Parallel Intersection and Serial Composition of Finite State Transducers

Mike REAPE[13] and Henry THOMPSON[123]

Centre for Cognitive Science[1]
University of Edinburgh
2 Buccleuch Place
Edinburgh EH8 9LW
Scotland

Department of Artificial Intelligence[2] and
Centre for Speech Technology Research[3]
University of Edinburgh
80 South Bridge
Edinburgh EH1 1HN
Scotland

Abstract

We describe a linguistically expressive and easy to implement parallel semantics for quasi-deterministic finite state transducers (FSTS) used as acceptors. Algorithms are given for determining acceptance of pairs of phoneme strings given a parallel suite of such transducers and for constructing the equivalent single transducer by parallel intersection. An algorithm for constructing the serial composition of a sequence of such transducers is also given. This algorithm can produce generally nondeterministic FSTS and an algorithm is presented for eliminating the unacceptable nondeterminism. Finally, the work is discussed in the context of other work on finite state transducers.

1. Introduction

Finite state transducers (FSTS) have been shown to be useful for modelling morphophonemic processes in an efficient way in (Karttunen 1983), (Kay 1983), (Kaplan and Kay 1985), (Karttunen, Koskenniemi and Kaplan 1987) and (Koskenniemi 1983) (but cf. (Barton 1986b), (Barton 1986a)). This paper presents a linguistically expressive parallel semantics for quasi-deterministic FSTS used as acceptors and algorithms for taking the parallel intersection and serial composition of such FSTS. The intersection and composition algorithms generate composite FSTS from sets of FSTS with the same semantics as the parallel semantics of the set. §2 presents the parallel semantics. §3 discusses the parallel intersection algorithm. §4 discusses the serial composition algorithm. §5 discusses the elimination of unacceptable general nondeterminism which can arise from the composition algorithm. §6 discusses the implementation of the interpreter which is based on the semantics presented here and the three algorithms. §7 discusses this research in the context of other work in this area and draws some conclusions.

2. A Parallel Semantics for Finite State Transducers

In the discussion that follows, we assume that the reader is familiar with the work of Karttunen and Koskenniemi on FSTS and with finite state automata (FSA) generally. The notation used is slightly different than that usually used to describe FSA but is more convenient for our purposes. Also, rather than discuss the algorithms directly, we give their semantics. In contrast to Karttunen and Koskenniemi's work, no higher level rule formalism is used. FSTS are stated directly.

An *FST*, M, is a pair $\langle N^{\alpha},\Sigma\rangle$ where N^{α} is a set of *start state* names and Σ is a set of *states*. A *state* $\Sigma_i \in \Sigma$ is a triple $\langle N,T,A\rangle$ where N is the name of the state, T is an ordered sequence of *transitions* T_i, $1\le i\le n$, $n = |T|$ and A is the truth value **T** if the state is an *accepting* state and the truth value **F** if it is a *nonaccepting* state. (The notion of final state is not relevant here. Only the accepting/nonaccepting distinction is important.) A *transition* $T_i \in T$ is a pair $\langle\Phi_i,N_i\rangle$ where Φ_i is a *transition pair* $\langle\phi^{\lambda}_i,\phi^{\rho}_i\rangle$. An element of a transition pair is either a *phoneme*, *a phoneme class name*, the symbol = or the *empty string* ε. A *phoneme* is a character and is a member of the alphabet set. A *phoneme class* is a set of phonemes. We will refer to phoneme classes and their names interchangeably. N_i is the *new state*. $\Phi_i = \langle\phi^{\lambda}_i,\phi^{\rho}_i\rangle$ *subsumes* $\Phi_t = \langle\phi^{\lambda}_t,\phi^{\rho}_t\rangle$ if ϕ^{λ}_i *subsumes* ϕ^{λ}_t and ϕ^{ρ}_i *subsumes* ϕ^{ρ}_t. ϕ_i subsumes ϕ_t if $\phi_i = \phi_t$ or $\phi_i = =$ or ϕ_i is a phoneme class and $\phi_t \in \phi_i$.

The *transition type* or *type* $\tau(\Phi)$ of a transition pair $\Phi = \langle\phi^{\lambda},\phi^{\rho}\rangle$ is $(x.x)'$ if both ϕ^{λ} and ϕ^{ρ} are phoneme classes and is $\tau(\phi^{\lambda}).\tau(\phi^{\rho})$ otherwise where $\tau(\phi)$ is the *phoneme type* of ϕ. (**x** is not a variable in this and the following definitions.)

$$\tau(\phi) = \begin{cases} \varepsilon & \text{if } \phi = \varepsilon \\ = & \text{if } \phi = = \\ x & \textit{otherwise} \end{cases}$$

The set of *types*, TYP, and the set of *final types*, TYP_{ω}, are defined below.

