Default Finite State Machines and Finite State Phonology

Gerald Penn
Computational Linguistics Program
Carnegie Mellon University
Pittsburgh, PA 15213
Internet: penn@lcl.cmu.edu

Richmond Thomason
Intelligent Systems Program
University of Pittsburgh
Pittsburgh, PA 15260
Internet: thomason@isp.pitt.edu

May 19, 1994

Abstract

We propose DFSM's as an extension of finite state machines, explore some of their properties, and indicate how they can be used to formalize naturally occurring linguistic systems. We feel that this implementation of two-level rules may be more linguistically natural and easier to work with computationally. We provide complexity results that shed light on the computational situation.

1. Introduction

Two-level phonology combines the computational advantages of finite state technology with a formalism that permits phenomena to be described with familiar-looking rules. The problem with such a scheme is that, in practice, the finite state machines (FSM's) can grow too large to be manageable; one wants to describe them and to run them without having to deal with them directly. The KIMMO approach[1]seeks to achieve this by (1) decomposing the computational process into a battery of parallel finite state machines and (2) compiling rules (which notationally resemble familiar phonological rules, but which are interpreted declaratively) into these parallel finite state implementations. But the KIMMO formalism unfortunately gains no tractability in the process of compilation. Moreover, the compiler is complex enough to create software engineering problems, and this has led to practical difficulties, which in turn have made the KIMMO technology less generally available than one might wish. Here, we describe a different finite-state foundation for two-level rules, involving generalizations of FSM's which we call *Default Finite State Machines* (DFSM's). Whether or not this approach remains intractable after compilation is an open question; but even without compilation, we believe that it has some conceptual advantages as well.

DFSM's extend FSM's (specifically, finite-state transducers) so that transitions can be context-sensitive, and enforce a preference for the maximally specific transitions. The first

[1]See the discussion and references in (Sproat, 1992).

change allows phonological rules to appear as labels of transition arcs in transducers; the second change incorporates the elsewhere condition into the computational model.[2] DFSM's can be implemented directly, although there may be a method to compile them into a more efficient machine. We believe that either approach will be feasible for realistic linguistic applications (though, of course, not in the theoretically worst case). In particular, the direct implementation of DFSM's is very straightforward; no rule compiler is needed, since rules are labels on the arcs of the machines themselves. This implementation may not provide an optimal allocation of space and time usage at run time, but we believe that it will be adequate for testing and research purposes.

This presentation of DFSM's is confined to defining the basic ideas, presenting some examples of linguistic description, and providing a partial complexity analysis. In later work, we hope to explore descriptive and implementational issues further.

2. Notational preliminaries

We assume an alphabet $\mathcal{L}$, with a reserved symbol $0 \notin \mathcal{L}$ for insertions and deletions. A *replacement* over $\mathcal{L}$ is a pair of the form $\mathbf{l} = \langle l, l' \rangle$ where (1) $l \in \mathcal{L}$ and (2) $l' \in \mathcal{L}$ or $l' = 0$; $Replacements_{\mathcal{L}}$ is the set of replacements over $\mathcal{L}$. $US\text{-}strings_{\mathcal{L}}$ is the set of strings over the set $\mathcal{L}^2 \cup [\mathcal{L} \times \{0\}]$ of replacements.

We use roman letters to denote themselves: for instance, 'l' denotes the letter l. Boldface letters denote constant replacements: for instance, $\mathbf{l}$ is the pair $\langle \mathrm{l}, \mathrm{l} \rangle$. Moreover, ε is the empty string over $\mathcal{L}$, and $\boldsymbol{\varepsilon}$ is the empty string over the $\mathcal{L}$ replacements. When the name of a subset of $\mathcal{L}$ (e.g. C) is written in boldface, (e.g. $\mathbf{C}$), the set of identity pairings is intended (e.g., $\mathbf{C} = \{l : l \,/\, l \in C\}$).

We use ordinary italics as variables over letters, and boldface italics as variables over replacements and strings of replacements. Ordinarily, we will use $\boldsymbol{l}$ for replacements and $\boldsymbol{x}$, $\boldsymbol{y}$ for strings of replacements. Finally, we use 'l:l'' for the pair $\langle l, l' \rangle$.

Where $\boldsymbol{x} \in US\text{-}strings_{\mathcal{L}}$, $U\text{-}String(\boldsymbol{x})$ is the underlying projection of $\boldsymbol{x}$, and $S\text{-}String(\boldsymbol{x})$ is its surface projection. That is, if $\boldsymbol{x} = \langle x, x' \rangle$, then $U\text{-}String(\boldsymbol{x}) = x$ and $S\text{-}String(\boldsymbol{x}) = x'$.

3. Rule notation and examples

The rules with which we are concerned are like the rewrite rules of generative phonology; they are general, context-conditioned replacements. That is, a rule allows a replacement if (1) the replacement belongs to a certain type, and (2) the surrounding context meets certain constraints.

If we represent the contextual constraints extensionally, as sets of strings, a rule will consist of three things: a replacement type, and two sets of US-Strings. Thus, we can think of a rule as a triple $\langle \boldsymbol{X}, \boldsymbol{Y}, \boldsymbol{F} \rangle$, where $\boldsymbol{X}$ and $\boldsymbol{Y}$ are sets of US-strings. Imagine that we are given a replacement instance $\boldsymbol{l}$ in a context $\langle \boldsymbol{x}, \boldsymbol{y} \rangle$, where $\boldsymbol{x}$ and $\boldsymbol{y}$ are US-strings. This contextualized replacement $\langle \boldsymbol{x}, \boldsymbol{l}, \boldsymbol{y} \rangle$ *satisfies* the rule if $\boldsymbol{x} \in \boldsymbol{X}$, $\boldsymbol{y} \in \boldsymbol{Y}$, and $\boldsymbol{l} \in \boldsymbol{F}$.

[2] The elsewhere condition is built into an implementation (due to Karttunen) of the TWOL rule compiler; see (Dalrymple et al., 1993), pp. 28–32. But on this approach, default reasoning and the elsewhere condition are not employed at a level of computation that is theoretically modeled; this reasoning is simply a convenient feature of the code that translates rules into finite state automata.

For linguistic and computational purposes, the sets that figure in rules must somehow be finitely represented. The KIMMO tradition uses regular sets, which can of course be represented by regular expressions, for this purpose. We have not been able to convince ourselves that regular sets are needed in phonological applications.[3] Instead, we make the stronger assumption that contexts can be *encoded* by finite sets of strings. A string satisfies such a context when its left (or its right) matches one of the strings in this set. (Note that satisfaction is not the same as membership; infinitely many strings can satisfy a finite set of strings.) Assuming a finite alphabet, all replacement types will be finite sets. With these assumptions, a rule can be finitely encoded as a pair $\langle\langle \boldsymbol{X}, \boldsymbol{Y}\rangle, \boldsymbol{F}\rangle$, where the sets $\boldsymbol{X}$ and $\boldsymbol{Y}$ are finite, and $\boldsymbol{F}$ is a replacement type.

