Chapter 4 : FINITE STATE MORPHOLOGY- THE THEORETICAL FRAMEWORK

In recent years finite-state techniques are probably the most prevalent approach employed for the purpose of automatic morphological analysis, mainly because of their simplicity and outstanding efficiency. A good number of alternative frameworks allow for straightforward implementation of finite-state networks. The FSA Utilities Toolbox developed at the University of Groningen (NL), foma (Hulden 2009), SFST tools by Helmut Schmid (2004), a collection of software tools for the generation, manipulation and processing of finite-state automata and transducers, etc. facilitates the realization of finite-state networks for automatic morphology systems. Even though foma, and the FSA Utilities Toolbox systems are open source, it is decided to use the Xerox Finite State Tools, in short xfst, for the purpose, as I was already familiar with this framework. However, a future re-implementation with one of the open source alternatives in order to make the system freely available to whomever might be interested.

The chapter presents the theoretical framework employed in this study. Starting with the origin of the finite-state automata to language processing theory, the subsequent sections introduce concepts related to automata theory. The finite-state technology, regular expression, regular language, regular relation, equivalence of regular expression to finite-state automaton and regular relation to finite-state transducer are discussed briefly. Operations on finite-state automata and finite-state transducers are briefed from the morphological processing point of view. Also an overview of the Xerox tools for morphological description and analysis are given here.

4.1 ORIGIN

In 1968 Noam Chomsky and Morris Halle (Chomsky and Halle, 1968) formalized a phonological grammar wherein an ordered sequence of "REWRITE RULES" was used to convert abstract phonological representations into its surface forms through a series of intermediate representations. These rewrite rules have the general form:

$$\alpha \rightarrow \beta / \gamma \_ \delta$$

where α, β, γ, and δ are arbitrarily complex strings and the rule is read as "α is rewritten as β in the environment between γ and δ". The problems with rewrite rules were that they are one-directional and cannot be used for generation purposes. α is no more available for other rules once it is rewritten as β. Also the morphological alternations were described by means of ordered rewrite rules but again there was no means to understand how such rules could be used for analysis.

In 1972, Johnson C Douglas observed in his published dissertation "Formal aspects of Phonological Rule description" that while the same context sensitive rule could be applied several times recursively to its own output, phonologists assumed it implicitly that the site of application moves either to the left or right in the string after each application. The constraint here is that any subsequent application of the same rule to the rewritten portion must not be affected and should be left unchanged. Kimmo Koskenniemi's "Two-level morphology" in his dissertation "Two-Level Morphology: General computational model for word-form recognition and generation" is the first computational approach to morphology and since then a good number of morphological analyzers has been developed using this theory.

4.2 FINITE-STATE TECHNOLOGY

Finite state technology in the area of natural language processing has been an important and active field of research and development for a number of decades. One of the fundamental results of formal language theory (Kleene, 1956) is the demonstration that finite-state languages are precisely the set of languages that can be

described by a regular-expression. Regular expressions and methods for compiling them into automata have been part of elementary computer science for decades (Hopcroft and Ullman, 1979). However, the classical regular expression calculus has been extended to accommodate the application of finite-state transducers to natural language application. In the finite-state formalism, languages are a set of strings of any kind of symbols; strings are concatenations of zero or more symbols. Here our notion of a symbol is single character such as ক, but multi-character symbols like +VR are also taken into consideration. Finite-state based systems study the behavior of a system composed of state, transition and actions.

Formally an FSA is a quintuple <Q, Σ, q0, F, δ> defined in the following way:

Q: a finite set of N states q0, q1, … qN

Σ: a finite symbols representing the input alphabet

q0: the start state

F: the set of final states $F \subseteq Q$

δ(q,i): the transition function or transition matrix between states. Given a state $q \in Q$. δ is thus a relation from $Q \times \Sigma$ to Q (Jurafsky and Martin 2000).

Finite-state automata are also described by using graphs and these graphical descriptions are called finite-state transition networks (FSTN). The following figure shows a simple finite-state automaton:

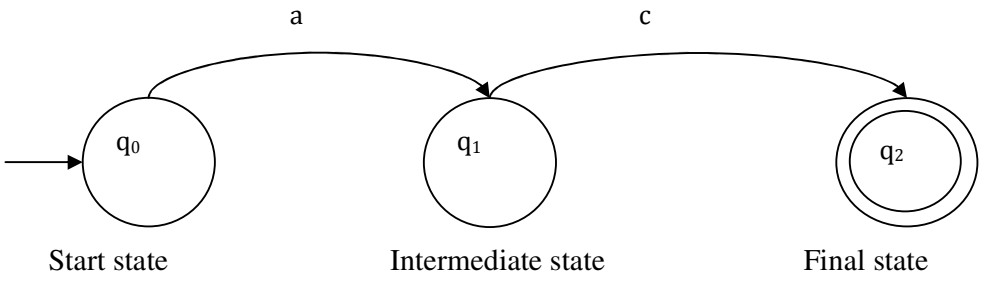

Figure 4-1: A Finite-State Automaton

q_0, q_1 and q_2 are states of the automaton of which q_0 is the start state and q_2, the final state. a and c are the two letters from the input alphabet. The double circle indicates a final state while a state with an arrow mark from nowhere indicates the start state. In

our figure above, when the machine is at the start state and reads an a, a transition is said to be made from state q_0 to q_1; at q1 it reads a c and the machine transits to q_2 and reaches the final state hence the string ac is said to be recognized or accepted by the machine. A path is a sequence of transition over arcs originating from the start state to a particular final state. In the realm of computational morphology with finite-state techniques, a path is a set of alphabets equivalent to a word in natural language. So at the core of any finite-state technology based system is the set of states with unique distinctive features and a set of arcs with direction that connects these states.

