Chapter 6

Assamese Morphological Analyser: Architecture & Prototype

(This chapter describes Assamese Morphological Analyser’s Architecture & Prototype and implementation details for finite-state transducer)
Chapter 6

Assamese Morphological Analyser: Architecture & Prototype

Morphological structure of Assamese Language is complex in nature. Assamese words may be a root words without any affixation or it may contain one or more affix, where affix may be prefix or suffix or both.

∅/prefix|root|∅/(suffix)*

Figure 6.1: Morphological structure of Assamese Word(s)

The above figure (Figure 6.1) shows morphological structure of Assamese words. In general Assamese words may contain one prefix but in case of suffix, word may contain more than one suffix.

There are different Finite State Transducer (FST) based models for implementing Morphological Analyser for a Language. For example; Helsinki Finite-State Transducer Technology (HFST), Foma finite-state library, OpenFST library, Xerox Finite State Tool (XFST), Apertium with lttoolbox, Stuttgart Finite State Transducer (SFST) etc. The models can be choosing according to the compatibility with the language.
Initially we start working with Apertium lttoolbox to design our morphological analyser. Later on we dropped the idea of using Apertium lttoolbox and choose SFST tool for our work.

Using SFST tool we have design our basic model of Assamese morphological analyser and extend this design to a hybrid model of Assamese Morphological Analyser which will be discussed in next chapter.

6.1. Implementation of Assamese Morphological Analyser Using Apertium lttoolbox

Apertium is a rule-based open-source shallow-transfer machine translation platform [40,41]. It is free software and released under the terms of the GNU General Public License. It includes the engine, maintenance tools, and open linguistic data for several language pairs. lttoolbox is a toolbox for lexical processing, morphological analysis and generation of words. lttoolbox used finite-state transducers (FST). FST are a type of finite-state automata, which may be used as one-pass morphological analysers.

In Apertium, the analyser data is stored in Apertium’s dictionary “.dix” format with XML syntax. The analyser can be easily converted to a morphological generator from the single morphological database “monodix”, depending on which direction the system read the dictionary. If the system read the dictionary from left to right, we obtain the analyser, and read from right to left, we obtain
the generator. It is proven that an XML based dictionary “monodix” is generally faster than a normal text or database based dictionary.

For creating Morphological Analyser, different modules of Apertium are required.

- **Dictionary**: An Apertium based system can use two types of dictionaries, Monolingual and Bilingual Dictionary. Monolingual dictionary is used for Morphological analyzer & generator and Bilingual dictionary is used for machine translation purpose. For our work, we use monolingual dictionary. All the dictionary entries for our monolingual dictionary is created manually. The root words lexical categories are verified by linguistics so that the output of the analyzer is proper.

- **Paradigm definitions <pardef>**: A Paradigm is the complete set of related inflectional and productive derivational word forms of a given category. A paradigm can be understood as a small dictionary of alternative transformations that can be concatenated to the parts of words (or to entries of another paradigm) to specify regularities in the lexical processing of the dictionary entries, such as inflection regularities. In the definition along with the root word it contains other information like category, gender, number, person, case marker, tense etc.
Figure 6.2: Dictionary entry of Assamese Word _eye (Eye) in Paradigm

- **Element for Reference to a Paradigm**: Apertium provides a “lexico-semantic” layer, for working with inflection of a word. The layer introduces the lexemes into derivation and concurrently follows the inflection of the derived lexeme. It is used inside `<pardefs>` entry. Main advantage of using reference paradigm is that, there is no need to write all the inflected forms of a lemma in a morphological dictionary entry because it can be referred from other paradigms.

  ```xml
  <e lm="_eye"> 
  <i>_eye</i> 
  <par n="_eye_n"/> — Reference Paradigms for Inflection
  </e>
  ```

  Figure 6.3: XML format for Reference Paradigm

- **Morpheme**: All the root word (morpheme) is included in the dictionary, generated for Morphological Analyser. The dictionary is different from a
conventional dictionary, because it contains other information with morpheme like lexical categories and their corresponding paradigm.

![Diagram of dictionary entry](image)

Figure 6.4: Dictionary Entry of Morpheme with their lexical category

- **lttoolbox Modules**: lttoolbox contains three modules, lexical processing “lt-comp”, morphological analysis & generation “lt-proc” and Expansion “lt-expand”.