Rule encodings, rule applicability and satisfaction are illustrated by the rule examples given below. The ideas are further formalized in the next section.

Language:

Let $\mathcal{L} = \{\mathrm{a}, \mathrm{b}, \ldots, \mathrm{z}, +, \#, ', '\}$

Declare the following subsets of $\mathcal{L}$:

$C = \{\mathrm{b, c, d, f, g, h, j, k, l, m, n, p, q, r, s, t, v, w, x, y, z}\}$

$Csib = \{\mathrm{s, x, z}\}$

Example rules:

Example 1

Rule encoding: $\langle\langle\{\boldsymbol{\varepsilon}\}, \{\boldsymbol{\varepsilon}\}\rangle, \{\langle +, 0\rangle\}\rangle$

Rule notation: $+ \rightarrow 0 \;/\; \_\_$

Rule description: Delete +.

Example 2

Rule encoding: $\langle\langle \mathbf{C}, \{\langle +, 0\rangle\}\rangle, \{\langle \mathrm{y}, \mathrm{i}\rangle\}\rangle$

Rule notation: $y \rightarrow i \;/\; \mathbf{C}\_\_ \; + {:} 0$

Rule description: Replace y by i before a morpheme boundary and after a constant US-consonant, i.e. after $\langle l, l\rangle$, where $l \in \mathbf{C}$.

Example 3

Rule encoding: $\langle\langle\{\mathbf{sh}\}, \{\boldsymbol{l}^\frown\langle \#, 0\rangle \;/\; \boldsymbol{l} \in \mathbf{Csib}\}\rangle, \{\langle +, \mathrm{e}\rangle\}\rangle$

Rule notation: $+ \rightarrow \mathrm{e} \;/\; \mathbf{sh}\_\_\mathbf{Csib} \,\#{:}0$

Rule description: Replace + with e after **sh** and before a suffix in *Csib*.

Example rule applications:

[3]The issue here is whether there are any linguistically plausible or well-motivated applications of the Kleene star in stating phonological rules. For instance, take the English rule that replaces "e by 0 after a morpheme boundary preceded by one or more consonants preceded by a vowel." You could represent the context in question with the regular expression VC^*C; but you could equally well use $VC \mid VCC \mid VCCC \mid VCCCC$.The only way to distinguish the two rule formulations is by considering strings that violate the phonotactic constraints of English; but as far as we can see, there are no intuitions about the results of applying English rules to underlying strings like typppppe+ed. We do not question the usefulness of regular expressions in many computational applications, but are not convinced that they are needed in a linguistic setting. We would be interested to see a well motivated case in which the Kleene star is linguistically indispensable in formulating a two-level phonological rule. Such a case would create problems for the approach that we adopt here.

1. The rule encoded in Example 1 is satisfied by $\langle +, 0\rangle$ in the context $\langle$**cat** , s$\rangle$ because (1) for some $\boldsymbol{x}$, **cat** $= \boldsymbol{x}^\frown\boldsymbol{\varepsilon}$, (2) for some $\boldsymbol{y}$, s $= \boldsymbol{\varepsilon}^\frown\boldsymbol{y}$, and (3) $\langle +, 0\rangle \in \{\langle +, 0\rangle\}$.

2. The rule encoded in Example 2 is not satisfied by $\langle$y, i$\rangle$ in the context $\langle$**spot** + :t , +:0 **ness**$\rangle$ because there is no $\boldsymbol{x}$ such that **spot** + :t $= \boldsymbol{x}^\frown\boldsymbol{l}$, where $\boldsymbol{l} \in$ **C**.

3. The rule encoded in Example 3 is not satisfied by $\langle +, 0\rangle$ in the context $\langle$**ash** , s #:0$\rangle$. In fact, the context is satisfied: (1) **sh** $= \boldsymbol{x}^\frown$**sh** for some $\boldsymbol{x}$ and (2) s #:0 $\in$ **Csib** $^\frown\boldsymbol{y}$ for some $\boldsymbol{y}$. (3.1) Moreover, the underlying symbol of the replacement (namely, +) matches the argument of the rule's replacement function. Under these circumstances, we will say that the rule is *applicable*. But the rule is not satisfied, because (3.2) the surface symbol of the replacement (namely, 0) does not match the value of the rule's replacement function (namely, e): thus, $\langle +, 0\rangle \notin \{\langle +, e\rangle\}$.

4. Indexed strings and rules

We now restate the above ideas in the form of formal definitions.

Definition 1. *Context type.*
A context type is a pair $\mathcal{C} = \langle \boldsymbol{X}, \boldsymbol{Y}\rangle$, where $\boldsymbol{X}$ and $\boldsymbol{Y}$ are sets of US-Strings.

Definition 2. *Indexed US-strings.*
An indexed US-String over $\mathcal{L}$ is a triple $\langle \boldsymbol{x}, \boldsymbol{l}, \boldsymbol{y}\rangle$, where $\boldsymbol{x}, \boldsymbol{y} \in$ *US-strings*$_{\mathcal{L}}$ and $\boldsymbol{l} \in$ *Replacements*$_{\mathcal{L}}$.

An indexed US-string is a presentation of a nonempty US-string that divides the string into three components: (1) a replacement occurring in the string, (2) the material to the left of that replacement, and (3) the material to the right of it. Where $\langle \boldsymbol{x}, \boldsymbol{l}, \boldsymbol{y}\rangle$ is an indexed string, we call $\boldsymbol{x}$ the *left context* of the string, $\boldsymbol{y}$ the *right context* of the string, and $\boldsymbol{l}$ the *designated replacement* of the string.

A rule licenses certain sorts of replacements in designated sorts of environments, or *context types.* For instance, we may be interested in the environment after a consonant and before a morpheme boundary. Here, the phrase "after a consonant" amounts to saying that the string before the replacement must end in a consonant, and the phrase "before a morpheme boundary" says that the string after the replacement must begin in a morpheme boundary. Thus, we can think of a context type as a pair of constraints, one on the US-string to the left of the replacement, and the other on the US-string to its right. If we identify such constraints with the set of strings that satisfy them, a context type is then a pair of sets of US-strings; and an indexed string satisfies a context type in case its left and right context belong to the corresponding types.

Definition 3. *Replacement types.*
A replacement type over $\mathcal{L}$ is a partial function $\boldsymbol{F}$ from $\mathcal{L} \cup \{0\}$ to $\mathcal{L} \cup \{0\}$. (Thus, a replacement type is a certain set of replacements.) $Dom(\boldsymbol{F})$ is the domain of $\boldsymbol{F}$.

Definition 4. *Rules.*
A rule is a pair $\mathcal{R} = \langle \mathcal{C}, \boldsymbol{F}\rangle$, where $\mathcal{C}$ is a context type and $\boldsymbol{F}$ is a replacement type.

Definition 5. *Rule applicability.*
A rule $\langle\langle \boldsymbol{X}, \boldsymbol{Y}\rangle, \boldsymbol{F}\rangle$ is applicable to an indexed string $\langle \boldsymbol{x}, \langle l, l'\rangle, \boldsymbol{y}\rangle$ if and only if $\boldsymbol{x} \in \boldsymbol{X}$, $\boldsymbol{y} \in \boldsymbol{Y}$, and $\boldsymbol{F}(l)$ is defined, i.e., $l \in Dom(\boldsymbol{F})$.