At the earlier stages of its development, finite-state technology was considered to be inefficient by the linguists- reason being that the technology is mathematically a formal and abstract device and believed that it doesn't have the descriptive power for natural language analysis. During its developing stages, it was not really powerful to account for the linguistic phenomena because of the lack of unfounded characteristics, properties and theories about the technology. But with the discovery of new theories and principles about the technique, it has become more appropriate and suitable for modeling the parts of languages that could be considered as finite and regular. In recent years, many areas of computational linguistics use finite-state machines. Their use can be justified by both linguistic and computational arguments (Mehrar Mohri, 1997). Linguistically, finite automata are convenient since they allow one to describe easily most of the relevant local phenomena encountered in the empirical study of language, often leading to a compact representation of lexical rules, or idioms and cliches that appears natural to linguists (Gross Maurice, 1989). Graphic tools also allow one to visualize and modify automata, which help in correcting and completing a grammar. Computationally, the use of finite-state machines is mainly motivated by considerations of time and space efficiency. Time efficiency is usually achieved using deterministic automata. The output of deterministic machines depends, in general, linearly, only on the input size and can therefore be considered optimal from this point of view. Space efficiency is achieved with classical minimization algorithms (Aho, Hopcroft, and Ullman, 1974) for deterministic automata though it is not an issue with the modern computers.

At its earlier stages, applications of finite-state automata in natural language processing ranges from the construction of lexical analyzers (Silverztein 1993) and the compilation of morphological and phonological rules (Kaplan and Kay, 1994; Karttunen, Kaplan and Zaenen, 1992) to speech processing (Mohri, Pereira, and Riley, 1996) show the usefulness of finite-state machines in many areas. In recent years the finite-state technology has become widely used in various natural language processing tasks such as POS disambiguation, tokenization, shallow parsing, etc. To understand the theory and realize our objectives one needs to understand some mathematical and computational notions and operations which are introduced in the following sections.

4.3 REGULAR EXPRESSION (RE) VIS-A-VIS FINITE-STATE AUTOMATA

Regular expressions are the standard notation for characterizing text sequences and it is used for specifying the text strings in searching text (Jurafsky and Martin, 2000:48-59). A regular expression denotes a set of strings or string pairs. They can be used to search for occurrences of these strings in a pattern. A simple regular expression may be a single character such as **a** from the English alphabet, or it can be as complex a combination of characters as [a b* c] which encodes an infinite set of strings: "ac", "abc", "abbc", "abbbc", etc.

Regular expression is a declarative formalism for a set of strings and the formula for specifying the set of strings is done with the help of regular expression operators. The regular expression meta-language consists of notations for basic symbols, notations for multi-character symbols, special symbol-like notations, grouping, iteration and optionality operators. The symbols and operators for regular expression may be different for different software but the underlying theoretical background is same for all. Our work uses the xfst notation for writing regular expression as we are employing the xfst tools (see section 4.7 xfst tools) for implementation of the language model (Kenneth R. Beesley, L. Karttunen, 2003, chapter 3). The following subsequent tables list the different notations of the regular expression meta-language for defining string patterns in a language. Every basic alphabetic symbol is, by itself,

a valid regular expression. Case is significant everywhere in xfst, so z is a separate symbol from Z.

Table 4-1: Notations For Regular Expression Basic Symbols

Symbol	Meaning
a	Alphabetic characters like a, b, c, etc.
"a"	Double-quoted normal alphabetic letter like "a" is equivalent to a.
"+"	A literal plus sign symbol; i.e. not a Kleene star.
%+	A literal plus sign; alternate notation.
"o" "oo" "ooo"	A symbol expressed as an octal value, where **o** is a digit 0-7, e.g. " 123".
"xHH"	A symbol expressed as a hexadecimal (hex) value, where **H** is 0-9, a-f or A-F, e.g. " xA3".
"uHHHH"	A 16-bit Unicode character, e.g. " u0633".
"n"	Newline symbol in escape notation as per Unix convention. Also \t (tab), \b (backspace), \r (carriage return), \f (formfeed), \v (vertical tab), and \a (alert).

Regular expressions may contain multi-character symbols like +VR, +N, +ASP, ^নি, ^বু, ত্রি, etc. That is**,** a multi-character symbol may also contain the regular expression operator symbols such as +, *, ^, etc. along with other symbols. The following table shows different ways of using multi-character symbols in a regular expression:

Table 4-2: Notations For Multi-Character Symbols

Multi-character symbol	Description
dog	Compiled as the single multi-character symbol dog.
"+Noun"	Compiled as the single multi-character symbol +Noun. The surrounding double quotes cause the plus sign to be treated as a literal letter, i.e. not the Kleene plus
%+Noun	Another way to notate the multi-character symbol +Noun. The percent sign literalizes the plus sign.
%^HIGH	The multi-character symbol ^HIGH, starting with a literal circumflex.
%[HIGH%]	The multicharacter symbol [HIGH], starting with a literal left square bracket and ending with a literal right square bracket
"[HIGH]"	Another way to notate the multi-character symbol [HIGH].