$$TYP = \{=.= \; x.= \; =.x \; x.\varepsilon \; \varepsilon.x \; x.x \; =.\varepsilon \; \varepsilon.= \; (x.\varepsilon)' \; (\varepsilon.x)' \; (x.x)'\}$$

$$TYP_{\omega} = \{(x.x)' \; x.x \; =.= \; =.x \; x.= \; x.\varepsilon \; \varepsilon.x \; \varepsilon.\varepsilon\}$$

Some examples should clarify the definitions. **<s,s>** is of type **x.x**. **<s,z>** is of type **x.x**. **<sib,sib>** is of type **(x.x)′** if **sib** is a phoneme class name. **<=,=>** is of type =.=. **<=,ε>** is of type =.ε.

The *type intersection* of a set of transition pairs $\{\Phi_i \mid 1 \le i \le n\}$ is $\bigcap_{i=1}^{n} \tau(\Phi_i)$ where $\cap_\tau$ is a partial function from pairs of transition types to transition types as defined below.

$$\tau_1 \cap_\tau \tau_2 = \begin{cases} \tau_1 \cap_\tau' \tau_2 & \text{if } \tau_1 \cap_\tau' \tau_2 \in TYP \\ undefined & otherwise \end{cases}$$

$\cap_\tau'$ is defined as follows.

(1) $\alpha.= \cap_\tau' =.\beta = (\alpha.\beta)'$
(2) $\alpha.= \cap_\tau' (\alpha.\beta)' = (\alpha.\beta)'$
(3) $=.\beta \cap_\tau' (\alpha.\beta)' = (\alpha.\beta)'$
(4) $\alpha.\beta \cap_\tau' (\alpha.\beta)' = \alpha.\beta$
(5) $\alpha.\beta \cap_\tau' \alpha.\beta = \alpha.\beta$
(6) $=.= \cap_\tau' \alpha.\beta = \alpha.\beta$
(7) $\alpha.= \cap_\tau' \alpha.\beta = \alpha.\beta$
(8) $=.\beta \cap_\tau' \alpha.\beta = \alpha.\beta$
(9) $\alpha.\beta \cap_\tau' \delta.\gamma = \delta.\gamma \cap_\tau' \alpha.\beta$

An unprimed type τ indicates that the transition type is *supported.* A primed type τ' indicates that the transition type is *unsupported.* That is, there have been no ε.x, x.ε or x.x types in the set of intersected types that produced the primed type. (1) is the origin of unsupported types. (2) and (3) state that neither $\alpha.=$ nor $=.\alpha$ can support a transition. (4) states that an unprimed type supports the corresponding primed type. (5) states that the intersection of two identical types is the same type. (6) states that the intersection of =.= and any type is that type. (7) and (8) state that the intersection of either $=.\alpha$ or $\alpha.=$ and a supported type is a supported type. (9) states that $\cap_\tau'$ is commutative and that the commutative closure of (1)-(8) also holds.

A set of transition pairs $\{\Phi_i\}$ which subsume Φ_t is *licensed* w.r.t. Φ_t if LICENSED($\{\Phi_i\},\Phi_t$) holds.

LICENSED($\{\Phi_i\},\Phi_t$) if
$\cap_\tau\tau(\Phi_i) \in$ TYP and
($\cap_\tau\tau(\Phi_i) \in$ {x.x x.ε ε.x} or
$\cap_\tau\tau(\Phi_i) \in$ {(x.x)', =.=, =.x x.=} and
$\Phi_t = <\phi,\phi>$)

This definition implements the "daisywheel". That is, although a set of transition pairs $\{\Phi_i\}$ is excluded in the general case if the type intersection of $\{\Phi_i\} \in$ {(x.x)', =.=, =.x x.=} we make an exception if Φ_t is a pair of identical phonemes. So, for example, if the type intersection of $\{\Phi_i\}$ is =.x and Φ_t = <s,s> then $\{\Phi_i\}$ is licensed. In practical terms, this means that the user does not need to encode a large set of "default" transition pairs of the form $<\phi,\phi>$ for each state. This effect is usually achieved in other FST formalisms in the rule compiler. However, such a compilation depends on the existence of an alphabet declaration. As we do not use a rule compiler, we have found it more convenient to build the effect into the parallel semantics.

A machine, M in state N *accepts* a phoneme pair Φ_t with *accepting transition pair* Φ and *new state* N' if ACCEPTS(M,N,Φ_t,Φ,N') holds.

ACCEPTS(M,N,Φ_t,Φ,N') if
M = $<N^\alpha,\Sigma>$ and
$\Sigma_i = <N,T,A> \in \Sigma$ and
$\exists T_k = <\Phi_k,N_k> \in$ P(T) ∋
Φ_k subsumes Φ_t and
$\nexists T_j = <\Phi_j,N_j> \in$ P(T) ∋
$1 \le j < k$ and Φ_j subsumes Φ_t.