Definition 6. *Rule satisfaction.*
An indexed string $\langle \boldsymbol{x}, \boldsymbol{l}, \boldsymbol{y}\rangle$ satisfies a rule $\langle \mathcal{C}, \boldsymbol{F}\rangle$ if and only if $\boldsymbol{x} \in \boldsymbol{X}$, $\boldsymbol{y} \in \boldsymbol{Y}$, and $\boldsymbol{F}(l) = l'$.

The above definitions do not assume that the contexts are finitely encodable. But, as we said, we are assuming as a working hypotheses that phonological contexts are finitely encodable; this idea was incorporated in the method of rule encoding that we presented above. We now make this idea explicit by defining the notion of a finitely encodable rule.

Definition 7. $LeftExp(\boldsymbol{X})$, $RightExp(\boldsymbol{X})$
$LeftExp(\boldsymbol{X}) = \{\boldsymbol{z}^\frown \boldsymbol{x} / \boldsymbol{x} \in \boldsymbol{X}\}$
$RightExp(\boldsymbol{X}) = \{\boldsymbol{x}^\frown \boldsymbol{z} / \boldsymbol{x} \in \boldsymbol{X}\}$

Definition 8. Finite encodability
A subset $\boldsymbol{X}$ of $US\text{-}strings_{\mathcal{L}}$ is *left-encoded* by a set $\boldsymbol{U}$ in case $\boldsymbol{X} = LeftExp(\boldsymbol{U})$, and is *right-encoded* by $\boldsymbol{V}$ in case $\boldsymbol{X} = RightExp(\boldsymbol{V})$. (It is easy to get confused about the usage of "left" and "right" here; in left encoding, the left of the encoded string is arbitrary, and the right must match the encoding set. We have chosen our terminology so that a left context type will be left-encoded and a right context type will be right-encoded.)
A context type $\mathcal{C} = \langle \boldsymbol{X}, \boldsymbol{Y}\rangle$ is encoded by a pair $\langle \boldsymbol{U}, \boldsymbol{V}\rangle$ of sets in case $\boldsymbol{U}$ left-encodes $\boldsymbol{X}$ and $\boldsymbol{V}$ right-encodes $\boldsymbol{Y}$.
A rule $\mathcal{R} = \langle \mathcal{C}, \boldsymbol{F}\rangle$ is finitely encoded by a rule encoding structure $\langle\langle \boldsymbol{U}, \boldsymbol{V}\rangle, \boldsymbol{g}\rangle$ in case $\langle \boldsymbol{U}, \boldsymbol{V}\rangle$ encodes $\mathcal{C}$, $\boldsymbol{g} = \boldsymbol{F}$, and $\boldsymbol{U}$ and $\boldsymbol{V}$ are finite.

In the following material, we will not only confine our attention to finitely encodable rules, but will refer to rules by their encodings; when the notation $\langle\langle \boldsymbol{X}, \boldsymbol{Y}\rangle, \boldsymbol{F}\rangle$ appears below, it should be read as a rule encoding, not as a rule. Thus, for instance, the indexed string $\langle$**cat**, +:0 , s$\rangle$ satisfies the rule (encoded by) $\langle\langle\{\boldsymbol{\varepsilon}\}, \{\boldsymbol{\varepsilon}\}\rangle, \{\langle +, 0\rangle\}\rangle$, even though **cat** $\notin \{\boldsymbol{\varepsilon}\}$.

5. Specificity of context types and rules

We have a good intuitive grasp of when one context type is more specific than another. For instance, the context type *preceded by a back vowel* is more specific than the type *preceded by a vowel*; the context type *followed by an obstruent* is neither more nor less specific than the type *followed by a voiced consonant*; the context type *preceded by a vowel* is neither more nor less specific than the type *followed by a vowel*.

Since we have identified context types with pairs of sets of strings, we have a very natural way of defining specificity relations such as "more specific than", "equivalent", and "more specific than or equivalent": we simply use the subset relation.

Definition 9. $\mathcal{C} \leq \mathcal{C}'$.
Let $\mathcal{C} = \langle \boldsymbol{X}_1, \boldsymbol{Y}_1\rangle$ and $\mathcal{C}' = \langle \boldsymbol{X}_2, \boldsymbol{Y}_2\rangle$ be context types. $\mathcal{C} \leq \mathcal{C}'$ if and only if $\boldsymbol{X}_1 \subseteq \boldsymbol{X}_2$ and $\boldsymbol{Y}_1 \subseteq \boldsymbol{Y}_2$.

Definition 10. $\mathcal{C} \equiv \mathcal{C}'$.
$\mathcal{C} \equiv \mathcal{C}'$ if and only if $\mathcal{C} \leq \mathcal{C}'$ and $\mathcal{C}' \leq \mathcal{C}$.

Definition 11. $\mathcal{C} < \mathcal{C}'$.
$\mathcal{C} < \mathcal{C}'$ if and only if $\mathcal{C} \leq \mathcal{C}'$ and $\mathcal{C}' \not\equiv \mathcal{C}$.

It is not in general true that if $\mathit{LeftExp}(\boldsymbol{X}) \subseteq \mathit{LeftExp}(\boldsymbol{Y})$, then $\boldsymbol{X} \subseteq \boldsymbol{Y}$; for instance, $\mathit{LeftExp}(\{aa, ba\}) \subseteq \mathit{LeftExp}(\{a\})$, but $\{aa, ba\} \not\subseteq \{a\}$. However, we can easily determine the specificity relations of two contexts from their finite encodings:

Lemma 1. $\mathit{LeftExp}(\boldsymbol{X}) \subseteq \mathit{LeftExp}(\boldsymbol{Y})$ iff for all $\boldsymbol{x} \in \boldsymbol{X}$ there is a $\boldsymbol{y} \in \boldsymbol{Y}$ such that for some $\boldsymbol{z}$, $\boldsymbol{x} = \boldsymbol{z}^\frown\boldsymbol{y}$. Similarly, $\mathit{RightExp}(\boldsymbol{X}) \subseteq \mathit{RightExp}(\boldsymbol{Y})$ iff for all $\boldsymbol{x} \in \boldsymbol{X}$ there is a $\boldsymbol{y} \in \boldsymbol{Y}$ such that for some $\boldsymbol{z}$, $\boldsymbol{x} = \boldsymbol{y}^\frown\boldsymbol{z}$.

Proof of the lemma is immediate from the definitions. It follows from the lemma that there is a tractable algorithm for testing specificity relations on finitely encodable contexts:

Lemma 2. Let $\mathcal{C}$ be finitely encoded by $\langle \boldsymbol{X}_1, \boldsymbol{X}_2 \rangle$ and $\mathcal{C}'$ be finitely encoded by $\langle \boldsymbol{Y}_1, \boldsymbol{Y}_2 \rangle$. Then there is an algorithm for testing whether $\mathcal{C} \leq \mathcal{C}'$ that is no more complex than $O(m \times n \times k)$, where $m = \max(|\boldsymbol{X}_1|, |\boldsymbol{X}_2|)$, $n = \max(|\boldsymbol{Y}_1|, |\boldsymbol{Y}_2|)$, and k is the length of the longest string in $\boldsymbol{Y}_1 \cup \boldsymbol{Y}_2$.