Regular expressions can also have notations for wild card symbols. The following table 4-3 shows it.

Table 4-3: Special symbol-like notations

Symbol-like notation	Meaning
?	Denotes ANY symbol
0	Denotes the empty (zero-length) string, also called epsilon.
[ ]	Denotes the empty string; equivalent to 0.

Complex regular expressions can be built up from simpler ones by means of regular expression operators (L. Karttunen, 2001). The regular expression calculus allows grouping simple regular expressions, repetition of certain/full part of a string pattern, optional selection of a certain pattern in the searched string to create complex regular expressions.

Table 4-4: Grouping, dteration and optionality operators

Symbol	Meaning
[ ]	Grouping brackets. [A] is equivalent to A.
A*	The Kleene star. Denotes zero or more concatenations of A with itself.
A+	The Kleene plus. Denotes one or more concatenations of A with itself. A+ is equivalent to [A A*].
(A)	Optional. (A) is equivalent to [A|0].
A^n	Where *n* is an integer, denotes n concatenations of A with itself.
A^{n,m} . .	Where *n* and *m* are integers, denotes n to m concatenations of A with itself
A^<n	Where *n* is an integer, denotes fewer than n concatenations of A with itself.
A^>n	Where *n* is an integer, denotes greater than n concatenations of A with itself

The traditional finite-state operations include union, intersection, minus (subtraction), complement (negation) and concatenation. It is to be noted that the concatenation operator does not have any explicit operator. Symbols required to be concatenated are simply typed one after another in the desired order. The following table shows operators and their syntax. The A and B are arbitrarily complex regular expressions.

Table 4-5: Operations on REs

Operator	Syntax and operands	Description
Union	A | B	The union of A and B.
Intersection	A & B	The intersection of A and B. Here A and B

		must denote languages, not relations
Subtraction	A - B	The subtraction of B from A. Here A and B must denote languages, not relations.
Complement	~ A	The language complement of A, i.e. [?* - A]. Here A must denote a language, not a relation
Concatenation	A B	The concatenation of B after A. The operands are separated by white space; there is no explicit concatenation operator.

The regular expression {অফব} is compiled as a concatenation of the three symbols অ, ফ and ব. Surrounding a string of symbols with curly braces "explodes" them into separate symbols, here {অফব} is equivalent to [অ ফ ব]. Some important operators used in the regular expressions, patterns and their meanings are listed in the following table:

Table 4-6: Examples of RE patterns and their meaning

Operators in RE	Pattern	Meaning
[ ]	[গী|কী]	গী or কী
[-]	[A-Z]	any one of the capital letters
^	[^a-z]	not a lowercase letter
*	ab*c	zero or more bs
.	অ.ব	any character between অ and ব
?	ab?c	either zero or b in between a and c
+	ab+	one or more bs
|	a|b	either a or b
()	appl(y|ies)	apply or applies
{n}		n occurrences
{n,m}		from n to m occurrences
\n		a new line
\t		a tab

Source: D. Jurafsky and Martin, 2000

Simple alphanumeric symbols with different operators can be combined together to form very complex regular expressions.

Finite state machines (or automata) (FSM, FSA) recognize or generate regular languages, exactly those specified by regular expressions. Any regular expression is equivalent to finite-state network that represents the corresponding sets of strings. Therefore they can be compiled into a finite-state network that compactly encodes the corresponding language or relation that may well be infinite. As for instance, the regular expression [ab*c] may be represented by the following finite-state transition network (FSTN).

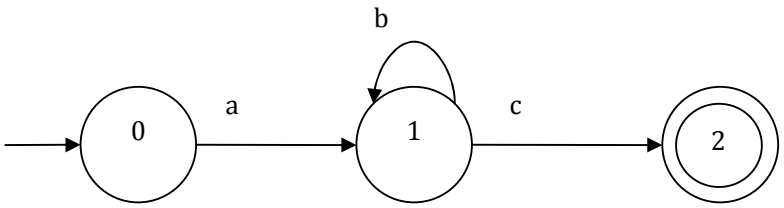

Figure 4-2: Finite-State Transition Network For RE [a b* c]

4.4 REGULAR LANGUAGES AND REGULAR RELATION

It is clear that regular expressions define a set of strings over an alphabet. So basically, regular expressions, like Boolean logic, denote sets. And the set of strings as defined by the regular expression is called a regular language. Formal languages are a set of strings each of which is composed of symbols from a finite symbol-set called an alphabet. A regular language is a formal language that is possibly an infinite set of finite sequences of symbols from a finite alphabet that satisfies particular mathematical properties. The following table shows regular expression with corresponding regular relation generated by the expression.

The language in right column of Table 4-7 is a regular language denoted by the RE in the left column. All the strings (words) matched by a regular expression [চা].*[গনিাকনি].* have the same pattern.