For Morphological analysis “lt-comp” and “lt-proc” module is required, lt-comp for processing and “lt-proc” for generation [42]. “lt-comp” module is responsible for compiling our morphological dictionaries into its own finite-state representation and “lt-comp” module is responsible for processing the compiled input data into required output.

- **Compilation**: “lt-comp” module compile the given “.dix” format file into binary format from left to right (LR) or from right to left (RL). When we compile with LR, it creates an analyser and RL usually creates a generator.
- **Syntax of lt-comp:**

  ```
  $ lt-comp lr apertium-asm.morph.dix analyser.bin
  ```

  Compile the *apertium-asm.morph.dix* dictionary in a left-to-right manner into the binary *analyser.bin*

  ```
  $ lt-comp rl apertium-asm.morph.dix generator.bin
  ```

  Compile the *apertium-asm.morph.dix* dictionary in a right-to-left manner into the binary *generator.bin*

- **Processing:** “lt-proc” module contains two functions, one is analysis (which is the default mode) and generation. Analysis converts surface forms into the set of possible lexical forms, while generation converts a lexical form into the corresponding surface form.

- **Syntax of lt-proc:**

  ```
  $ echo "চুুুুুী" | lt-proc analyser.bin
  ```

  Output: `চুুুুুী/চুুুুী<n><pl>$`

  Here we analyse the Assamese word `চুুুুী` (Eyes) with the binary format dictionary (left-to-right) analyser.in

  ```
  $ echo "^চুুুুী<n><pl>$" | lt-proc -g generator.bin
  ```

  Output: `চুুুুী`

  Here we generate the plural form of Assamese word `চুুুী` (Eye) with the binary format dictionary (right-to-left) *generator.bin*
Meaning of Analyzers Output format: Apertium lttoolbox provides output according to their own format.

Output format for Analyser mode

^Inputted word / Analysed word <word information1><word information2>..<word information n> $

![Figure 6.5: Meaning of output format (Morphological Analyser)](image)

If the word analysed by the analyser is to be displayed according to the user requirement; we need to take the help of other programming interface to display the word accordingly.

In our experiment, we have not designed any interface for Apertium based Analyser. We have analysed words in this analyser using command prompt only.
6.2. Implementation of Assamese Morphological Analyser Using SFST tool

In previous section, we have discussed a basic morphological analyser which is designed with Apertium lttoolbox. We have found that Apertium lttoolbox is good for analysing morphology of a word. But installing the Apertium package we have found that, lots of dependency is there. Moreover, we have found complexity for setting the analyser for prefix words. So we use another FST model; SFST for implementing our Morphological Analyser for Assamese Language

6.2.1. System Architecture for SFST based Morphological Analyser

In our Morphological Analyser, we mix up SFST tool command with Java program to get our requirements. We have divided our Morphological Analyser into two Java modules and one SFST module. We named these two modules as Java Module 1 and Java Module 2. We have design a Graphical User Interface (GUI) for easy interaction by the user with the system. The GUI is design with Java Server Page (JSP). The complete system is working as a web based Morphological analyser. The users provide input in the web based GUI and it is process by our Java module 1 and passed it to SFST module. The output of SFST module is process by Java module 2 and display the final result in web base GUI.

- **Java Module 1**: This Java Module is a combination of several sub modules to control the behaviour of SFST Module.
- **Input Module**: Our Morphological Analyser can take input as a single word, a sentence. After taking input, it passes the word or sentence to the next sub-module.

- **Tokenizer**: This module takes input from the input module and processes it to individual tokens. Here we consider valid words as tokens for processing. Token or word is selected based on the delimiter we consider. In our work, we consider space, (comma),” (double quote), .(dot), -(hyphen), | (Assamese stop) as a delimiter for selecting tokens. Then the tokenizer creates a file where it contains all the tokens as one token in a line.

- **Corpus Cleaner**: This module is used to clean unwanted characters in our inputted text corpus. Since all the delimiters and stop words we considered in the previous step for selecting tokens are selected for removing in this step.

- **Communicator**: This module is used to communicate with the SFST module and send all the cleaned tokens to the SFST module.
Figure 6.6: System Architecture of SFST Based Analyser
Java Module 2:

- **Output Processor:** After processing a word by SFST Module, result is taken by this module as input. SFST Module has its own output format. So, this module processes the output of SFST Module and provides morphology of a word in easily understandable format as output.