Proof. Test whether for each $\boldsymbol{x}_1 \in \boldsymbol{X}_1$ there is a $\boldsymbol{y}_1 \in \boldsymbol{Y}_1$ that matches the end of $\boldsymbol{x}_1$. Then perform a similar test on $\boldsymbol{X}_2$ and $\boldsymbol{Y}_2$.

6. Dfsm's

A DFSM's transitions are labelled with finitely encodable rules rather than with pairs of symbols. Moreover, nondeterminism is restricted so that in case of conflicting transitions, a maximally specific transition must be selected. The critical definition is that of *minimal satisfaction of an arc by an indexed path*, where an indexed path represents a DFSM derivation, by recording the state transitions and replacements that are traversed in processing a US-String.

Definition 12. *Arcs.*
An arc over a set $\mathcal{S}$ of states and alphabet $\mathcal{L}$ is a triple $\mathbf{A} = \langle s, s', \mathcal{R} \rangle$, where $s, s' \in \mathcal{S}$ and $\mathcal{R}$ is a rule over $\mathcal{L}$.

Definition 13. *DFSMs.*
A DFSM on $\mathcal{L}$ is a structure $\mathcal{M} = \langle \mathcal{S}, i, \mathcal{T}, \mathcal{A} \rangle$, where $\mathcal{S}$ is a finite set of states, $i \in \mathcal{S}$ is the initial state, $\mathcal{T} \subseteq \mathcal{S}$ is the set of terminal states, and $\mathcal{A}$ is a set of arcs over $\mathcal{S}$ on $\mathcal{L}$.

Definition 14. *Paths.*
A path π or $\pi(s_0, s_n)$ over $\mathcal{M}$ from state s_0 to state s_n is a string $s_0\mathbf{l}_1s_1\mathbf{l}_1 \ldots \mathbf{l}_ns_n$, where for all m, $0 \leq m \leq n$, s_m is a state of $\mathcal{M}$ and $\mathbf{l}_m \in \mathit{US\text{-}strings}_{\mathcal{L}}$.

Remark 1: $n \geq 0$, so that the simplest possible path has the form s, where s is a state. *Remark 2:* we use the notations π and $\pi(s, s')$ alternatively for the same path; the second notation provides a way of referring to the beginning and end states of the path.

Definition 15. *Recovery of strings from paths.*
Let $\pi = s_0\mathbf{l}_1s_1\mathbf{l}_1 \ldots \mathbf{l}_ns_n$. Then $\mathit{String}(\pi) = \mathbf{l}_1 \ldots \mathbf{l}_n$.

Definition 16. *Indexed paths.*
An indexed path over $\mathcal{M}$ is a triple $\langle \pi, \mathbf{l}, \pi' \rangle$ where π, π' are paths, and $\mathbf{l}_m \in \mathit{US\text{-}strings}_{\mathcal{L}}$. $\langle \pi, \mathbf{l}, \pi' \rangle$ is an indexing of path σ if and only if $\sigma = \pi^\frown\mathbf{l}^\frown\pi'$.

Definition 17. *Applicability of an arc to an indexed path.*
An arc $\langle u, u', \mathcal{R}\rangle$ is applicable to an indexed path $\langle \pi(s,t), \mathbf{l}, \pi'(s',t')\rangle$ if and only if $t = u$ and the rule $\mathcal{R}$ is applicable to the indexed string $\langle String(\pi), \mathbf{l}, String(\pi')\rangle$.

Definition 18. *Satisfaction of an arc by an indexed path.*
$\langle \pi(s,t), \mathbf{l}, \pi'(s',t')\rangle$ satisfies an arc $\langle u, u', \mathcal{R}\rangle$ if and only if $t = u$, $s' = u'$, and the indexed string $\langle String(\pi), \mathbf{l}, String(\pi')\rangle$ satisfies the rule $\mathcal{R}$.

Definition 19. *Minimal satisfaction of an arc by an indexed path.*
$\langle \pi, \mathbf{l}, \pi'\rangle$ minimally satisfies an arc $\mathbf{A} = \langle s, s', \mathcal{R}\rangle$ of $\mathcal{M}$ if and only if $\langle \pi, \mathbf{l}, \pi'\rangle$ satisfies $\mathbf{A}$ and there is no state s'' and arc $\mathbf{A}' = \langle s, s'', \mathcal{R}'\rangle$ of $\mathcal{M}$ such that $\mathbf{A}' = \langle s, s'', \mathcal{R}'\rangle$ is applicable to $\langle \pi, \mathbf{l}, \pi'\rangle$ and $\mathcal{R}' < \mathcal{R}$.

As we said, the above definition is the crucial component of the definition of DFSM's. According to this definition, to see whether a DFSM derivation is correct, you must check that each state transition represents a maximally specific rule application. This means that at each stage the DFSM does not provide another arc with a competing replacement and a more specific context. ("Competing" means that the underlying symbols of the replacement match; a replacement competes even if the surface symbols does not match the letter in the US-String being tested.)[4]

Definition 20. *Indexed path acceptance by a DFSM.*
$\mathcal{M} = \langle \mathcal{S}, i, \mathcal{T}, \mathcal{A}\rangle$ accepts an indexed path $\langle \pi, \mathbf{l}, \pi'\rangle$ if and only if there is an arc $\mathbf{A}' = \langle s, s', \mathcal{R}'\rangle$ of $\mathcal{M}$ that is minimally satisfied by $\langle \pi, \mathbf{l}, \pi'\rangle$.

Definition 21. *Path acceptance by a DFSM.*
$\mathcal{M} = \langle \mathcal{S}, i, \mathcal{T}, \mathcal{A}\rangle$ accepts a path $\pi(s, s')$ if and only if $\mathcal{M}$ accepts every indexing of π, $s = i$, and $s' \in \mathcal{T}$.

Definition 22. *US-String acceptance by a DFSM.*
$\mathcal{M}$ accepts $\boldsymbol{x} \in$ *US-strings*$_{\mathcal{L}}$ if and only if there is a path π such that $\mathcal{M}$ accepts π, where $\boldsymbol{x} = String(\pi)$.

Definition 23. *Generation of SF from UF by a DFSM.*
$\mathcal{M}$ generates a surface form x' from an underlying form x (where x and x' are strings over $\mathcal{L}$) if and only if there is a $\boldsymbol{x} \in$ *US-strings*$_{\mathcal{L}}$ such that $\mathcal{M}$ accepts $\boldsymbol{x}$, where *U-String*$(\boldsymbol{x}) = x$ and *S-String*$(\boldsymbol{x}) = x'$.

