Finite-State Non-Concatenative Morphotactics

Kenneth R. Beesley and Lauri Karttunen
Xerox Research Centre Europe
Grenoble Laboratory
6, chemin de Maupertuis
38240 MEYLAN France
beesley@xrce.xerox.com, karttunen@xrce.xerox.com

Abstract

We describe a new technique for constructing finite-state transducers that involves reapplying the regular-expression compiler to its own output. Implemented in an algorithm called compile-replace, this technique has proved useful for handling non-concatenative phenomena; and we demonstrate it on Malay full-stem reduplication and Arabic stem interdigitation.

1 Introduction

Most natural languages construct words by concatenating morphemes together in strict orders. Such concatenative morphotactics can be impressively productive, especially in agglutinative languages like Aymara or Turkish, and in agglutinative/polysynthetic languages like Inuktitut. In such languages a single word may contain as many morphemes as an average-length English sentence.[1]

Finite-state morphology in the tradition of the Two-Level (Koskenniemi, 1983) and Xerox implementations (Beesley and Karttunen, 2000) has been very successful in implementing large-scale, robust and efficient morphological analyzer-generators for concatenative languages, including the commercially important European languages and non-Indo-European examples like Finnish, Turkish and Hungarian. However, Koskenniemi himself understood that his initial implementation had significant limitations in handling non-concatenative morphotactic processes:

> "Only restricted infixation and reduplication can be handled adequately with the present system. Some extensions or revisions will be necessary for an adequate description of languages possessing extensive infixation or reduplication" (Koskenniemi, 1983, 27).

This limitation has of course not escaped the notice of various reviewers, e.g. Sproat(1992). We shall argue that the morphotactic limitations of the traditional implementations are the direct result of relying solely on the concatenation operation in morphotactic description.

We describe a technique, within the Xerox implementation of finite-state morphology, that corrects the limitations at the source, going beyond concatenation to allow the full range of finite-state operations to be used in morphotactic description. Regular-expression descriptions are compiled into finite-state automata or transducers (collectively called networks) as usual, and then the compiler is re-applied to its own output, producing a modified but still finite-state network. This technique, implemented in an algorithm called COMPILE-REPLACE, has already proved useful for handling Malay full-stem reduplication and Arabic stem interdigitation, which will be described below. Before illustrating these applications, we will first outline our general approach to finite-state morphology.

2 Finite-State Morphology

2.1 Analysis and Generation

In the most theory- and implementation-neutral form, morphological analysis/generation of written words can be modeled as a relation between the words themselves and analyses of those words. The basic claim or hope of the finite-state approach to natural-language morphology is that the mapping from words to their analyses (and vice versa) constitutes a regular relation, i.e. a relation that can be represented by a finite-state transducer. The language to be analyzed consists of strings (= words = sequences of symbols) writ-

[1] Aymara *utamankapxarakiwa* = "also they are in your house;" Inuktitut: *Parimunngaujumaniralauqsimanngittunga* = "I never said I wanted to go to Paris."

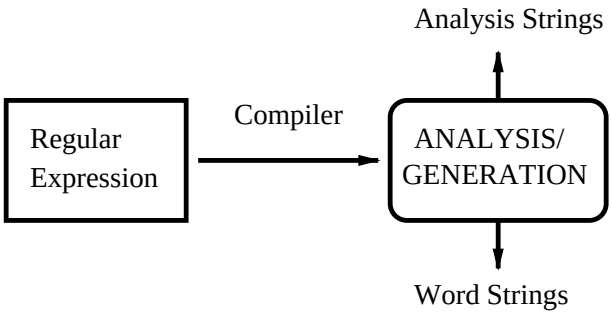

Figure 1: Compilation of a Regular Expression into an FST that Maps between Two Regular Languages

ten according to some defined orthography. In a commercial application for a given natural language, the language to be analyzed is usually a given, e.g. the set of valid French words as written according to standard French orthography. The analysis language again consists of strings, but strings designed according to the needs and taste of the linguist, representing analyses of the orthographical words. It is sometimes convenient to design these analysis strings to show all the constituent morphemes in their morphophonemic form, separated and identified. In other applications, it may be useful to design the analysis strings to contain the traditional dictionary citation form, together with linguist-selected "tag" symbols like `+Noun`, `+Verb`, `+SG`, `+PL`, that convey category, person, number, tense, mood, case, etc. Thus the analysis string representing the first-person singular, present indicative form of the French verb *payer* ("to pay") might be spelled `payer+IndP+SG+P1+Verb`.