Table 4-7: Example of an RE with corresponding Regular Language

Regular Expression	Regular language
[চা].*[গনিাকনি].*	চাগনি cagəni, চাখিগনি cakhigəni, চারগনি cargəni, চারমগনি carəmgəni, চারককনি carəkkəni, চারকলমগনি carəkləmgəni, চাহনগনি cahəngəni, চাহনলমগনি cahənləmgəni, চারুগনি carugəni, চানগনি canəgəni, চানরগনি canərəgəni, চামিননগনি caminnəgəni, চাহনলমগনি cahənləmgəni, চাবিগনি cabigəni, চাগনিকো cagəniko..

In the context of finite-state morphology we need to distinguish between two types of sets- set of simple strings and set of pairs of strings. The former case defines a regular language and the later defines a regular relation between two regular languages. Or conversely, regular language and regular relation refer to sets that can be described by regular expressions. No doubt, regular relations can also be captured by using finite-state networks (this automaton is called finite-state transducer or FST) just like regular languages. The difference here is that the labels of the arcs are a pair of symbols instead of a single, individual symbol. The pair of symbols, say for two symbols a and b is denoted as a:b in regular expressions; symbol a is referred to as the upper symbol and b, the lower symbol. In computational morphology, the upper symbols represent the lexical category and hence the underlying representation of the morphemes of a language and the lower symbols represent the morphemes in the language. A regular relation may always be viewed as a mapping between two regular languages. The a:b relation is simply the cross product of the languages denoted by the expressions a and b. The kind of relation it offers is of one : one; one : many; many : one; many : many.

Regular relations participate in the same set of operations as regular languages, namely

Concatenation
Iteration
Complementation
Alternation (Union)
Intersection

A class of languages is closed under some operation if applying the operator to languages in the class always produces another language in the class. The regular expression calculus allows us to perform algebraic calculations on regular expressions thereby on regular languages and regular relations. The following table shows the regular expression operators applicable to regular languages and regular relations.

Table 4-8: Operations on RE and RL

Operation	**Regular Languages**	**Regular Relations**
Union	yes	yes
Concatenation	yes	yes
Kleene star	yes	yes
Iteration	yes	yes
Intersection	yes	[5]no
Subtraction/difference	yes	no
Complement	yes	no

Regular relations can be constructed by means of two basic operators (Karttunen, Lauri.2001):

A .x. B /Crossproduct

A .o. B /Composition

The crossproduct operator, .x., is used only with expressions that denote a regular language; it constructs a relation between them. [A .x. B] designates the relation that maps every string of A to every string of B. The notation a:b is a convenient shorthand for [a .x. b].

Composition operation on relations yields a new relation doing a mapping of strings Qfrom the upper language P to strings that are in the lower language of P. Symbolically, composition is expressed as

[5] However, regular relations with input and output of equal lengths are closed under intersection.

[P.o.Q]

Here P and Q are two relations and .o. denotes the composition operation. So for a pair <p,q> in P and for a pair <q,r> in Q, the pair <p,r> is said to be in composite relation.

Composition operation can be performed on more than two or more relations producing a single relation.

4.5 FINITE-STATE TRANSDUCER

A Finite-State Transducer (FST) is basically an enhanced finite-state machine/automaton. A finite-state automaton can only accept or reject a string; whereas an FST can transform one string into another. In that an FSA is a trivial FST where the input and output alphabets are identical and the transitions in the machine always reproduce the input as the output.

Formally an FST can be defined in the following way in terms of six parameters:

- A finite **set of states** $Q = \{q0, q1,.., qn\}$
- A finite **alphabet** Σ of input symbols (e.g. $\Sigma = \{a, b, c,...\}$)
- A finite **alphabet** Δ of output symbols (e.g. $\Sigma = \{+N, +pl,...\}$)
- A designated **start state** $q0 \in Q$
- A set of **final states** $F \subseteq Q$
- A **transition function** $\delta: Q \times \Sigma \rightarrow 2Q$

$$\delta(q,w) = Q' \text{ for } q \in Q, Q' \subseteq Q, w \in \Sigma$$

A finite-state transducer accordingly implements a relation between two formal languages: an upper-side and a lower-side regular language, and it literally 'transduces' strings from one language into the other. The process of transformation is called transduction. Given a string as input it produces a corresponding output. It works on two tapes- reading an input from a tape and writing an output to another tape- unlike finite-state automaton (which works with only one tape). In a non-deterministic finite-state transducer, more than one possible output for a given string

may be produced. There are many applications of transductions in natural language processing applications. It can be imagined to transform strings of letters into strings of phonemes (sounds), or word strings into part-of-speech strings (noun, verb, etc.) with the help of finite-state transducers.