7. Example: spelling rules for english stem+suffix combinations

The following is an adaptation of the treatment in Antworth (1990) of English spelling rules, which in turn is taken from Karttunen and Wittenburg (1983).

- $\mathcal{M} = \langle \mathcal{S}, i, \mathcal{T}, \mathcal{A}\rangle$, where $\mathcal{S} = \{i, s, t\}$. $\mathcal{T} = \{t\}$.

[4] This use of competition builds some directional bias into the definition of DFSM's, i.e., some preference for their use in generation. Even if we are using DFSM's for recognition, we will need to verify that the recognized string is generated from an underlying form by a derivation that does not allow more specific competing derivations.

- *Task of i:* Begin and process left word boundary.
- *Task of s:* Process stem and suffixes.
- *Task of t:* Quit, having processed right word boundary.

- *Remark:* the small number of states is deceptive, since contexts are allowed on the arcs. An equivalent finite-state transducer would have many hundreds of states at least.

- *Remark:* the relatively small number of arcs enumerated below is also deceptive, since two of these "arcs," arc 3 and arc 13, are actually schemes. In the following discussion we will speak loosely and refer to these schemes as arcs; this will simplify the discussion and should create no confusion.

- Declare the following subsets of $\mathcal{L}$:

$$\begin{aligned}
Ltr &= \{\mathrm{a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r} \\
&\quad \mathrm{s, t, u, v, w, x, y, z}\} \\
C &= \{\mathrm{b, c, d, f, g, h, j, k, l, m, n, p, q, r, s, t, v, w,} \\
&\quad \mathrm{x, y, z}\} \\
Csib &= \{\mathrm{s, x, z}\} \\
Cpal &= \{\mathrm{c, g}\} \\
V &= \{\mathrm{a, e, i, o, u}\} \\
Vbk &= \{\mathrm{a, o, u}\};
\end{aligned}$$

- Where $s, s' \in \mathcal{S}$, let $\mathcal{A}_{s,s'} = \{\mathbf{A} \,/\, \mathbf{A} \in \mathcal{A}$ and for some $\mathcal{R}, \mathbf{A} = \langle s, s', \mathcal{R}\rangle\}$. We present arcs by listing the rules associated with the arcs, for each appropriate pair $\langle s, s'\rangle$ of states. We will give each arc a numerical label, and give a brief explanation of the purpose of the arc.

- **Arcs in $\mathcal{A}_{i,s}$:**

  1. $\# \rightarrow 0 \,/\, \_$
     Delete left word boundary.

- **Arcs in $\mathcal{A}_{s,s}$:**

  2. $+ \rightarrow 0 \,/\, \_$
     Delete morpheme boundary.
  3. $l \rightarrow l \,/\, \_ \;:\; l \in Ltr$
     Any underlying letter is normally unchanged.
  4. ' $\rightarrow$ ' $/\, \_$
     Apostrophe is normally unchanged.
  5. $' \rightarrow' \,/\, \_$
     Stress is normally unchanged.

6. $+ \rightarrow$ e / [**Csib** | **ch** | **sh** | y:i]__**s** [+:0 | #:0]
   Epenthesis before -s suffix.

7. y $\rightarrow$ i / **C**__ + :0
   Spell y as i after consonant and before suffix.

8. y $\rightarrow$ y / **C**__ + :0 [i:i | ':']
   Exception to Rule 7; cf. "trying", "fly's".

9. s $\rightarrow$ 0 / [+:0 | +:e]**s** +:0 '__
   Delete possessive 's after plural suffix.

10. e $\rightarrow$ 0 / **VCC**$^+$__ + :0 **V**
    Elision.[5]

11. e $\rightarrow$ e / **VC**$^+$**Cpal**__ +:0 **Vbk**
    Exception to Rule 10.

12. i $\rightarrow$ y / __e:0 +:0 **i**
    Spell i as y before elided e before i-initial suffix.

13. $+ \rightarrow l \,/\, ' : 0\, \mathbf{C}^+\, \mathbf{V}\, l : l$ __$[V \,|\, y]$:
    $l \in \{\mathrm{b, d, g, l, m, n, p, r, t}\}$
    Gemination.

- **Arcs in $\mathcal{A}_{s,t}$:**

14. # $\rightarrow$ 0 / __
    Delete right word boundary.

- **Illustrations**

I. The derivation that relates #kiss+s# to 0kisses0 proceeds as follows.

1. Begin in state i looking at #:0.
2. Follow arc 2 to s, recognizing k:k. (This is the only applicable arc.)
3. Follow arc 3 to s, recognizing i:i. (This is the only applicable arc.)
4. Follow arc 3 to s, recognizing s:s. (This is the only applicable arc.)
5. Follow arc 3 to s, recognizing s:s. (This is the only applicable arc.)
6. Follow arc 6 to s, recognizing +:e. (Arc 2 is also applicable here; but see the next illustration.)
7. Follow arc 3 to s, recognizing s:s. (This is the only applicable arc.)

[5] Here, $\mathbf{C}^+$ can be any string of no more than four consonants.

8. Follow arc 14 to f, recognizing #:0. (This is the only applicable arc.)

II. No derivation relates #kiss+s# to 0kiss0s0. Any such derivation would have to proceed like the above derivation through Step 5. At the next step, the conditions for two arcs are met: arc 2 (replacing + with 0) and arc 6 (replacing + with e). Since the context of the latter arc is more specific, it must apply; there is no derivation from this point using arc 2.

III. The derivation that relates #try+ing# to 0try0ing0 proceeds as follows.

1. Begin in state i looking at #:0.
2. Follow arc 2 to s, recognizing t:t. (This is the only applicable arc.)
3. Follow arc 3 to s, recognizing r:r. (This is the only applicable arc.)
4. Follow arc 8 to s, recognizing y:y. (There are three applicable arcs at this point: arc 3, arc 7, and arc 8. However, arcs 3 and 7 are illegal here, since their contexts are both less specific than arc 8's.)
5. Follow arc 2 to s, recognizing +:0. (This is the only applicable arc.)
6. Follow arc 3 to s, recognizing i:i. (This is the only applicable arc.)
7. Follow arc 3 to s, recognizing n:n. (This is the only applicable arc.)
8. Follow arc 3 to s, recognizing g:g. (This is the only applicable arc.)
9. Follow arc 14 to f, recognizing #:0. (This is the only applicable arc.)

IV. No derivation relates #try+ing# to 0tri0ing0. Any such derivation would have to proceed like the above derivation through Step 3. At the next step, arc 7 cannot be traversed, since arc 8 is also applicable and its context is more specific. Therefore, no arc is minimally satisfied and the derivation halts at this point.

8. Computational complexity

We now consider the complexity of using DFSM's to create one side of a US-string, given the other side as input. There are basically two tasks to be analyzed:

- **DFSM GENERATION**: Given a DFSM, D, over an alphabet, $\mathcal{L}$, and an underlying form, u, does D generate a surface form, s, from u?
- **DFSM RECOGNITION**: Given a DFSM, D, over an alphabet, $\mathcal{L}$, and a surface form, s, does D generate an underlying form, u, from s?