- **Output Writer:** This module is responsible for writing output of previous module in required display unit. After processing the output file according, the required format, this module sends the output to display in web based GUI.

SFST Module: This module contains SFST tool for processing morphology of Assamese words. SFST. This module takes input from Java Module 1, process word and sends output to Java Module 2 for display. Following section describes SFST based Assamese Morphological analyser.

6.2.1.1. Finite State Morphological Analysis with SFST Tool

Implementation of morphological analysis by compiling regular expressions into finite state networks is purposely defining a data structure rather than procedural
programs. The skeleton for morphological analysis using finite-state technique has been the construction of lexical transducers.

A finite-state source lexicon that defines the set of valid lexicon forms of the language (possibly infinite), and is a specification for the morphotactics of a category of a word and a set of finite state rules that assign the proper surface realization to all lexical forms and morphological categories of the language. The rules are compiled to transducers and merged with the source lexicon using intersection and composition operation. Every rule transducers represents a single change rule.

A lexical transducer is a set of transducers for orthographic rules with a transducer encoding the source lexicon. The lexical transducers encode the relation between the infected surface forms of root/stem and their corresponding lexical forms or lemmas, each containing a citation form of the word followed by a part-of speech tag.

A computational morphology of an inflected word can be analysing using SFST tool and as a result, it will return root word along with its morphological information.

For Example, if we input word “dogs” to SFST tool, it will return

(i) The base form (lemma) “dog”
(ii) The part of speech “Noun” and
(iii) Additional features such as the number “plural”.


The root of the word “dogs” to “dog” is accomplished by the transducer 
dog<>:s

The operator ‘:’ is here used to map the final symbol “s” to the empty symbol 
which is represented by “<>”.

The expression dog<>:s is an abbreviation of the expression 
d:d o:o g:g <>:s

The symbol on the left-hand side of a colon belongs to the analysis string, the 
symbol on the right-hand side to the surface string.

In order to add information about part of speech and the number feature to the 
analysis string, the transducer should be modified as dog<Noun>:s<plural>:<> 
<Noun> and <plural> are multi-character symbols which are treated like a single 
character. The above transducer analyses the string “dogs” by mapping 

(i) the characters “d”, “o”, and “g” to themselves 
(ii) “s” to the multi-character symbol <Noun>, and 
(iii) the empty string to the symbol <plural>

Figure 6.7: Block diagram for SFST based Morphological Analyser

Lexicon file  
FST Rules 
(Hand Written) 

Morphological Analyser 

Input word in 
Surface Forms 

Output in Lexicon forms 
(Grammatical Structure) 
e.g. - <noun><plural>

Figure 6.7: Block diagram for SFST based Morphological Analyser
A FST based Morphological analyser can be divided into three phases 1) Generation of lexicon file 2) Creation of rule list for inflection and derivational morphology 3) Morphological processor.

- Generation of lexicon file:

Figure 6.8: Generation of Lexicon file

The above figure (Figure 6.8) shows the different steps for generating a lexicon file. The lexicon was generated using the raw corpus collected from different source discussed in section 5.1.1 of chapter 5. Unique words from the corpus are extracted and sorted to make the task of processing of the words manually easier. These words are then manually classified into various classes and according to their inflection, and derivations types. In order to decrease complexity of our program, we put all root words belonging to the same inflection class into a separate file and add the different inflectional endings in the main program. Inflection class files /lexical files are saved with “.lex” extension. Depending on the affix added with the root word,
source corpus is divided into several "lex" file. The total number of base words contained in the classified lexicon files is 59470.

- **Finite State Transducer Rule:**

We write rule file for our FST manually for inflected words (for both inflectional and derivational). It is not a good practice to add the entire lexicon falling under different category and the rules in a same file. So we create separate lexicon file and rule file. In the rule file we processed the entire lexicon file according to their Morphological behaviour. This file is saved with "fst" extension (say "assamese.fst")

![Sample of assamese.fst file](image)

The above figure (Figure 6.9) shows few lines of codes of "assamese.fst" file, where the suffixes used with Verbs are stored under "verb-reg-infl" FST variable name. According to the above codes, if we input the word "অনন্য" to the analyser, it analyses the output as `<V><PastPerfect-1stPerson>`. If we
input only the base form of the word “কবলে” i.e “কব”, the analyser gives output as কব<V><root>

