EthioMorph: Technical Implementation of a Rule-Based Ge'ez Morphological Engine

Esubalew Chekol
College of Technology and Built Environment
Addis Ababa University
NLP Course Project

***Abstract*—This paper presents the technical implementation of EthioMorph, a rule-based morphological analysis and generation system for Classical Ge'ez. The system implements all eight canonical verb types: ቀተለ (Type A), ቀደሰ (Type B/Geminate), ባረከ (Type C), ሖመረ (Type C-O), ሴሰየ (Weak-Final), ክህለ (Laryngeal), ማሕረከ (Quadriliteral), and ተንበለ (T-Quad). We describe the core algorithms including Unicode-based vowel arithmetic, radical extraction using consonant skeleton analysis, and template-driven word generation. The system operates on the mathematical principle that Ethiopic Unicode characters follow a predictable 7-order vowel pattern, enabling vowel transformations through simple arithmetic: $\text{char}_{\text{new}} = \text{base} + (\text{order} - 1)$. We detail the implementation of the C1-C2-C3 radical system, suffix fusion rules, laryngeal vowel shift algorithms, and the verb home detection classifier.**

***Index Terms*—Ge'ez, morphological analysis, NLP, Unicode processing, Semitic morphology, rule-based systems**

I. Introduction

EthioMorph is a computational morphology engine for Classical Ge'ez (ግእዝ) that implements linguistic rules as algorithms. Unlike statistical or neural approaches, EthioMorph uses explicit mathematical transformations based on the structure of the Ethiopic Unicode block and the grammatical rules of Ge'ez morphology.

This document explains **how** the system works—the actual code logic, data structures, and algorithms that drive morphological processing.

A. Core Problem

Ge'ez, like other Semitic languages, forms words by combining:

- **Root**: A sequence of consonants (typically 3) carrying lexical meaning
- **Pattern**: A template of vowels and affixes encoding grammatical information

For example, the root **q-t-l** (**ቀተለ**, "kill") combines with different patterns:

$$\text{q-t-l} + [\text{1-1-1}] \rightarrow \textbf{ቀተለ} \text{ (Perfective: "he killed")}$$
$$\text{q-t-l} + [\text{yi-1-6-6}] \rightarrow \textbf{ይቅትል} \text{ (Imperfective: "he kills")}$$

The challenge is implementing this bidirectionally:

1) **Analysis**: **ይቀትሉ** → root: **ቀተለ**, tense: imperfective, person: 3rd plural masculine
2) **Generation**: (**ቀተለ**, imperfective, 3PM) → **ይቀትሉ**

II. Unicode Mathematics for Ethiopic

A. The Ethiopic Block Structure

The Ethiopic Unicode block (U+1200–U+137F) organizes characters in a systematic pattern. Each consonant has 7 forms representing vowel variations:

TABLE I
Unicode Pattern for Consonant ቀ (qäf)

Order	Unicode	Character	Vowel
1	U+1240	ቀ	ä (schwa)
2	U+1241	ቁ	u
3	U+1242	ቂ	i
4	U+1243	ቃ	a
5	U+1244	ቄ	e
6	U+1245	ቅ	(none)
7	U+1246	ቆ	o

B. Vowel Arithmetic

Given any Ethiopic character, we can compute its base consonant and vowel order:

$$\text{order}(c) = ((c - \text{base\_offset}) \mod 8) + 1 \tag{1}$$

$$\text{base}(c) = c - (\text{order}(c) - 1) \tag{2}$$

To change a character's vowel:

$$\text{revowelize}(\text{base}, \text{order}) = \text{base} + (\text{order} - 1) \tag{3}$$

Implementation:

```
def get_vowel_order(char):
    """Extract␣vowel␣order␣(1-7)␣from␣
        character."""
    return ORDER_MAP.get(char, 1)

def devowelize(char):
    """Return␣base␣consonant␣(1st␣order␣
        form)."""
    return DEVOWELIZATION_MAP.get(char,
        char)

def get_char_by_order(base, order):
    """Apply␣vowel␣order␣to␣base␣consonant.
        """
    return REVOWELIZATION_MAP.get((base,
        order), base)
```

Example transformation:

$$\text{devowelize}(\textbf{ቃ}) = \textbf{ቀ}$$
$$\text{get\_vowel\_order}(\textbf{ቃ}) = 4$$
$$\text{get\_char\_by\_order}(\textbf{ቀ}, 6) = \textbf{ቅ}$$

III. The C1-C2-C3 Radical System

A. Radical Representation

Ge'ez verbs use a consonantal skeleton notation:

- **C1**: First radical (consonant 1)
- **C2**: Second radical
- **C3**: Third radical
- **C4**: Fourth radical (quadriliterals only)