If the relation is finite-state, then it can be defined using the metalanguage of regular expressions; and, with a suitable compiler, the regular-expression source code can be compiled into a finite-state transducer (FST), as shown in Figure 1, that implements the relation computationally. Following convention, we will often refer to the upper projection of the FST, representing analyses, as the LEXICAL language, a set of lexical strings; and we will refer to the lower projection as the SURFACE language, consisting of surface strings. There are compelling advantages to computing with such finite-state machines, including mathematical elegance, flexibility, and for most natural-language applications, high efficiency and data-compaction.

One computes with FSTs by applying them, in either direction, to an input string. When one such FST that was written for French is applied in an upward direction to the surface word *maisons* ("houses"), it returns the related lexical string `maison+Fem+PL+Noun`, consisting of the citation form and tag symbols chosen by a linguist to convey that the surface word is a feminine noun in the plural form. A single surface string can be related to multiple lexical strings, e.g. applying this FST in an upward direction to the surface string *suis* produces the four related lexical strings shown in Figure 2. Such ambiguity of surface strings is very common.

```
être+IndP+SG+P1+Verb
suivre+IndP+SG+P2+Verb
suivre+IndP+SG+P1+Verb
suivre+Imp+SG+P2+Verb
```

Figure 2: Multiple Analyses for *suis*

Conversely, the very same FST can be applied in a downward direction to a lexical string like `être+IndP+SG+P1+Verb` to return the related surface string *suis*; such transducers are inherently bidirectional. Ambiguity in the downward direction is also possible, as in the relation of the lexical string `payer+IndP+SG+P1+Verb` ("I pay") to the surface strings *paie* and *paye*, which are in fact valid alternate spellings in standard French orthography.

2.2 Morphotactics and Alternations

There are two challenges in modeling natural language morphology:

- Morphotactics
- Phonological/Orthographical Alternations

Finite-state morphology models both using regular expressions. The source descriptions may also be written in higher-level notations (Beesley and Karttunen, 2000) that are simply helpful shorthands for regular expressions and that compile, using their dedicated compilers, into finite-state networks. In practice, the most commonly separated modules are a lexicon FST, containing lexical strings, and a separately written set of rule FSTs that map from the strings in the lexicon to properly spelled surface strings. The lexicon description defines the morphotactics of the language,

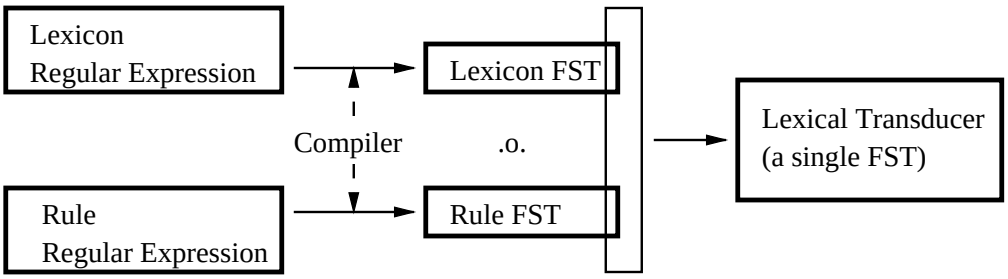

Figure 3: Creation of a Lexical Transducer

and the rules define the alternations. The separately compiled lexicon and rule FSTs can subsequently be composed together as in Figure 3 to form a single LEXICAL TRANSDUCER (Karttunen et al., 1992) that could have been defined equivalently, but perhaps less perspicuously and less efficiently, with a single regular expression.

For example, the information that the comparative of the adjective *big* is *bigger* might be represented in the English lexical transducer by the path (= sequence of states and arcs) in Figure 4, where the zeros represent epsilon symbols.[2] The

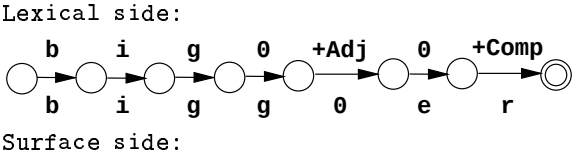

Figure 4: A Path in a Transducer for English

gemination of `g` and the epenthetical `e` in the surface form *bigger* result from the composition of the original lexicon FST with the rule FST representing the regular morphological alternations in English.