These two tasks are related to the tasks of KIMMO GENERATION and KIMMO RECOGNITION, the various versions of which Barton et al. (1987) proved to be NP-complete or worse.

8.1. Relationship to Kimmo

The DFSM is not a generalization of KIMMO; it is an alternative architecture for two-level rules. KIMMO takes a programming approach; it provides a declarative rule formalism, which can be related to a very large FS automaton or to a system of parallel FS automata. The automata are in general too unwieldy to be pictured or managed directly; they are manipulated using the rules. By integrating rules into the automata, the DFSM approach provides a procedural formalism that is compact enough to be diagrammed and manipulated directly.

DFSM rules are procedural; their meaning depends on the role that they play in an algorithm. In a DFSM with many states, the effect achieved by a rule (where a rule is a context-dependent replacement type) will in general depend on how the rule is attached to states. In practice, however, the proceduralism of the DFSM approach can be limited by allowing only a few states, which have a natural morphonemic interpretation. The English spelling example that we presented in the previous section illustrates the idea. There are only four states. Of these, two of them delimit word processing; one of them begins processing by traversing a left word boundary, the other terminates processing after traversing a final word boundary. Of the remaining two states, one processes the word; all of the rules concerning possible replacements are attached to arcs that loop from this state to itself. The other is a nonterminal state with no arcs leading from it. In the example, the only purpose of this state is to render certain insertions or deletions obligatory, by "trapping" all US-strings in which the operation is not performed in the required context.

In cases of this kind, where the ways in which rules can be attached to arcs are very restricted, the proceduralism of the DFSM formalism is limited. The uses of rules in such cases correspond roughly to two traditional types of phonological constructs: rules that allow certain replacements to occur, and constraints that make certain replacements obligatory.

Although DFSM's are less declarative than KIMMO, we believe that it may be possible to interpret at least some DFSM's (those in which the roles that can be played by states are limited) using a nonmonotonic formalism that provides for prioritization of defaults, such as prioritized default logic; see (Brewka, 1993). In this way, DFSM's could be equated to declarative, axiomatic theories with a nonmonotonic consequence relation. But we have not carried out the details of this idea.

Though it is desirable to constrain the number of states in a DFSM, there may be applications in which we may want more states than in the English example. For instance, one natural way to process vowel harmony would multiply states by creating a word-processing state for each vowel quality. Multiple modes of word-processing could also be used to handle cases (as in many Athabaskan languages) where different morphophonemic processes occur in different parts of the word.

If they are desired, local translations of the four varieties of KIMMO rules[6] into DFSM's are available, by using only one state plus a sink state. The following correspondences provide translations, in polynomial time, to one or more DFSM arcs:

- *Exclusion*, $u : s / \Leftarrow LC\_RC$: an arc $u \rightarrow s \ / \ LC\_RC$ from the state to a sink state.

[6]Sproat (1992), p. 145.

- *Context Restriction*, $u : s \Rightarrow LC\_RC$: a loop $u \rightarrow s$ / LC_RC, and an arc $u \rightarrow s$ / $\_$ to a sink state.
- *Surface Coercion*, $u : s \Leftarrow LC\_RC$: a loop $u \rightarrow s$ / LC_RC, and for each surface character $s' \in \mathcal{L}$, an arc $u \rightarrow s'$ / LC_RC to a sink state.
- *Composite*, $u : s \Leftrightarrow LC\_RC$: all of the arcs mentioned in Context Restriction or Surface Coercion.

8.2. Extended DFSM's

The differences between Kimmo and DFSM's prohibit the complexity analysis for the corresponding two Kimmo problems from naturally extending to an analysis of DFSM generation and recognition. In fact, we can define an *extended DFSM (EDFSM)*, which drops the finite encodability requirement that Kimmo lacks, for which we have the following result:

Theorem 1. EDFSM GENERATION is PSPACE-hard

Proof by reduction of REGULAR EXPRESSION NON-UNIVERSALITY (see Figure 1). Given an alphabet Σ, and a regular expression, $a \neq \phi$, over Σ, we define an EDFSM over the alphabet, $\Sigma \cup \{\$\}$, where $\$ \notin \Sigma$. We choose one non-empty string $\alpha \in L(a)$ of length n. The EDFSM first recognizes each character in α, completing the task at state s_0:

$$\alpha_1 \rightarrow \alpha_1 \;/\; (\mathcal{L}{:}\mathcal{L})^*\_(\mathcal{L}{:}\mathcal{L})^* \text{ [7]}$$
$$\vdots$$
$$\alpha_n \rightarrow \alpha_n \;/\; (\mathcal{L}{:}\mathcal{L})^*\_(\mathcal{L}{:}\mathcal{L})^*$$

From s_0, there are two arcs, which map to different states:

$$\$ \rightarrow \$ \;/\; \mathbf{\Sigma}^*\_\mathbf{\Sigma}^*$$
$$\$ \rightarrow \$ \;/\; (\mathbf{a}+\$)\_(\mathbf{a}+\$)$$

where the latter rule traverses to some state s_1, with $\mathbf{a}$ being the expression which replaces each atom, b, in a by its constant replacement, $b{:}b$, and likewise for $\mathbf{\Sigma}$.

From s_1, the EDFSM then recognizes α again, terminating at the only final state. We provide this EDFSM, along with the input $\alpha\$\alpha$ to EDFSM GENERATION. This EDFSM can accept $\alpha\$\alpha$ if and only if, at state s_0, the context $\langle \mathbf{\Sigma}^*, \mathbf{\Sigma}^* \rangle$ is not more specific than the context $\langle (\mathbf{a}+\$), (\mathbf{a}+\$) \rangle$. So, we have:

$$\begin{aligned}
&\langle \mathbf{\Sigma}^*, \mathbf{\Sigma}^* \rangle \not< \langle (\mathbf{a}+\$), (\mathbf{a}+\$) \rangle \\
&\Leftrightarrow \langle \mathbf{\Sigma}^*, \mathbf{\Sigma}^* \rangle \not\leq \langle (\mathbf{a}+\$), (\mathbf{a}+\$) \rangle \\
&\text{or } \langle \mathbf{\Sigma}^*, \mathbf{\Sigma}^* \rangle \equiv \langle (\mathbf{a}+\$), (\mathbf{a}+\$) \rangle \\
&\Leftrightarrow \mathbf{\Sigma}^* \not\subseteq L(\mathbf{a}+\$) \\
&\text{or } \mathbf{\Sigma}^* = L(\mathbf{a}+\$) \\
&\Leftrightarrow \mathbf{\Sigma}^* \not\subseteq L(\mathbf{a}+\$), \text{ since } \$ \notin \Sigma, \\
&\Leftrightarrow \mathbf{\Sigma}^* \not\subseteq L(\mathbf{a}) \cup \{\$\}
\end{aligned}$$

[7] Unlike with normal DFSM's, we will use regular expressions for the contexts themselves in EDFSM's, not their encodings, since they may be infinite anyway.