For **ቀተለ**: C1=**ቀ**, C2=**ተ**, C3=**ለ**

B. Template Definition

Each conjugation is defined by a **vowel map** specifying the order for each radical:

```
# Perfective 3rd singular masculine for Type A
template = {
    "vowel_map": {"C1": 1, "C2": 1, "C3": 1},
    "prefix": "",
    "suffix": ""
}
# Result: C1(1st) + C2(1st) + C3(1st) = qä-tä-lä
```

```
# Imperfective 3rd singular masculine
template = {
    "vowel_map": {"C1": 1, "C2": 6, "C3": 6},
    "prefix": "yi",
    "suffix": ""
}
# Result: yi + C1(1st) + C2(6th) + C3(6th) = yi-qä-t-l
```

C. The Generation Algorithm

Algorithm 1 Word Generation from Root

Require: root (string), tense, subject, verb_type
Ensure: conjugated word (string)

```
template ← TEMPLATES[verb_type][tense][subject]
vowel_map ← template["vowel_map"]
radicals ← extract_radicals(root)
result ← template["prefix"]
for each position in ["C1", "C2", "C3"] do
    base ← devowelize(radicals[position])
    order ← vowel_map[position]
    result ← result + get_char_by_order(base, order)
end for
result ← result + template["suffix"]
result ← apply_suffix_fusion(result)
return result
```

IV. Suffix Fusion Rules

A. The Collision Problem

When a vowel suffix attaches to a consonant-only (6th order) ending, they must fuse:

$$\text{C}_{\text{6th}} + \text{V}_{\text{suffix}} \rightarrow \text{C}_{\text{order}(V)} \tag{4}$$

Example: **ቅትል** + **ኡ** ("they") ≠ **ቅትልኡ**
Instead: **ቅትል** + **ኡ** → **ቅትሉ**

B. Fusion Algorithm

```
def apply_suffix_fusion(stem, suffix):
    """Fuse 6th-order final + vowel suffix."""
    if not suffix:
        return stem

    last_char = stem[-1]
    last_order = get_vowel_order(last_char)

    # Check if stem ends in 6th order (consonant)
    if last_order != 6:
        return stem + suffix

    # Map suffix to target order
    suffix_vowel_map = {
        '\u12A1': 2,  # u-vowel suffix
        '\u12A0': 4,  # a-vowel suffix
        '\u12A5': 5,  # e-vowel suffix
    }

    first_suffix_char = suffix[0]
    if first_suffix_char in suffix_vowel_map:
        target_order = suffix_vowel_map[first_suffix_char]
        base = devowelize(last_char)
        fused = get_char_by_order(base, target_order)
        return stem[:-1] + fused + suffix[1:]

    return stem + suffix
```

V. Verb Home Detection Algorithm

A. The Eight Canonical Verb Types

In Classical Ge'ez grammar, verbs are classified into "homes" (**ቤት**) named after their canonical representative. EthioMorph implements all eight standard verb types:

B. Classification Rules

The verb "home" determines which conjugation templates to use. Detection is based on:

1) **Radical count**: 3 (triradical) vs 4 (quadriliteral)
2) **C1 vowel order**: The vowel of the first radical in citation form
3) **Consonant quality**: Laryngeals (**ሀ, ሐ, ኀ, አ, ዐ**) or weak (**ወ, የ**)
4) **Gemination pattern**: Whether C2 doubles (Type B)

TABLE II
The Eight Canonical Ge'ez Verb Types (Homes)

#	Type Head	Name	Description
1	**ቀተለ**	Type A	Strong triradical (C1 = 1st order)
2	**ቀደሰ**	Type B	Geminate (C2 doubles in conjugation)
3	**ባረከ**	Type C	Long vowel (C1 = 4th order **ራብዕ**)
4	**ጦመረ**	Type C-O	O-initial (C1 = 7th order **ሳብዕ**)
5	**ሴሰየ**	Weak-Final	Final radical is **የ** (weak)
6	**ክህለ**	Laryngeal	Contains laryngeal (**ህ**) radical
7	**ማሕረከ**	Quadriliteral	Four consonant radicals
8	**ተንበለ**	T-Quad	Quadriliteral with **ተ-** prefix

TABLE III
Algorithmic Classification Rules

Condition	Type	Example
$\|\text{radicals}\| \geq 4$	Quadriliteral	**ማሕረከ**, **ተንበለ**
C1 order = 7	Type C-O	**ጦመረ**
C1 order = 4	Type C	**ባረከ**
C3 $\in$ {weak}	Weak-Final	**ሴሰየ**
C2 $\in$ {laryngeal}	Laryngeal	**ክህለ**
C2 geminated	Type B	**ቀደሰ**
Default	Type A	**ቀተለ**