"Noun.lex" $noun-reg-infl$
"Pronoun.lex" $pronoun-reg-infl$
"Verb.lex" $verb-reg-infl$
"Adverb.lex" $adverb-reg-infl$
"Adjective.lex" $adjective-reg-infl$
"Numeral.lex" $numeral-reg-infl$
"Conjunction.lex" $Conjunction-reg-infl$

Figure 6.10: Merging of lexicon entries with suffixes

The above figure (Figure 6.10) shows few lines of codes of “assamese.fst” file, where we are merging the root word files with the suffixes of that category. “Noun.lex” file contains root words falling into category noun and “noun-reg-infl” is a FST variable which contains the suffixes can be added with category noun.

$v1$ = "Verb.lex"
$v1_1$ = $verb-reg-infl$
$v1_1$ = $v1$ $v1_1$
$v1_1$ = (<prefix>) ? $v1_1$
$v1_2$ = (<prefix>) ? $v1_1$
$v1_3$ = (<prefix>) ? $v1_1$
$v1_1$ | $v1_2$ | $v1_2$

Figure 6.11: Merging of lexicon entries with prefix

The above figure (Figure 6.11) shows few lines of codes of “assamese.fst” file, where we are merging prefix entries with the lexicon. In the above code we have added “κ”, “κι” and “κη” prefixes with the verb lexicon entry. So
whenever we try to analyse the word “নকন” (say “কন” is already added in the verb lexicon entry), the analyser result as $<\text{prefix}>কন<\text{verb}>$. The result we can be parsed and display according to our requirement.

| $\%$ন্যঃকন্নী$ \rightarrow$ গান্ধীনী | $S_{N1} = ~<=>> ন (#:<>অ) |
| $\%$ন্যঃকন্নী$ \rightarrow$ নন্দ | $S_{N2} = ~<=>> ন (#:<>অ) |
| $\%$ন্যঃকন্নী$ \rightarrow$ তা | $S_{N3} = ~<=>> ত (#:<>অ) |
| $S_{N4} = S_{N1} \mid S_{N2} \mid S_{N3}$ |

Figure 6.12: Alphabet conversion rules

The above figure (Figure 6.12) shows few lines of codes of “assamese.fst” file, where we have added two-level rule for our analyser to change few alphabets. The rules are added to the analyser according to the conversion rule discussed in section 5.1.2. of chapter 5.

- **Morphological processor:**

  Morphological processor is the main processing unit of an FST based analyser for finding out the lexical form of an inputted surface form of a word. In our work we are using SFST tool to design Morphological processor (Figure 6.13 shows the block diagram).
Two file is required for processing a word in SFST. Lexicon file and rule file. These two required file is already created manually and we input these two file to SFST tool.

- **Compilation of “.fst” file:**

  The “.fst” file (assamese.fst) should be first compile before using it for Morphological Analysis process. Compilation process use the command

  \$fst-compiler  assamese.fst assamese.a

  Here “fst-compiler” command is used to compile the rules and lexicons in to object fie.Since we are using UTF-8 character set for our work, we are using another command to compile our “.fst” file

  \$fst-compiler-utf8  assamese.fst assamese.a

  Here “fst-compiler-utf8” command is used to compile the rules and lexicons in to object fie written in UTF-8 character set. In our example “assamese.fst” file is compiled to “assamese.a” object file. If anything is modified in “.lex” file or rule file, every time we need to execute above command to create a modified object file.
Analysing input word(s):

SFST tool provides two commands for morphological analysis “fst-mor” and “fst-infl”. “fst-mor” command is used for analysing single word in terminal window. “fst-infl” command is used to analyse batch of words. We are using “fst-infl” command, since we need to pass the command from our Java module and execute a process a batch of words.

```
$fst-infl assamese.a write.txt > output.txt
```

In the above example, words to be analysed are stored in “write.txt” file. “fst-infl” command analyse the words from “write.txt” file one by one and finally we will get the analyse output in “output.txt” file. In the analyser we are providing words in its surface form and the analyser provides output in its lexicon form.