(ACCEPTS replaces the more usual state transition function δ.)

P(T) is a total function that takes the transition sequence T as argument and returns a transition sequence T' containing the same set of elements as T with the following ordering of the elements of T'. All =.= transitions follow all non-=.= transitions. All $=.\alpha$ or $\alpha.=$ transitions precede all =.= transitions and follow all other transitions. Relative ordering of transitions in T' is as in T otherwise.

The definition above implies that transition precedence is by citation order with two exceptions. All transition pairs which have non-= first and second elements take precedence over any pairs of the form $<\alpha,=>$ and $<=,\alpha>$ and all non-<=,=> transition pairs take precedence over a transition pair of the form <=,=>.

A set of machines $\{M_i\}$ in states $\{N_i\}$ *accept* a phoneme pair Φ_t with accepting transitions pairs $\{\Phi_i\}$ and new states $\{N_i'\}$ if S-ACCEPTS($\{M_i\},\{N_i\},\Phi_t,\{N_i'\}$) holds.

S-ACCEPTS($\{M_i\},\{N_i\},\Phi_t,\{N_i'\}$) if
$\forall i \; \exists \Phi_i$
ACCEPTS($\{M_i\},\{N_i\},\Phi_t,\{\Phi_i\},\{N_i'\}$) and
LICENSED($\{\Phi_i\},\Phi_t$)).

A *string* is a sequence of phoneme pair elements. A *string pair* $<\mu,\nu>$ is a pair of strings μ and ν. $<\alpha,\beta>$ is a *prefix* of the string pair $<\mu,\nu>$ and the string pair $<\mu',\nu'>$ is the corresponding *suffix* of $<\mu,\nu>$ if CONCAT($<\alpha,\beta>,<\mu',\nu'>,<\mu,\nu>$) holds.

CONCAT($<\alpha,\beta>,<\mu',\nu'>,<\mu,\nu>$) if
$\mu = \alpha\mu'$ and
$\nu = \beta\nu'$ and
$\neg (\alpha = \varepsilon \wedge \beta = \varepsilon)$.

In particular, this means that prefixes can be of the schematic types x.x, x.ε and ε.x but not ε.ε.

A set of machines $\{M_i\}$ in states $\{N_i\}$ *accept* a string pair $<\mu,\nu>$ with new states $\{N_i'\}$ if STR-ACCEPTS($\{M_i\},\{N_i\},<\mu,\nu>,\{N_i'\}$) holds.

STR-ACCEPTS($\{M_i\},\{N_i\},<\varepsilon,\varepsilon>,\{N_i\}$).
STR-ACCEPTS($\{M_i\},\{N_i\},<\mu,\nu>,\{N_i'\}$) if
$\exists<\alpha,\beta> \; \exists<\mu',\nu'> \; \exists N_i'' \ni$
CONCAT($<\alpha,\beta>,<\mu',\nu'>,<\mu,\nu>$) and
S-ACCEPTS($\{M_i\},\{N_i\},<\alpha,\beta>,\{N_i''\}$) and
STR-ACCEPTS($\{M_i\},\{N_i''\},<\mu',\nu'>,\{N_i'\}$).

The following definition is the top-level relation of our semantics. A set of machines $\{M_i\}$ accepts a string pair $<\mu,\nu>$ if ACCEPTS($\{M_i\},<\mu,\nu>$) holds.

ACCEPTS($\{M_i\}$,<μ,ν>) if

$\forall M_i = \langle N^\alpha, \Sigma\rangle \in \{M_i\}$

$\exists N_i^\alpha \in N^\alpha$

$\exists \Sigma_i = \langle N_i, T, T\rangle \in \Sigma$ ∋

STR-ACCEPTS($\{M_i\}$,$\{N_i^\alpha\}$,<μ,ν>,$\{N_i\}$).

The reader may have noticed that there is no explicit declaration of the set of phonemes which define the alphabet of the FSTS. This is the reason that no mention was made of the alphabet in the definition of an FST above as is usually done for finite state machines. This complicates the algorithms to be discussed below a great deal. In particular, phoneme classes cannot in general be replaced by their definitions, the = notation cannot be compiled away nor can transition sequences be replaced by transition sequences in which ϕ^λ and ϕ^ρ are both phonemes for every transition pair $\Phi = \langle\phi^\lambda,\phi^\rho\rangle$. However, explicitly declaring the alphabet is unnecessary and a certain flexibility in the semantics of the FSTS is gained by not doing so.

3. The Parallel Intersection Algorithm

As (Karttunen and Wittenburg 1983) points out, it is possible to merge a set of parallel FSTS into one large FST. In the worst case, the number of states of the intersected FST is the product of the number of states of the intersected FSTS. In theory, this number can be very large. In practice, it is usually much smaller because the intersection of most state pairs is undefined.