For the sake of clarity, Figure 4 represents the upper (= lexical) and the lower (= surface) side of the arc label separately on the opposite sides of the arc. In the remaining diagrams, we use a more compact notation: the upper and the lower symbol are combined into a single label of the form `upper:lower` if the symbols are distinct. A single symbol is used for an identity pair. In the standard notation, the path in Figure 4 is labeled as

`b i g 0:g +Adj:0 0:e +Comp:r`.

Lexical transducers are more efficient for analysis and generation than the classical two-level systems (Koskenniemi, 1983) because the morphotactics and the morphological alternations have been precompiled and need not be consulted at runtime.

[2]The epsilon symbols and their placement in the string are not significant. We will ignore them whenever it is convenient.

Most languages build words by simply stringing morphemes (prefixes, roots and suffixes) together in strict orders. The morphotactic (word-building) processes of prefixation and suffixation can be straightforwardly modeled in finite state terms as concatenation. But some natural languages also exhibit non-concatenative morphotactics. Sometimes the languages themselves are called "non-concatenative languages", but most employ significant concatenation as well, so the term "not completely concatenative" is usually more appropriate.

In Arabic, for example, prefixes and suffixes attach to stems in the usual concatenative way, but stems themselves are formed by a process known informally as interdigitation; while in Malay, noun plurals are formed by a process known as full-stem reduplication. Although Arabic and Malay also include prefixation and suffixation that are modeled straightforwardly by concatenation, a complete lexicon cannot be obtained without non-concatenative processes.

We will proceed with descriptions of how Malay reduplication and Semitic stem interdigitation are handled in finite-state morphology using the new compile-replace algorithm.

3 Compile-Replace

The central idea in our approach to the modeling of non-concatenative processes is to define networks using regular expressions, as before; but we now define the strings of an intermediate network so that they contain appropriate substrings that are themselves in the format of regular expressions. The compile-replace algorithm then reapplies the regular-expression compiler to its own output, compiling the regular-expression substrings in the intermediate network and replacing them with the result of the compilation.

To take a simple non-linguistic example, Figure 5 represents a network that maps the regular expression `a*` into `^[a*^]`; that is, the same expression enclosed between two special delimiters,

`^[` and `^]`, that mark it as a regular-expression substring.

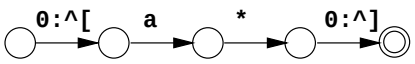

Figure 5: A Network with a Regular-Expression Substring

The application of the compile-replace algorithm on the lower side of the network eliminates the markers, compiles the regular expression, and maps the upper side of the path to the language resulting from the compilation. The network created by the operation is shown in Figure 6.

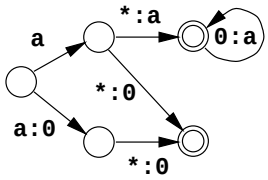

Figure 6: After the Application of Compile-Replace

When applied in the "upward" direction, the transducer in Figure 6 maps any string of the infinite `a*` language into the regular expression from which the language was compiled.

The compile-replace algorithm is essentially a variant of a simple recursive-descent copying routine. It expects to find marked regular-expression substrings on the designated side (upper or lower) of the network. Until an opening limiter `^[` is encountered, the algorithm constructs a copy of the path it is following. If the network contains no regular-expression substrings, the result will be a copy of the original network. When a `^[` is encountered, the algorithm looks for a closing `^]` and extracts the path between the markers to be handled in a special way:

1. The symbols along the indicated side of the path are concatenated into a string and eliminated from the path leaving just the symbols on the opposite side.
2. A separate network is created that contains the modified path.
3. The extracted string is compiled into a second network with the standard regular-expression compiler.
4. The two networks are combined into a single one using the crossproduct operation.
5. The result is spliced between the states representing the origin and the destination of the regular-expression path.

After the special treatment of the regular-expression path is finished, normal processing is resumed in the destination state of the closing `^]` arc.