For the purpose of computational morphological analysis, finite-state transducers can be used to map between the lexical and surface levels of Kimmo's 2-level morphology. An FST to map the vowel letters to its corresponding matra in Bengali is shown below:

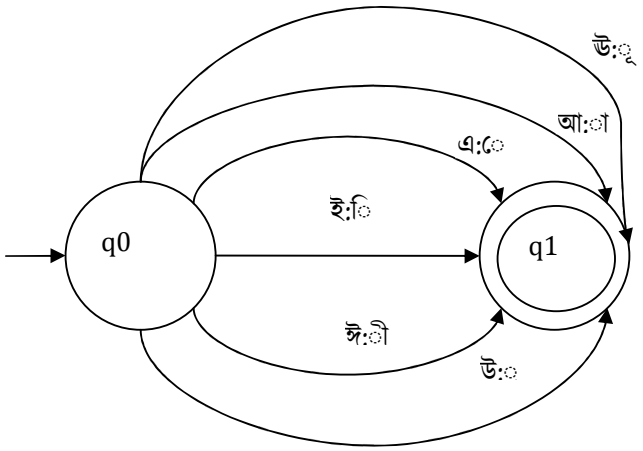

Figure 4-3: FST for some Bengali Vowel letters to its corresponding matra

A regular relation on strings can be modeled by a finite-state transducer if its non-contextual part is not allowed to apply to its own output. So, regular relations are well represented by finite-state transducers for a mapping from one language to another. The pair of string labels of each arc of an FST represents, therefore, a relation or a mapping between two sets of strings. For the purpose of morphological analysis, the process of transduction is a simple mapping of surface forms to its corresponding lexical forms. FST networks are inherently bi-directional; so these systems can also be used for generation of valid surface forms of words by processing in the reverse direction from the lexical to the surface form.

4.5.1 SOME IMPORTANT OPERATIONS ON FINITE-STATE TRANSDUCERS

Regular relations are well represented by finite-state transducer networks. So are all its operations, viz. concatenation, iteration, complementation, alternation (Union), are applicable to the finite-state transducers representing regular relations.

However, the operations- complement (negation), intersection, relative complement (minus) can only be combined with regular expressions that denote a regular language.

Union: Union operation can be performed on two or more finite-state transducers to yield a single finite-state transducer. The resulting transducer contains all the elements of the constituent transducers. The ordering of the arcs in the network is immaterial. When two FST networks A and B are operated upon with union, it is written as [A|B].

To illustrate the concept of union operation on transducers, consider three transducer networks Verb, Adverb and Adjective. When union operation is performed on the three networks, it results into a single transducer network which contains all the elements of the three networks as illustrated in the following figures:

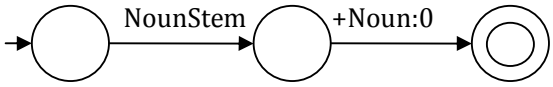

Figure 4-4: FST for noun stems

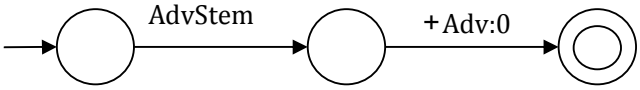

Figure 4-5: FST for adverbs

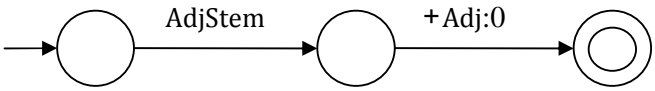

Figure 4-6: FST for sdjectives

The result of union on the above three transducers, viz. is shown below:

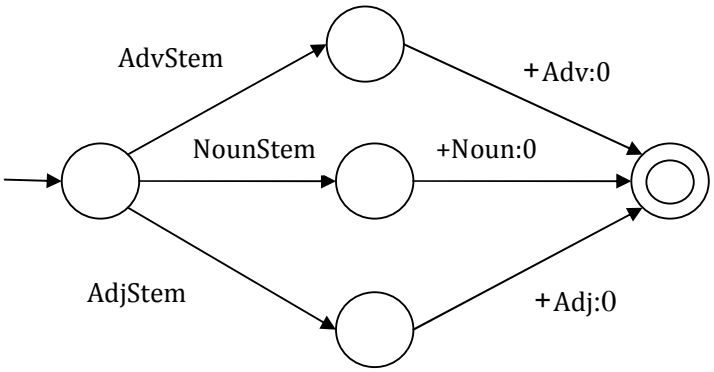

Figure 4-7: Union of noun, adverb and adjective FSTs

Union operation is helpful in developing and constructing large and more complex networks from smaller and simple FST networks of morphological word classes, thereby allows working on modular concept.

Concatenation: Concatenation operation allows the networks to be kept in sequence. Two existing finite state networks can be concatenated with one another to build up new words productively or dynamically (Beesley and Kartumnen, 2003). For two networks A and B, the concatenation operation on A and B is denoted as [A B].
The following figures illustrate the phenomena of concatenation on FST networks.