$\Leftrightarrow \mathbf{\Sigma}^* \not\subseteq L(\mathbf{a})$, since $\$ \notin \Sigma$,
$\Leftrightarrow \Sigma^* \not\subseteq L(a)$
$\Leftrightarrow L(a) \neq \Sigma^*$ (we know $L(a) \subseteq \Sigma^*$)

The translation function is linear in the size of the input.$\Box$

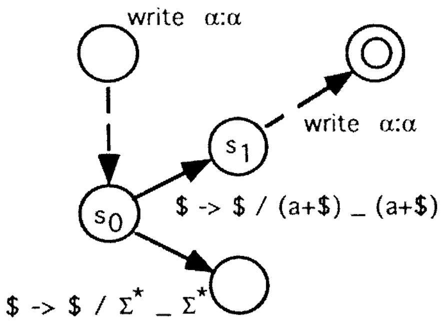

Figure 1. EDFSM constructed in Theorem 1.

8.3. The Complexity of DFSM GENERATION

Finite encodability foils the above proof technique, since one can no longer express arbitrary regular expressions over pairs in the contexts of rules. In fact, as we demonstrated above, there is a polynomial-time algorithm for comparing the specificities of finitely-encodable contexts. Finite encodability does not, however, restrict the complexity of DFSM's enough to make DFSM GENERATION polynomial time:

Theorem 2. DFSM GENERATION is NP-complete.

Proof DFSM GENERATION is obviously in NP. The proof of NP-hardness is a reduction of 3-SAT. Given an input formula, w, we construct a DFSM consisting of one state over an alphabet consisting of 0, 1, #, one symbol, u_i, for each variable in w, and one symbol, c_j, for each conjunct in w. Let m be the number of variables in w, and n, the number of conjuncts. For each variable, u_i, we add four loops:

$$u_i \rightarrow 1 \ / \ \#{:}\# \ u_1{:}\mathcal{L} \ \ldots \ u_{i-1}{:}\mathcal{L} \_\_,$$

$$u_i \rightarrow 0 \ / \ \#{:}\# \ u_1{:}\mathcal{L} \ \ldots \ u_{i-1}{:}\mathcal{L} \_\_,$$

$$u_i \rightarrow 1 \ / \ u_i{:}1 \ u_{i+1}{:}\mathcal{L} \ \ldots \ u_m{:}\mathcal{L} \ \mathcal{L}{:}\mathcal{L} \ u_1{:}\mathcal{L} \ \ldots \ u_{i-1}{:}\mathcal{L} \_\_,$$

$$u_i \rightarrow 0 \ / \ u_i{:}0 \ u_{i+1}{:}\mathcal{L} \ \ldots \ u_m{:}\mathcal{L} \ \mathcal{L}{:}\mathcal{L} \ u_1{:}\mathcal{L} \ \ldots \ u_{i-1}{:}\mathcal{L} \_\_$$

The first two choose an assignment for a variable, and the second two enforce that assignment's consistency. For each conjunct, $l_{j_1} \vee l_{j_2} \vee l_{j_3}$, where the l's are literals, we also add three loops, one for each literal. The loops enforce a value of 1 on the symbol u_{j_i} if l_{j_i} is a positive literal, or 0, if it is negative. For example, for the conjunct $u_1 \vee \neg u_3 \vee u_4$, we add the following three rules:

$$c_j \rightarrow c_j \ / \ u_1{:}1 \ u_2{:}\mathcal{L} \ \ldots \ u_m{:}\mathcal{L} \ \_$$
$$c_j \rightarrow c_j \ / \ u_3{:}0 \ u_4{:}\mathcal{L} \ \ldots \ u_m{:}\mathcal{L} \ \_$$
$$c_j \rightarrow c_j \ / \ u_4{:}1 \ u_5{:}\mathcal{L} \ \ldots \ u_m{:}\mathcal{L} \ \_$$

Thus, the input to DFSM GENERATION is the above DFSM plus an input string created by iterating the substring $u_1 \ldots u_m c_j$ for each conjunct. The input string corresponding to the formula, $(\neg u_1 \vee u_2 \vee u_4) \wedge (\neg u_2 \vee u_3 \vee \neg u_4) \wedge (u_1 \vee u_2 \vee u_3)$, would be $\# u_1 u_2 u_3 u_4 c_1 u_1 u_2 u_3 u_4 c_2 u_1 u_2 u_3 u_4 c_3$. The DFSM accepts this input string if and only if the input formula is satisfiable; and this translation is linear in $m + n$. □

8.4. Compilation

Of course, we should consider whether the complexity of DFSM GENERATION can be compiled out, leaving a polynomial-time machine which accepts input strings. This can be formalized as the separate problem:

- **FIXED-DFSM-GENERATION**: For some DFSM, D, over alphabet, $\mathcal{L}$, given an underlying form, u, does D generate a surface form, s, from u?

Whether or not FIXED DFSM GENERATION belongs to P remains an open problem. It is, of course, no more difficult than the general DFSM GENERATION problem, and thus no more difficult than NP-complete. The method used in the proof given above, however, does not naturally extend to the case of FIXED DFSM GENERATION, since we cannot, with a fixed DFSM, know in advance how many variables to expect in a given input formula, without which we cannot use the same trick with the left context to preserve the consistency of variable assignment.

Even more interestingly, the technique used in the proof of PSPACE-hardness of EDFSM GENERATION does not naturally extend to fixed EDFSM's either; thus, whether or not FIXED DFSM GENERATION belongs to P is an open question as well[8]. Dropping finite encodability, of course, affects the compilation time of the problem immensely.

8.5. Nulls

The two proofs we have given remain valid if we switch all of the underlying forms with their surface counterparts. Thus, without nulls, EDFSM RECOGNITION is PSPACE-hard, DFSM RECOGNTION is NP-complete, and, if FIXED DFSM GENERATION is in P, then we can presumably use the same compilation trick with the roles of underlying and surface strings reversed to show that FIXED DFSM RECOGNITION is in P as well.

If nulls are permitted in surface realizations, however, DFSM RECOGNTION becomes much more difficult, even with finite encodability enforced:

Theorem 3. DFSM RECOGNTION with nulls is PSPACE-hard.

Proof by reduction of CONTEXT-SENSITIVE LANGUAGE MEMBERSHIP (see Figure 2). Given a context-sensitive grammar and an input string of length m, we let the input surface

[8]It is quite unlikely, however, since the reduction can probably be made with a different PSPACE-complete problem, from which the NP-completeness of FIXED EDFSM GENERATION would follow as a corollary.

form to the DFSM RECOGNTION problem be the same as the input string. We then design a DFSM with an alphabet equal to $\Sigma \cup \{\$, !\}$, where Σ is the the set of non-terminals plus the set of terminals. The DFSM first copies each surface input symbol to the corresponding position in the underlying form, and then adds the pair $\$$:0, completing the task in a state s_0.