C. Detection Algorithm

D. Feature Detection

Additional features modify conjugation behavior:

```
LARYNGEALS = {'h', 'H', 'x', 'a', 'A'}  # Gutturals
WEAK_CONSONANTS = {'w', 'y'}

def detect_features(root, radicals):
    features = {}

    # Check for laryngeal consonants
    for rad in radicals:
        base = devowelize(rad)
        if base in LARYNGEAL_BASES:
            features['has_laryngeal'] = True
            break

    # Check for hollow verb (weak C2)
    if len(radicals) >= 2:
        c2_base = devowelize(radicals[1])
        if c2_base in WEAK_BASES:
            features['is_hollow'] = True

    # Check for weak initial
    c1_base = devowelize(radicals[0])
    if c1_base in WEAK_BASES:
        features['weak_initial'] = True

    return features
```

VI. Laryngeal Vowel Shift Rules

A. The Laryngeal Problem

Laryngeal consonants (**ሀ, ሐ, ኀ, አ, ዐ**) cannot take certain vowels and trigger shifts:

Algorithm 2 Verb Home Detection (All 8 Types)

Require: root (string)
Ensure: (type, features)
radicals ← extract_consonants(root)
features ← {}
// Check radical count first
if len(radicals) ≥ 4 **then**
  if root starts with **ተ** **then**
    return ("type_tanbala", features) {**ተንበለ**}
  else
    return ("type_mahraka", features) {**ማሕረከ**}
  end if
end if
// Check C1 vowel order
c1_order ← get_vowel_order(root[0])
if c1_order == 7 **then**
  return ("type_c_o", features) {**ጦመረ**}
else if c1_order == 4 **then**
  return ("type_c", features) {**ባረከ**}
end if
// Check for weak final radical
if devowelize(root[2]) ∈ WEAK_CONSONANTS **then**
  features["weak_final"] ← True
  return ("type_sesaya", features) {**ሴሰየ**}
end if
// Check for laryngeal radicals
for each radical in radicals **do**
  if devowelize(radical) ∈ LARYNGEALS **then**
    features["has_laryngeal"] ← True
    return ("type_kahla", features) {**ክህለ**}
  end if
end for
// Check for gemination (Type B)
if is_geminate_pattern(root) **then**
  return ("type_b", features) {**ቀደሰ**}
end if
// Default: Type A
return ("type_a", features) {**ቀተለ**}

$$\text{Laryngeal} + \text{1st order} \rightarrow \text{Laryngeal} + \text{4th order} \quad (5)$$

B. Implementation

```
def apply_laryngeal_rules(char, position, features):
    """Shift␣vowels␣near␣laryngeal␣consonants."""
    if not features.get('has_laryngeal'):
        return char

    base = devowelize(char)
    order = get_vowel_order(char)

    # Laryngeals prefer 4th order in jussive
    if is_laryngeal(base) and order == 1:
```

```
        return get_char_by_order(base, 4)

    return char
```

VII. Stemmer: The Analysis Pipeline

The Stemmer reverses word formation to extract roots and grammatical information.

A. Processing Pipeline

1) **Normalize**: Map homophonous characters to canonical forms
2) **Strip Prefixes**: Identify and remove grammatical prefixes
3) **Strip Suffixes**: Identify and remove grammatical suffixes
4) **Extract Skeleton**: Get consonant-only representation
5) **Reconstruct Root**: Handle weak consonant restoration
6) **Identify Pattern**: Determine tense/person/stem
7) **Detect Home**: Classify verb type

B. Prefix Priority

Prefixes must be checked longest-first to avoid partial matches:

```
PREFIXES = [
    # Stem IV (Causative-Passive) - check first
    'yaste', 'taste', 'naste', 'laste',
    # Stem III (Causative)
    'yas', 'tas', 'nas', 'las',
    'ya', 'ta', 'na', 'a',
    # Stem II (Passive)
    'yit', 'tit', 'te',
    # Stem I (Basic)
    'yi', 'ti', 'ni', 'li'
]
# Sorted by length descending
PREFIXES.sort(key=len, reverse=True)
```

C. Pattern Identification

```
def identify_verb_pattern(stem, stripped_prefixes):
    """Determine grammatical pattern from affixes."""

    # Check stem markers in priority order
    if 'aste' in stripped_prefixes:
        return {'stem': 4, 'name': 'Causative-Passive'}

    if 'as' in stripped_prefixes or has_causative_vowel(stem):
        return {'stem': 3, 'name': 'Causative'}

    if 'te' in stripped_prefixes or 'yit' in stripped_prefixes:
        return {'stem': 2, 'name': 'Passive'}

    return {'stem': 1, 'name': 'Basic'}
```