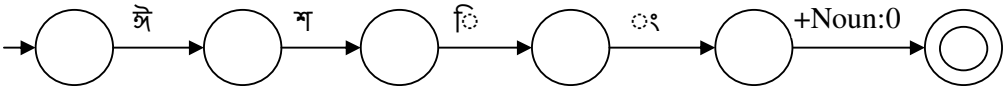

Figure 4-8: FST for noun stem "ঈশিং"/ water

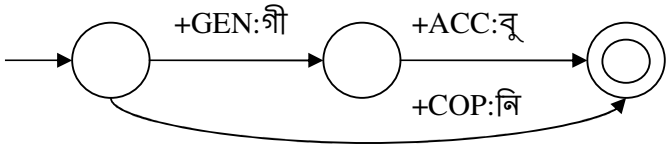

Figure 4-9: FST for copula নি, genitive গী, and accusative suffix বু

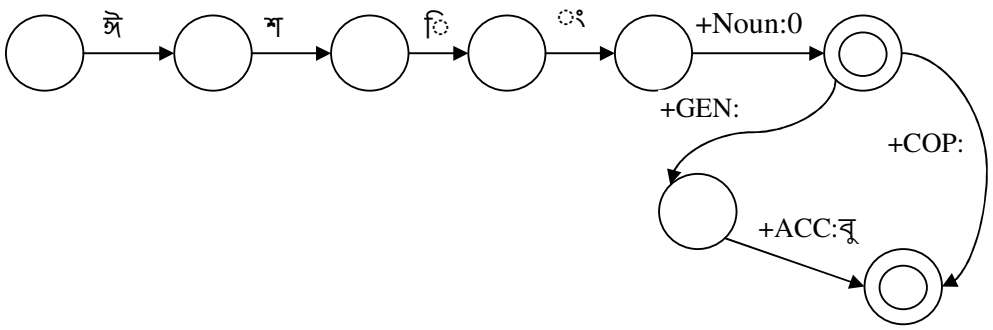

Figure 4-10: Concatenated FST for the above two FSTs

Figure 4-8 and 4-9 represents two FSTs- one for noun stem "ঈশিং" and another for genitive –গী/gi, copula -নি/ni and accusative –বু/bu. Figure 4-10 is the result of concatenation operation on these two FSTs. This FST can analyze and generate the copula, genitive and accusative forms of the noun ঈশিং. Concatenation operation is useful in handling the inflectional and derivational morphological phenomena of the nominal and verb stems.

Composition: Composition operation is performed on two or more languages/ relations to remove common elements from the participating FST networks. Composition is denoted as [A.o.B] (Beesley and Kartumnen, 2003). The following figures illustrate the composition operation.

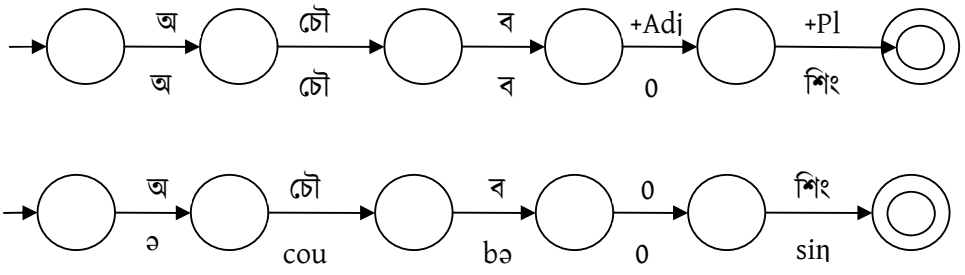

Figure 4-11: FSTs for <অচৌব+Adj+Pl, অচৌবশিং> and <অচৌবশিং, əcoubəsiŋ>

When the two transducer networks are composed, it results into a single FST network as depicted in the following FST. The composition operation removes the common

symbols in the middle. The upper symbols from the first FST network and the lower symbols from the second network forms a relation for the resulting transducer.

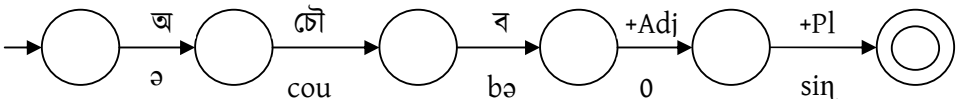

Figure 4-12: FST after applying composition on the above two FST networks

Composition operation forms a sequence of transducers. It transforms a cascade of FSTs into a single FST by eliminating the common intermediate outputs. This feature of composition allows working for a modular structure. Each spelling rule for morphophonemic alternation can be compiled as a single rule transducer and composing these rules with root/stem lexicon network helps to obtain the correct surface forms of morphological words.

Intersection: Denoted as [A & B], intersection produces a network which has elements common to both the networks. Though not a major player in this study, it can be used to find the common words between two sets of words.

Subtraction: Subtraction of one network from another contains the set containing elements that are in A but not in B. Denoted as [A-B], the operation is normally performed to find the words in a network which are not in another network.

Complementation or negation: The complement language of a network is the set of all strings that are not in the language of the network. The operation is written as ~A, where A is a FST network. Complementation operation is useful for filtering the words from a network.

Projection: Projection operation returns all the strings of a regular language participating in a regular relation. It can either project the set of strings of the upper language .or the lower language one at a time.

Among all the operations explained above, union, concatenation and composition operations are used during the implementation of the analysis of morphological categories and rules to create a single lexical transducer while others are used elsewhere.

4.5.2 CLOSURE PROPERTIES OF FINITE-STATE TRANSDUCERS

The strength of the formalism of finite-state automata mainly comes from a very few important results. As for finite-state automata, finite-state transducers get their strengths from various closure properties and algorithmic properties. [6] Kleene's theorem is one of the first and most important results about finite-state automata. The theorem relates the class of languages generated by finite-state automata to some closure properties. This result makes finite-state automata a very versatile descriptive framework for implementation of different applications especially in the field of natural language processing.

A set operation such as union has a corresponding operation on finite-state networks only if the set of regular relations and languages is closed under that operation. Closure means that if the sets to which the operation is applied are regular, the result is also regular, that is, encodable as a finite-state network. Languages are made up of words; words on the other hand are made up of characters or letters from an alphabet. The descriptive power of the automata theory combined with the closure property of different operations make finite-state transducers a very suitable choice for the representation of the word structure of languages. Finite-state transducers are closed under union, concatenation and composition but not under intersection (possible for equal length regular relations) and complement.