For example, the result shown in Figure 6 represents the crossproduct of the two networks shown in Figure 7.

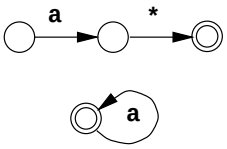

Figure 7: Networks Illustrating Steps 2 and 3 of the Compile-Replace Algorithm

In this simple example, the upper language of the original network in Figure 5 is identical to the regular expression that is compiled and replaced. In the linguistic applications presented in the next sections, the two sides of a regular-expression path contain different strings. The upper side contains morphological information; the regular-expression operators appear only on the lower side and are not present in the final result.

3.1 Reduplication

Traditional Two-Level implementations are already capable of describing some limited reduplication and infixation as in Tagalog (Antworth, 1990, 156–162). The more challenging phenomenon is variable-length reduplication, as found in Malay and the closely related Indonesian language.

An example of variable-length full-stem reduplication occurs with the Malay stem *bagi*, which means "bag" or "suitcase"; this form is in fact number-neutral and can translate as the plural. Its overt plural is phonologically *bagibagi*, formed by repeating the stem twice in a row. Although this pluralization process may appear concatenative, it does not involve concatenating a predictable pluralizing morpheme, but rather copying the preceding stem, whatever it may be and however long it may be. Thus the overt plural of *pelabuhan* ("port"), itself a derived form, is phonologically *pelabuhanpelabuhan*.

Productive reduplication cannot be described by finite-state or even context-free formalisms. It is well known that the copy language, $\{ww \mid w \in L\}$, where each word contains two copies of the same string, is a context-sensitive language. However, if the "base" language L is finite, we can of course construct a finite-state network that encodes L and the reduplications of all the strings

```
Lexical:      b a g i   +Noun +Plural
Surface: ^[ { b a g i }         ^ 2   ^]

Lexical:      p e l a b u h a n   +Noun +Plural
Surface: ^[ { p e l a b u h a n }         ^ 2   ^]
```

Figure 8: Two Paths in the Initial Malay Transducer Defined via Concatenation

```
Lexical:      b a g i         +Noun +Plural
Surface:      b a g i b a g i

Lexical:      p e l a b u h a n                    +Noun +Plural
Surface:      p e l a b u h a n p e l a b u h a n
```

Figure 9: The Malay FST After the Application of Compile-Replace to the Lower-Side Language

in L. We will show a simple and elegant way to do this using strictly finite-state operations.

To understand the solution to full-stem reduplication using the compile-replace algorithm requires a bit of background. In the Xerox regular-expression calculus there are several operators that involve concatenation. For example, if `A` is a regular expression denoting a language or a relation, `A*` denotes zero or more and `A+` denotes one or more concatenations of `A` with itself. There are also operators that express a fixed number of concatenations. Expressions of the form `A^n`, where `n` is an integer, denote n concatenations of `A`. `{abc}` denotes the concatenation of symbols `a`, `b`, and `c`. We employ `^[` and `^]` as delimiter symbols around regular-expression substrings.

The reduplication of any string w can then be notated as `{w}^2`, and we start by defining a network where the lower-side strings are built by simple concatenation of a prefix `^[`, a root enclosed in braces, and an overt-plural suffix `^2` followed by the closing `^]`. Figure 8 shows the paths for two Malay plurals in the initial network.

The compile-replace algorithm, applied to the lower side of this network, recognizes each individual delimited regular-expression substring like `^[{bagi}^2^]`, compiles it, and replaces it with the result of the compilation, here *bagibagi*. The same process applies to the entire lower-side language, resulting in a network that relates pairs of strings such as the ones in Figure 9. This provides the desired solution, still finite-state, for analyzing and generating full-stem reduplication in Malay.[3]

[3]It is well-known (McCarthy and Prince, 1995) that reduplication can be a more complex phenomenon than it is in Malay. In some languages only a part of the stem is reduplicated, and there may be systematic differences between the reduplicate and the base form. We believe that our approach to reduplication can account for these complex phenomena as well, but we cannot discuss the issue here due to lack of space.