VIII. Data Structures

A. Templates JSON

```
{
  "type_a": {
    "perfective": {
      "3sm": {
        "vowel_map": {"C1": 1, "C2": 1, "C3": 1},
        "prefix": "",
        "suffix": ""
      },
      "3sf": {
        "vowel_map": {"C1": 1, "C2": 1, "C3": 4},
        "prefix": "",
        "suffix": "t"
      }
      // ... 10 person forms
    },
    "imperfective": {
      "3sm": {
        "vowel_map": {"C1": 1, "C2": 6, "C3": 6},
        "prefix": "yi",
        "suffix": ""
      }
      // ...
    }
    // jussive, imperative, derived forms
  }
}
```

B. Lookup Maps

Three pre-computed maps enable O(1) character transformations:

```
# Character -> Base consonant
DEVOWELIZATION_MAP = {
    '\u1240': '\u1240',  # qe -> qe (already base)
    '\u1241': '\u1240',  # qu -> qe
    '\u1242': '\u1240',  # qi -> qe
    # ... all 300+ characters
}

# Character -> Vowel order (1-7)
ORDER_MAP = {
    '\u1240': 1,  # qe = 1st order
    '\u1241': 2,  # qu = 2nd order
    # ...
}

# (Base, Order) -> Character
REVOWELIZATION_MAP = {
    ('\u1240', 1): '\u1240',
    ('\u1240', 2): '\u1241',
    ('\u1240', 3): '\u1242',
    # ...
}
```

IX. System Architecture

X. API Usage

A. Analysis Endpoint

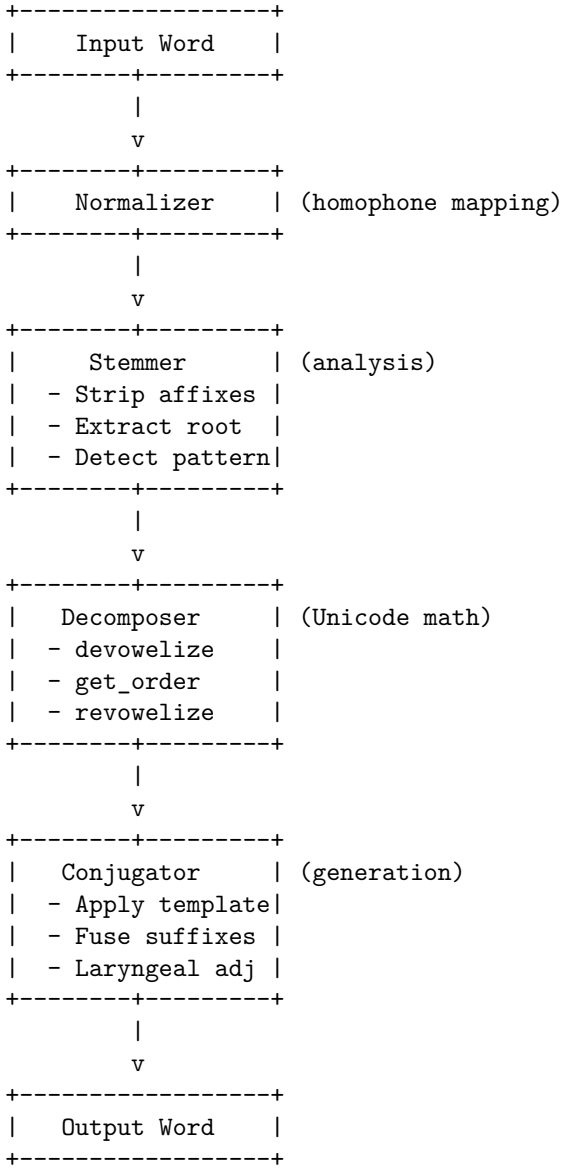

Fig. 1. EthioMorph Processing Pipeline

```
POST /api/xray
{
  "word": "yiqatlu"
}

Response:
{
  "root": "qatala",
  "pattern": "imperfective",
  "stem": 1,
  "person": "3pm",
  "verb_type": "type_a"
}
```

B. Generation Endpoint

```
POST /api/morph
{
  "root": "qatala"
}

Response:
{
  "perfective": {
    "3sm": "qatala",
    "3sf": "qatalat",
    ...
  },
  "imperfective": {...},
  "jussive": {...}
}
```

XI. Conclusion

EthioMorph demonstrates that Ge'ez morphology can be effectively implemented through:

1) **Unicode arithmetic**: Exploiting the systematic Ethiopic block structure for O(1) vowel transformations
2) **Template-driven generation**: Encoding linguistic rules as data rather than code
3) **Algorithmic classification**: Detecting verb types from formal properties rather than lexicon lookup
4) **Bidirectional processing**: Using the same core rules for both analysis and generation

The source code is available at: `github.com/esubalew/EthioMorph`