Closed under concatenation

Finite state transducers are closed under concatenation operation. That means if T1 and T2 are two FSTs, there exists another FST [T1 T2] which contains all the elements from both the networks in the order of concatenations.

[6] Kleene's theorem refers to S.C. Kleene who originated the regular expression notation and Kleene operator *.

Closed under union

Finite-state transducers are closed under union operation.

That is, if T1 and T2 are two FSTs, there exists a FST $\mathbf{T1\ U\ T2}$ such that $\mathbf{|T1UT2|= |T1|U|T2|}$, i.e. such that $\forall \mathbf{u} \in \mathbf{|T1UT2|(u) = |T1|(u)U\ |T2|(u)}$

Closed under composition

Finite-state transducers are also closed under the composition operation. This operator removes the common elements from the networks used for composing (Beesley and Kartumnen 2003).

In the regular expression notation, it is written as
T1 .o. T2;
where T1 and T2 are finite-state transducers.

If L1, L2, and L3 are regular languages and there are two transducers T1 and T2 such that
$\mathbf{T1 = L1\ x\ L2}$ and $\mathbf{T2 = L2\ x\ L3}$,
then, a third transducer T3 can be composed of T1 and T2,
i.e. $\mathbf{T3 = T1\ o\ T2 = (L1\ x\ L2)o\ (L2\ x\ L3) = L1xL3}$.

T3 accepts the strings of the input of T1 and generates the output which would have been generated by first transforming the input to the output of T1 and then using that as the input of T2. In other words, composition creates a machine which produces the same results as processing input serially through two FSTs. A pair of transducers connected through a common tape models the composition of the relations that those transducers represent. The pair can be regarded as performing a transduction between the outer tapes, and it turns out that a single finite-state transducer can be constructed that performs exactly this transduction without incorporating any analog of the intermediate tape. In short, the relations accepted by finite-state transducers are closed under serial composition (Kaplan and Kay, 1994). The composition operation of T1 and T2 can be depicted diagrammatically using a block diagram as in the following figure:

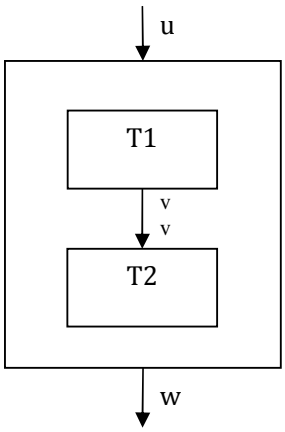

Figure 4-13: Block Diagram of Transducer composition

An example to perform composition on two transducers is shown below:

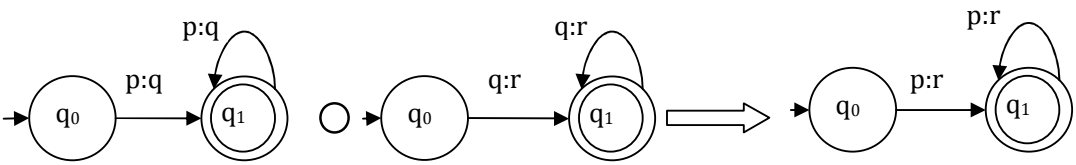

Figure 4-14: Example showing Transducer composition

Use of finite-state transducers in natural language processing applications such as morphological analysis, the complexity involved in specifying one (big) transducer that can deal with all spelling rules, can be well imagined. But the closure properties of the operations applicable to finite-state transducers make it possible to specify one (smaller) transducer per rule for each spelling change. There are ways of combining these individual transducers into one big transducer with the help of the operations like composition, union, concatenation, etc.

4.6 TWO-LEVEL MORPHOLOGY

Two-Level Morphology describes phonological alternations in finite-state terms. Rules constraining the surface realizations of lexical strings are applied in parallel and constrain a certain lexical/surface correspondence and the environment in which the correspondence was allowed, required, or prohibited. Two-Level Morphology formalism was a constraint-based model that did not depend on a rule compiler,

composition or any other finite-state algorithm. The main idea behind the two-level morphology is based on three concepts: (L. Kartunnen, 2001)

- Rules are symbol-to-symbol constraints that are applied in parallel, not sequentially like rewrite rules.
- The constraints can refer to the lexical context, to the surface context, or to both contexts at the same time.
- Lexical lookup and morphological analysis are performed in tandem.

Koskenniemi's 1983 system represented the lexicon as a forest of tries (known as letter trees), tied together by continuation-class links from leaves of one tree to roots of another tree or trees in the forest (Kenneth R Beesley, L Kartunnen, 2003). In the history of computational linguistics, Koskenniemi's two-level morphology was the first practical language independent general model for morphological analysis of morphologically complex languages. The language-specific components, the rules and the lexicon, were combined with a universal runtime engine applicable to all languages. The two-level model differs from generative phonology in that it proposes parallel rules instead of successive ones. In this way it avoids the existence of intermediate stages in the derivation of single word forms (Kimmo Koskenniemi, 1983). A copyrighted, freely distributed implementation of the Two-Level Morphology, called PC-KIMMO, available from the Summer Institute of Linguistics (E.L. Antworth, 1990), runs on PCs, Macs and Unix systems. This lexicon can be thought of as a non deterministic, un-minimized finite-state network.