Having copied the string onto the underlying side of the pair, the remainder of the recognized underlying form will consist of rewritings of the string for each rule application, and will be paired with surface nulls at the end of the input string. Each rewriting will be separated by a $ symbol, and, as the string length changes, it will be padded by ! symbols. For each rule $\alpha \rightarrow \beta$, we add a cycle to the DFSM, emanating from state s_0, which first writes j copies of the ! symbol to the underlying form, where $j = b - a$, $b = |\beta|$, and $a = |\alpha|$:

$$! \rightarrow 0 \;/\; \Sigma{:}\mathcal{L}\,(\,\mathcal{L}{:}\mathcal{L}\, \ldots_m\, \mathcal{L}{:}\mathcal{L}\,)\_\_$$

$j \geq 0$ since the rules are context-sensitive.

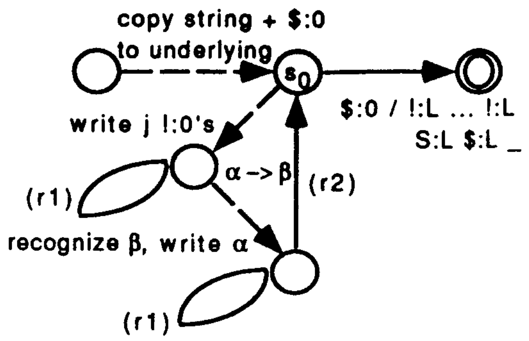

Figure 2. DFSM constructed in Theorem 3.

The cycle then copies part of the most recent $-bounded string of symbols with a family of loops of the form:

$$\sigma \rightarrow 0 \;/\; \sigma{:}\mathcal{L}\,(\,\mathcal{L}{:}\mathcal{L}\, \ldots_{m+j}\, \mathcal{L}{:}\mathcal{L}\,)\_\_ \qquad \text{(r1)}$$

for each $\sigma \in \Sigma$. It then recognizes β, and : writes α, with:

$$\begin{aligned}\alpha_1 \rightarrow 0 \;/\; &(\,\beta_1{:}\mathcal{L}\, \ldots_b\, \beta_b{:}\mathcal{L}\,)\\ &(\,\mathcal{L}{:}\mathcal{L}\, \ldots_{m+j+1-b}\, \mathcal{L}{:}\mathcal{L}\,)\_\_,\end{aligned}$$

followed by:

$$\begin{aligned}&\alpha_2 \rightarrow 0 \;/\; \_\_,\\ &\vdots\\ &\alpha_a \rightarrow 0 \;/\; \_\_\end{aligned}$$

It then copies the rest of the most recent $-bounded string, using copy of the family of loops in (r1), and then adds a new $ with a rule that also ensures that this second loop has iterated the appropriate number of times by checking that the length has been preserved:

$$\$ \rightarrow 0 \;/\; \$ {:}\mathcal{L}\,(\,\mathcal{L}{:}\mathcal{L}\, \ldots_m\, \mathcal{L}{:}\mathcal{L}\,)\_\_ \qquad \text{(r2)}$$

The DFSM also has a loop emanating from s_0 which adds more ! symbols:

$$! \rightarrow 0 \;/\; !{:}\mathcal{L}\,(\,\mathcal{L}{:}\mathcal{L}\;\ldots_m\;\mathcal{L}{:}\mathcal{L}\,)\_\_$$

All of the rule-cycles will use this to copy previously-added ! symbols, as the string shrinks in size. The proper application of this loop is also ensured by the length-checking of (r2).

Finally, we add one arc to the DFSM from s_0 to the only final state which checks that the final copy of the string contains only the distinguished symbol, S:

$$\$ \rightarrow 0 \;/\; (\,!{:}\mathcal{L}\;\ldots_{m-1}\,!{:}\mathcal{L}\,)\,S{:}\mathcal{L}\;\${:}\mathcal{L}\,\_\_$$

Thus, the DFSM recognises the surface form if and only if there is a series of rewritings from the input string to S using the rules of the grammar, and the translation is linear in the size of the input string times the number of rules. □

Since there exist fixed context-sensitive grammars for which the acceptance problem is NP-hard[9], the NP-hardness of FIXED DFSM RECOGNITION with nulls follows as a corollary.

9. Conclusion

We claimed that DFSM's provide an approach to rules that is likely to seem more natural and intuitive to phonologists. Bridging the gap between linguistically adequate formalisms and computationally useful formalisms is a long-term, community effort, and we feel that it would be premature to make claims about the linguistic adequacy of the approach; this depends on whether two-level approaches can be developed and deployed in a way that will satisfy the theoretical and explanatory needs of linguists. A specific claim on which our formalism depends is that all natural two-level phonologies can be reproduced using DFSM's with finitely encodable rules. We feel that this claim is plausible, but it needs to be tested in practice.

Computationally, our complexity work so far on DFSM's does not preclude the possibility that compilers for generation and recognition (without nulls) exist which will allow for polynomial-time behavior at run-time. Although this question must eventually be resolved, we feel that any implementation is likely to be simpler than that required for KIMMO, and that even a direct implementation of DFSM's can prove adequate in many circumstances. We have not constructed an implementation as yet.

Like other two-level approaches, we have a problem with surface nulls. It is possible in most realistic recognition applications to bound the number of nulls by some function on the length of the overt input; and it remains to be seen whether a reasonable bound could sufficiently improve complexity in these cases.

We have dealt with the problem of underlying nulls by simply ruling them out. This simplifies the formal situation considerably, but we do not believe that it is acceptable as a general solution; for instance, we can't expect all cases of epentheses to occur at morpheme boundaries. If underlying nulls are allowed, though, we will somehow need to limit the places where underlying nulls can occur; this is another good reason to pay attention to a phonotactic level of analysis.

[9] Garey and Johnson, (1979), p. 271.

10. Acknowledgements

This material is based upon work supported under a National Science Foundation Graduate Research Fellowship. This work was funded by National Science Foundation grant IRI-9003165. We thank the anonymous referees for helpful comments.

11. References

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

Edward Barton, Robert Berwick, and Eric Ristad. 1987. *Computational Complexity and Natural Language.* Cambridge, Massachusetts: MIT Press.

Gerhard Brewka. 1993. *Adding priorities and specificity to default logic.* DMG Technical Report. Sankt Augustin, Germany: Gesellschaft für Mathematik und Datenverarbeitung.

Mary Dalrymple, Ronald Kaplan, Lauri Karttunen, Kimmo Koskenniemi, Sami Shaio, and Michael Wescoat. 1987. *Tools for Morphological Analysis.* Stanford, California, 1987: CSLI Technical Report CSLI–87-108.

Michael Garey and David Johnson. 1979. *Computers and Intractability: A Guide to the Theory of NP-completeness.* New York, New York: Freeman and Co.

Lauri Karttunen. 1991. "Finite-state constraints." *International conference on current issues in computational linguistics.* Penang, Malaysia.

Lauri Karttunen and Kent Wittenburg. 1983. "A two-level morphological analysis of English." *Texas Linguistic Forum 22* pp. 217–228.

Graeme Ritchie, Graham Russell, Alan Black and Stephen Pulman. 1992. *Computational morphology.* Cambridge, Massachusetts: MIT Press.

Richard Sproat. 1992. *Morphology and computation.* Cambridge, Massachusetts: MIT Press.