The special delimiters `^[` and `^]` can be used to surround any appropriate regular-expression substring, using any necessary regular-expression operators, and compile-replace may be applied to the lower-side and/or upper-side of the network as desired. There is nothing to stop the linguist from inserting delimiters multiple times, including via composition, and reapplying compile-replace multiple times. The technique implemented in compile-replace is a general way of allowing the regular-expression compiler to reapply to and modify its own output.

3.2 Semitic Stem Interdigitation

3.2.1 Review of Earlier Work

Much of the work in non-concatenative finite-state morphotactics has been dedicated to handling Semitic stem interdigitation. An example of interdigitation occurs with the Arabic stem `katab`, which means "wrote". According to an influential autosegmental analysis (McCarthy, 1981), this stem consists of an all-consonant root `ktb` whose general meaning has to do with writing, an abstract consonant-vowel template `CVCVC`, and a voweling or vocalization that he symbolized simply as `a`, signifying perfect aspect and active voice. The root consonants are associated with the `C` slots of the template and the vowel or vowels with the `V` slots, producing a complete stem *katab*. If the root and the vocalization are thought of as morphemes, neither morpheme occurs continuously in the stem. The same root `ktb` can combine with the template `CVCVC` and a different vocalization `ui`, signifying perfect aspect and passive voice, producing the stem *kutib*, which means "was written". Similarly, the root `ktb` can combine with template `CVVCVC` and `ui` to produce *kuutib*, the root `drs` can combine with `CVCVC` and `ui` to form *duris*, and so forth.

Kay (1987) reformalized the autosegmental tiers of McCarthy (1981) as projections of a

multi-level transducer and wrote a small Prolog-based prototype that handled the interdigitation of roots, CV-templates and vocalizations into abstract Arabic stems. This general approach, with multi-tape transducers, has been explored and extended by Kiraz in several papers, see Kiraz (2000) for a summary and further references.

In work more directly related to the current solution, it was Kataja and Koskenniemi (1988) who first demonstrated that Semitic (Akkadian) roots and patterns could be formalized as regular languages, and that the non-concatenative interdigitation of stems could be elegantly formalized as the intersection of those regular languages.

This was the key insight: morphotactic description could employ various finite-state operations, not just concatenation; and languages that required only concatenation were just special cases. By extension, the widely noticed limitations of early finite-state implementations in dealing with non-concatenative morphotactics could be traced to their dependence on the concatenation operation in morphotactic descriptions.

This insight of Kataja and Koskenniemi was applied by Beesley in a large-scale morphological analyzer for Arabic, first using an implementation that simulated the intersection of stems in code at runtime (Beesley, 1991), and ran rather slowly; and later, using Xerox finite-state technology (Beesley, 1996), a new implementation that intersected the stems at compile time and performed well at runtime. The 1996 algorithm that intersected roots and patterns into stems, and substituted the original roots and patterns on just the lower side with the intersected stem, took two hours to handle about 90,000 stems on a SUN Ultra workstation. The compile-replace algorithm is a vast improvement in both generality and efficiency, producing the same result in a few minutes.

Following the lines of Kataja and Koskenniemi (1988), we could define intermediate networks with regular-expression substrings that indicate the intersection of suitably encoded roots, templates, and vocalizations. However, because the interdigitation of stems represents a special case of intersection we compute it using a specialized, more efficient, finite-state algorithm called MERGE.

3.2.2 Merge

The merge algorithm is a pattern-filling operation that combines two regular languages, a template and a filler, into a single one. The strings of the filler language consist of ordinary symbols such as `d`, `r`, `s`, `u`, `i`. The template expressions may contain special class symbols such as `C` (= consonant) or `V` (= vowel) that represent a predefined set of ordinary symbols. The objective of the merge operation is to align the template strings with the filler strings and to instantiate the class symbols of the template as the matching filler symbols.

Like intersection, the merge algorithm operates by following two paths, one in the template network, the other in the filler network, and it constructs the corresponding single path in the result network. Every state in the result corresponds to two original states, one in the template, the other in the filler. If the original states are both final, the resulting state is also final; otherwise it is non-final.

The operation starts in the initial state of the original networks. At each point, the algorithm tries to find all the successful matches between the template arcs and filler arcs. A match is successful if the filler arc symbol is included in the class designated by the template arc symbol. The main difference between merge and classical intersection is in Conditions 1 and 2 below:

1. If a successful match is found, a new arc is added to the current result state. The arc is labeled with the filler arc symbol; its destination is the result state that corresponds to the two original destinations.

