. If σ and γ are not '*', the meaning of the transition is as before. If $\sigma = *$ then the transition can be taken only if the input symbol is γ; similarly, if $\gamma = *$ then the transition can be taken only if the contents of the i-th register is σ. Finally, if both σ and γ are '*', the transition can be taken only if the input symbol and the i-th register's contents are identical.

An FSRA* for the language L_n, $n = 4$, is given in Figure 7. In the general case, the number of registers is n+1; registers $1, \ldots, n$ 'remember' the first n input symbols to be duplicated. Notice that the number of states depends only on n and not on the size of Σ. The language $\{ww \mid |w| \leq n\}$ for some $n \in \mathbb{N}$ can be generated by a union of FSRA*, each one generating L_n for some $i \leq n$.

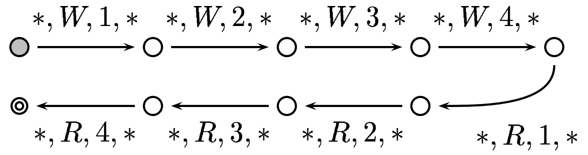

Figure 7: FSRA* for L_n, $n = 4$

4 Conclusions

We presented a novel FS model which efficiently accounts for medium-distance dependencies introduced by non-concatenative morphology. Our main objective was to overcome the inherent space inefficiency of existing approaches, while maintaining the desirable properties of FS techniques, such as the closure properties of FS networks. We provided a complete model, along with its mathematical and computational analysis, and exemplified its usefulness through concrete examples; in particular, we showed a dedicated operator for interdigitation and how it compiles to an FSRA.

The greatest appeal of FS technology for NLP is time efficiency. To maintain linear recognition time, automata must be determinized. We still do not have a determinization algorithm for FSRAs, although the examples we presented all guarantee

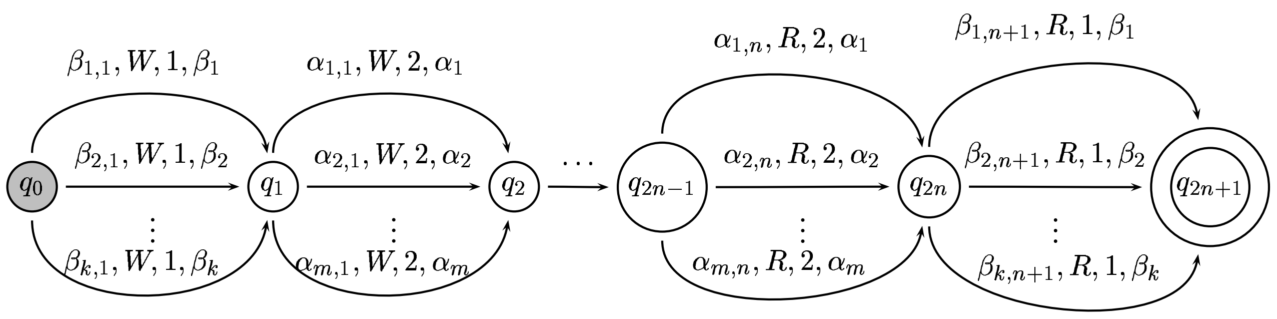
q_0
q_1
q_2
$\cdots$
q_{2n-1}
q_{2n}
q_{2n+1}
$\beta_{1,1}, W, 1, \beta_1$
$\beta_{2,1}, W, 1, \beta_2$
$\beta_{k,1}, W, 1, \beta_k$
$\alpha_{1,1}, W, 2, \alpha_1$
$\alpha_{2,1}, W, 2, \alpha_2$
$\alpha_{m,1}, W, 2, \alpha_m$
$\alpha_{1,n}, R, 2, \alpha_1$
$\alpha_{2,n}, R, 2, \alpha_2$
$\alpha_{m,n}, R, 2, \alpha_m$
$\beta_{1,n+1}, R, 1, \beta_1$
$\beta_{2,n+1}, R, 1, \beta_2$
$\beta_{k,n+1}, R, 1, \beta_k$

Figure 6: FSRA resulting from *splice*

linear recognition time. We intend to extend the results reported here by providing constructions for determinization, as well as for other closure properties (e.g., complementation). Future work also includes an extension of regular expressions to allow direct access to registers and dedicated operators for other non-concatenative phenomena.

We also extended the FSRA model to transducers. FSRTs are space-efficient devices that can be used to analyze, for example, Hebrew lexemes, providing their roots and patterns. For lack of space, we cannot exemplify this model here; however, it is a straight-forward extension of FSRAs.

Acknowledgments

We are grateful to the anonymous reviewers for detailed comments. This work was supported by the Israeli Science Foundation (grant no. 136/01).

References

Kenneth R. Beesley and Lauri Karttunen. 2000. Finite-state non-concatenative morphotactics. In *Proceedings of the fifth workshop of the ACL special interest group in computational phonology, SIGPHON-2000*, Luxembourg, August.

Kenneth R. Beesley. 1998. Constraining separated morphotactic dependencies in finite-state grammars. In *FSMNLP-98.*, pages 118–127, Bilkent, Turkey.

Glenn D. Blank. 1985. A new kind of finite-state automaton: Register vector grammar. In *IJCAI-1985*, pages 749–755.

Glenn D. Blank. 1989. A finite and real-time processor for natural language. *Communications of the ACM*, 32(10):1174–1189.

Yael Cohen-Sygal. Forthcoming. Computational implementation of non-concatenative morphology. Master's thesis, University of Haifa.

Michael Kaminski and Nissim Francez. 1994. Finite memory automata. *Theoretical Computer Science*, 134(2):329–364, November.

Ronald M. Kaplan and Martin Kay. 1994. Regular models of phonological rule systems. *Computational Linguistics*, 20(3):331–378, September.

Lauri Karttunen, Jean-Pierre Chanod, Gregory Grefenstette, and Anne Schiller. 1996. Regular expressions for language engineering. *Natural Language Engineering*, 2(4):305–328.

Laura Kataja and Kimmo Koskenniemi. 1988. Finite-state description of Semitic morphology: A case study of Ancient Akkadian. In *COLING*, pages 313–315.

Martin Kay. 1987. Nonconcatenative finite-state morphology. In *Proceedings of the Third Conference of the European Chapter of the Association for Computational Linguistics*, pages 2–10.

George Anton Kiraz. 2000. Multitiered nonlinear morphology using multitape finite automata: a case study on Syriac and Arabic. *Computational Linguistics*, 26(1):77–105, March.

András Kornai. 1999. Vectorized finite state automata. In András Kornai, editor, *Extended Finite State Models of Language*, Studies in Natural Language Processing, chapter 10, pages 95–107. Cambridge University Press.

Kimmo Koskenniemi. 1983. *Two-Level Morphology: a General Computational Model for Word-Form Recognition and Production.* The Department of General Linguistics, University of Helsinki.

Alon Lavie, Alon Itai, Uzzi Ornan, and Mori Rimon. 1988. On the applicability of two-level morphology to the inflection of Hebrew verbs. In *Proceedings of the International Conference of the ALLC*, Jerusalem, Israel.

Mehryar Mohri. 1996. On some applications of finite-state automata theory to natural language processing. *Natural Language Engineering*, 2(1):61–80.

Gertjan van Noord and Dale Gerdemann. 2001. An extendible regular expression compiler for finite-state approaches in natural language processing. In O. Boldt and H. Jürgensen, editors, *Automata Implementation*, number 2214 in Lecture Notes in Computer Science. Springer.