6.2.1.1.1. Working principle of FST

![Diagrammatic representation of few Assamese Words](image)

Figure 6.14: Diagrammatic representation of few Assamese Words
The above figure (Figure 6.14) shows the diagrammatic representation of Finite State Transducer of few Assamese words /ma/, /mas/, /mama/, /manuɦ/, /manuɦɔɹ/, /manuɦɔk/ and /manuɦbɔɹ/.

Here 0 is the initial state and the set of final state include the state 2,4,6,8,9,10 and 13. The above FST will accept the words “/manuɦ/” as follows

\[ /\text{manuɦ}/ - 0 + 0 + 0 + 0 + 0 + 0 \]

1. At First FST will match symbol; /m/, since there is a transition from state 0 to state 1 for mapping the symbol /m/ to /m/, FST will pass the state to state 1.

2. FST will try to match Assamese symbol /a/ from state 1 to state 2 and pass to state 2, since there is a mapping of the symbol /a/ to /a/.

3. State 2 is a final state for the Assamese word /maa/, but our FST will not take this state as final state, since our word “/manuɦ/”(/manuɦɔɹ/) is not completely parsed. From state 2, there are 3 transition state 2⇒5, 2⇒4, 2⇒3. FST will move to state 5 since next character to be recognized is /n/ and the mapping of symbol /n/ to /n/ is exist there.

4. The next symbol for the word “/manuɦ/”(/manuɦɔɹ/) is /ə/ and there is a transition from state 5 to state 7, having the mapping of symbol /ə/ to /ə/ So our FST will move to state 7.
5. From state 7 there is a single transition to state 8 with the mapping of symbol / h/ to / h/ and since our next symbol to be parsed for the word “मानूह” (manufi) is / h/, our FST will move to the state 8.

6. State 8 is another final state for the word “ मानूह” (manufi) which is the lemma of the word “मानूह” (manufi) From state 8, there are 3 transition state 8→9, 8→10, 8→11. Our FST will move to state 9, which have a mapping of an empty symbol to the symbol /h/. Thus our FST will give output as “मानूह” (manufi) for the input word “मानूह” (manufi) and the symbol /h/ as a suffix.

6.2.1.1.2. A complete work flow of our Morphological Analyser

Let’s take an example of Assamese Sentence: গকরে যাঁই খাইছে।

– **Java Module 1:** After inputting the sentence, it will be tokenizing into four tokens

1) গকরে 2) যাঁই 3) খাইছে 4) খাইছে

After tokenizing into words, we have find that number 4 token is an end marker for Assamese sentence, so it will be removed from processing list. Then the other three words will be stored in processing list (write.txt file) as one word in a line.

কেকরে

যাঁই

খাইছে
After creating token file, the control will be transferred to the SFST module and module will access the prepared list of words for analysing.

- **SFST Module**: In SFST module, we execute the word list file (say write.txt) with our precompiled rule list (we named as assamese.a) and write the output in a text file (say output.txt)

  $ fst-infl assamese.a write.txt > output.txt

In our rule list if the rule for inputted word is stored, it can be process properly. For our Assamese sentence (পকে খাইছে) rule will be like

পক<N> :<>কে<Nominative Singular Case Marker> : রে

খাঁহ<V>:<Present Continuous Tense Marker>:ইছে

After processing input file (write.txt), the output will be written in output.txt file according to the command we have passed to SFST.

The content of output file will be

>পকরে

পক<N>রে<Nominative Singular Case Marker>

>খাঁহ

খাঁহ<V><Present Continuous Tense Marker>
The token stored in inputted file can be process properly, if the rules for that inputted tokens are stored in transducer. We may get unprocessed words as a result, if a rule for a word is not stored.

For example, if no rules are precompiled for the word “খাইছে”; then output for our analyser will be

>খাইছে

no result for খাইছে

This output file will be again process with Java programming language (Java Module 2) for proper display.

- **Java Module 2**: This module processes the output of SFST module and display in our required format. After processing the output of SFST module it creates an output file (*out_process.txt*) with display format. Followings are the contents of *out_process.txt* file.

```
গকরে = গক<\N> +রে<\Nom\S\C\M\K>

খাও = খাও<\N><\r\O\T>

খাইছে = খা<\V>+ ইছে<\P\r\C\T\M\K>
```

The content of *out_process.txt* file is finally transferred to the web based GUI for displaying as a result of analysis.
Figure 6.15: Input and outputs in GUI