2. If no successful match is found for a given template arc, the arc is copied into the current result state. Its destination is the result state that corresponds to the destination of the template arc and the current filler state.

In effect, Condition 2 preserves any template arc that does not find a match. In that case, the path in the template network advances to a new state while the path in the filler network stays at the current state.

We use the networks in Figure 10 to illustrate the effect of the merge algorithm. Figure 10 shows a linear template network and two filler networks, one of which is cyclic.

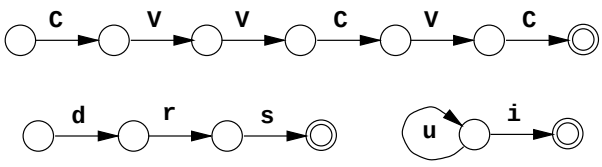

Figure 10: A Template Network and Two Filler Networks

It is easy to see that the merge of the `drs` network with the template network yields the result shown in Figure 11. The three symbols of the filler

string are instantiated in the three consonant slots in the CVVCVC template.

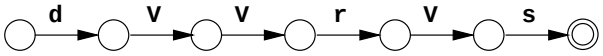

Figure 11: Intermediate Result.

Figure 12 presents the final result in which the second filler network in Figure 10 is merged with the intermediate result shown in Figure 11.

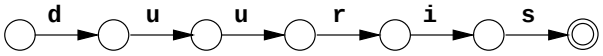

Figure 12: Final Result

In this case, the filler language contains an infinite set of strings, but only one successful path can be constructed. Because the filler string ends with a single i, the first two V symbols can be instantiated only as u. Note that ordinary symbols in the partially filled template are treated like the class symbols that do not find a match. That is, they are copied into the result in their current position without consuming a filler symbol.

To introduce the merge operation into the Xerox regular-expression calculus we need to choose an operator symbol. Because merge, like subtraction, is a non-commutative operation, we also must distinguish between the template and the filler. For convenience, we introduce two variants of the merge operator, .<m. and .m>., that differ only with respect to whether the template is to the left (.<m.) or to the right (.m>.) of the filler. The expression [A .<m. B] represents the same merge operation as [B .m>. A]. In both cases, A denotes the template, B denotes the filler, and the result is the same. With these new operators, the network in Figure 12 can be compiled from an expression such as

```
d r s .m>. C V V C V C .<m. u* i
```

3.2.3 Merging Roots and Vocalizations with Templates

Following the tradition, we can represent the lexical forms of Arabic stems as consisting of three components, a consonantal root, a CV template and a vocalization, possibly preceded and followed by additional affixes. In contrast to McCarthy, Kay, and Kiraz, we combine the three components into a single projection. In a sense, McCarthy's three tiers are conflated into a single one with three distinct parts. In our opinion, there is no substantive difference from a computational point of view.

For example, the initial lexical representation of the surface forms *katab* and *duuris* may be represented as a concatenation of the three components shown in Figure 13. We use the symbols =Root, =Template, and =Voc to designate the three components of the lexical form. The corresponding initial surface forms are regular-expression substrings of the type we have just discussed. They contain two merge operators that indicate that the root consonants and the vocalism should be merged into the template.

The application of the compile-replace operation to the initial lexicon yields a transducer that maps the Arabic interdigitated forms directly into their corresponding tripartite analyses and vice versa, as illustrated in Figure 14. Alternation rules are subsequently composed on the lower side of the result to map the interdigitated, but still morphophonemic, strings into real surface strings.

4 Status of the Implementations

4.1 Malay Morphological Analyzer/Generator

Malay and Indonesian are closely-related languages characterized by rich derivation and little or nothing that could be called inflection. The Malay morphological analyzer prototype, written using **lexc**, Replace Rules, and compile-replace, implements approximately 50 different derivational processes, including prefixation, suffixation, prefix-suffix pairs (circumfixation), reduplication, some infixation, and combinations of these processes. Each root is marked manually in the source dictionary to indicate the idiosyncratic subset of derivational processes that it undergoes.