4.7 XEROX FINITE-STATE TOOL

Xerox finite-state tools are an integrated set of software tools to facilitate linguistic development of traditional grammar components such as lexicons, morphotactic rules, morphotactic filters, phonological and orthographical alternation rules.

xfst is an interactive interface providing access to the basic algorithms of the Finite-State Calculus, providing maximum flexibility in defining and manipulating finite-state networks. **xfst** also provides a compiler for an extended metalanguage of regular expressions, which includes a powerful rule formalism known as replace rules.

lexc is a high-level declarative language used to specify natural-language lexicons. The syntax of the language is designed to facilitate the definition of morphotactic structure, the treatment of gross irregularities, and the addition of the tens of thousands of baseforms typically encountered in a natural language. **lexc** source files are produced with a text editor such as emacs, and the result of the compilation is a finite-state network.

twolc is a high-level declarative language designed for specifying, in classic two-level format, the alternation rules required in phonological and orthographical descriptions. As with **lexc**, **twolc** source files are written using a common text editor and are compiled into finite-state networks. (Beesley & Karttunen, 2003).

Our research study uses lexc and xfst. lexc is used for compiling the lexica of different sub category morphological word classes into a single lexicon and xfst for compiling the spelling change rules for morphophonemic alternations. The xfst interface is used for running the scripts to perform various FST network operations.

4.7.1 LEXC

lexc, for **Lex**icon **C**ompiler, is a high-level declarative programming language and associated compiler for defining finite-state automata and transducers (Finite State Morphology, Kenneth R Beesley, L Karttunen, 2003). Suitable for defining natural language lexicons, lexc is normally employed for defining the morphotactics of a language in a Xerox morphological analyzer. The input to the lexc program is a text file. The result of lexc compilation of a valid input file is a finite-state network for automat`a or a transducer. The idea here is based on the theory of two-level morphology (Kimmo Koskenniemi, 1983, 1984). In the Xerox formalism, the lexicon is actually compiled into a minimized network, typically a transducer, but the filtering principle is the same. Individual entries in the lexicon consist of two parts- a form and a continuation class. While a form is single symbol in case of finite-state automata, a form is represented as a two way symbol – upper and lower, specifying a two-level relation between the upper and the lower symbol in case of finite-state transducers. The lexicon contains roots, stems, inflectional/derivational morphemes. The format of a lexc text file may be represented as follows:

Multichar_Symbols declaration
symbols

Lexicon Root
root entries

Lexicon class1
upper : lower class2; ! a comment. Each entry is terminated with a semicolon
#; ! # is a special continuation class specifying end of word
! equivalent to a final state in the network.
Lexicon class2
.
.
Lexicon class3
.
.
.
.
.
Lexicon classN

End ! An optional **End** at the end of the file indicates that the compiler !wont read anything beyond this.

All the tags like Noun, Verb, Acc (for accusative), Gen (for genitive), etc. should be declared under the Multichar_Symbols declaration section. The conventional way of declaring them is to add a "+" sign before the tags. e.g. to declare Noun for +Noun and Accusative for +Acc. The Root lexicon is must for every lexc file. It represents the start state in the compiled finite-state network. Every other lexicon in the file represents a state which is a continuation class. The form entry in the lexicon is the arc in the finite-state network.

4.7.2 XFST

xfst is a programming language for regular expressions. The result of the compilation is a finite-state network. The language of regular expressions is a formal language, similar to describing Boolean logic formulae. Regular expressions have a simple syntax but the expressions can be arbitrarily complex. Like formulas of Boolean logic, regular expressions denote sets.

xfst includes many other regular expression operators other than concatenation, union, and minus. A .x. B is the crossproduct of two regular expressions representing the languages A and B denotes a regular relation that maps every string in A (upper side) to all strings in B (lower side), and vice versa. Thus [a .x. [b | c]] is equivalent to [a:b | a:c]. Here A and B must be simple languages, not relations. Another very important operation implemented by xfst is the composition operator applicable to regular relations. A .o. B is composition of the relations A and B. The intersection of the lower language of A and the upper language of B works like a filter here. For example, the expression [a:b | b:c] .o. [a:x | b:y] is equivalent to [a:y]. The b:c and a:x pairs do not contribute anything to the result because neither \c" nor \a" is in the intersection of [b | c] (the lower language of A) and [a | b] (the upper language of B).

Xerox has addressed the two central problems of computational morphology – morphotactics and alternation and can be solved with the help of finite-state networks.

1. The legal combinations of morphemes (MORPHOTACTICS) can be encoded as a finite-state network;

2. The rules that determine the form of each morpheme (ALTERNATION) can be implemented as finite-state transducers; and

3. The lexicon network and the rule transducers can be composed together into a single network, a LEXICAL TRANSDUCER, that incorporates all the morphological information about the language including the lexicon of morphemes, derivation, inflection, alternation, infixation, interdigitation, compounding, etc. (Beesley and Karttunen, 2003).

The Xerox finite-state tools have been written to accommodate UNICODE input, which would have distinct advantages in many applications (Beesley and Karttunen, 2003). Any Unicode compliant alphabet letter can be used for an entry as input at the xsft interface.