The small prototype dictionary, stored in an XML format, contains approximately 1000 roots, with about 1500 derivational subentries (i.e. an average of 1.5 derivational processes per root). At compile time, the XML dictionary is parsed and "downtranslated" into the source format required for the **lexc** compiler. The XML dictionary could be expanded by any competent Malay lexicographer.

4.2 Arabic Morphological Analyzer/Generator

The current Arabic system has been described in some detail in previous publications (Beesley, 1998) and is available for testing on the Internet.[4] The modification of the system to use the compile-replace algorithm has not changed the size or the behavior of the system in any way, but it has reduced the compilation time from hours to minutes.

[4] http://www.xrce.xerox.com/research/mltt/arabic/

```
Lexical:     k t b =Root C V C V C =Template a + =Voc
Surface:  ^[ k t b  .m>. C V C V C    .<m.   a +      ^]

Lexical:     d r s =Root C V V C V C =Template u * i =Voc
Surface:  ^[ d r s  .m>. C V V C V C   .<m.    u * i      ^]
```

Figure 13: Initial paths

```
Lexical:     k t b =Root C V C V C =Template a + =Voc
Surface:                k a t a b

Lexical:     d r s =Root C V V C V C =Template u * i =Voc
Surface:                d u u r i s
```

Figure 14: After Applying Compile-Replace to the Lower Side

5 Conclusion

The well-founded criticism of traditional implementations of finite-state morphology, that they are limited to handling concatenative morphotactics, is a direct result of their dependence on the concatenation operation in morphotactic description. The technique described here, implemented in the compile-replace algorithm, allows the regular-expression compiler to reapply to and modify its own output, effectively freeing morphotactic description to use any finite-state operation. Significant experiments with Malay and a much larger application in Arabic have shown the value of this technique in handling two classic examples of non-concatenative morphotactics: full-stem reduplication and Semitic stem interdigitation. Work remains to be done in applying the technique to other known varieties of non-concatenative morphotactics.

The compile-replace algorithm and the merge operator introduced in this paper are general techniques not limited to handling the specific morphotactic problems we have discussed. We expect that they will have many other useful applications.

References

Evan L. Antworth. 1990. *PC-KIMMO: a two-level processor for morphological analysis.* Summer Institute of Linguistics, Dallas.

Kenneth R. Beesley and Lauri Karttunen. 2000. *Finite-State Morphology: Xerox Tools and Techniques.* Cambridge University Press. Forthcoming.

Kenneth R. Beesley. 1991. Computer analysis of Arabic morphology: A two-level approach with detours. In *Perspectives on Arabic Linguistics III: Papers from the Third Annual Symposium on Arabic Linguistics*, pages 155–172. Amsterdam.

Kenneth R. Beesley. 1996. Arabic finite-state morphological analysis and generation. In *COLING'96*, volume 1, pages 89–94.

Kenneth R. Beesley. 1998. Arabic morphology using only finite-state operations. In *Computational Approaches to Semitic Languages: Proceedings of the Workshop*, pages 50–57, Montréal, Québec.

Lauri Karttunen, Ronald M. Kaplan, and Annie Zaenen. 1992. Two-level morphology with composition. In *COLING'92*, pages 141–148.

Laura Kataja and Kimmo Koskenniemi. 1988. Finite-state description of Semitic morphology: A case study of Ancient Akkadian. In *COLING'88*, pages 313–315.

Martin Kay. 1987. Nonconcatenative finite-state morphology. In *EACL'87*, pages 2–10.

George Anton Kiraz. 2000. Multi-tiered nonlinear morphology: A case study on Semitic. *Computational Linguistics*, 26(1).

Kimmo Koskenniemi. 1983. Two-level morphology: A general computational model for word-form recognition and production. Publication 11, University of Helsinki, Department of General Linguistics, Helsinki.

John J. McCarthy and Alan Prince. 1995. Faithfulness and reduplicative identity. Occasional papers in Linguistics 18, University of Massachusetts, Amherst, MA. ROA-60.

John J. McCarthy. 1981. A prosodic theory of nonconcatenative morphology. *Linguistic Inquiry*, 12(3):373–418.

Richard Sproat. 1992. *Morphology and Computation.* MIT Press, Cambridge, MA.