Parallel intersection is associative and commutative. Thus, the following definition of the intersection of a sequence of FSTS is adequate

$$\cap\langle M_1 \cdots M_n\rangle = \bigcap_{i=1}^{n} M_i.$$

The intersection $M_1 \cap M_2$, of two FSTS

$$M_1 = \langle N_1^\alpha,\Sigma_1\rangle \text{ and } M_2 = \langle N_2^\alpha,\Sigma_2\rangle$$

is their cross product

$$\langle N_1^\alpha \times N_2^\alpha, \Sigma_1 \times \Sigma_2\rangle$$

The cross product of two state name sets $\{N_i' \mid 1\le i\le n\}$ and $\{N_j'' \mid 1\le j\le m\}$ is the set $\{\langle N_i',N_j''\rangle \mid 1\le i\le n \text{ and } 1\le j\le m\}$.

The intersection $\Sigma_{<1,2>} = \Sigma_1 \cap \Sigma_2$ of two states

$$\Sigma_1 = \langle N_1,T_1,A_1\rangle \text{ and } \Sigma_2 = \langle N_2,T_2,A_2\rangle$$

is

$$\Sigma_{<1,2>} = \langle\langle N_1,N_2\rangle, T_1 \times T_2, A_1 \wedge A_2\rangle.$$

I.e., the name of the intersection is the pair of the names of the two intersected states. The intersection is an accepting state if both of the intersected states are accepting states and is a nonaccepting state otherwise.

The cross product of two transition sequences T_1 and T_2 is a sequence $T_1 \times T_2 = \langle T',\le\rangle$ where T' is the set defined below and ≤ is a *total ordering*.

$$T' = \{T_k \mid T_i \in T_1 \text{ and } T_j \in T_2 \text{ and } T_k = T_i \cap T_j \text{ is defined}\}.$$

≤ can be any total ordering which satisfies the following partial ordering on T':

$\forall T_m \in T'$ ∋

$T_m = T_i \cap T_j$ and $T_i \in T_1$ and $T_j \in T_2$

$\forall T_n \in T'$ ∋

$T_n = T_o \cap T_p$ and $T_o \in T_1$ and $T_p \in T_2$

$(m < n \leftrightarrow$

$\neg (o < i \text{ and } p \le j) \text{ and } \neg (o \le i \text{ and } p < j))$

In particular, the ordering of the following sequence satisfies the partial order:

$$\langle T_{<1,1>} \cdots T_{<1,n>} \cdots T_{<m,1>} \cdots T_{<m,n>}\rangle$$

where $T_{<i,j>}$ names the intersection of the transitions $T_i \in T_1$ and $T_j \in T_2$, $m = |T_1|$ and $n = |T_2|$.

The intersection $T_i \cap T_j$ of two transitions $T_i = \langle\Phi_i,N_i\rangle$ and $T_j = \langle\Phi_j,N_j\rangle$ is $\langle\Phi_i \cap \Phi_j, \langle N_i,N_j\rangle\rangle$.

If $\Phi_i = \langle\alpha_i,\beta_i\rangle$ and $\Phi_j = \langle\alpha_j,\beta_j\rangle$ then $\Phi_i \cap \Phi_j$ is defined as follows

$$\Phi_i \cap \Phi_j = \begin{cases} \langle\alpha_i \cap \alpha_j, \beta_i \cap \beta_j\rangle & \text{if } \tau(\Phi_i) \cap_\tau' \tau(\Phi_j) \in TYP_\omega \\ \textit{undefined} & \textit{otherwise} \end{cases}$$

The intersection of two phoneme pair elements x and y is defined as follows

$$x \cap y = \begin{cases} x & \text{if } x = y \\ x & \text{if } y = = \\ y & \text{if } x = = \\ x & \text{if } y \textit{ is a phoneme class and } x \in y \\ y & \text{if } x \textit{ is a phoneme class and } y \in x \\ x \cap y & \text{if } \textit{both } x \text{ and } y \textit{ are phoneme classes} \\ \textit{undefined} & \textit{otherwise} \end{cases}$$

The composite FST is nondeterministic with respect to ε and the set of start states and is deterministic otherwise. All phoneme class and = notation is preserved in the intersected transitions. This is actually quite useful for debugging purposes. In general, it will often be the case that elements of an intersected transition sequence are subsumed by preceding elements in the same sequence. It is a simple matter to remove such transitions (although this is not necessary as they are unreachable). Furthermore, it is often the case that transitions with phoneme classes are partially subsumed by preceding elements in the same transition sequence. It is straightforward to split the phoneme class transitions into disjoint phoneme class transitions which are not subsumed by preceding transitions in the same sequence. Our implementation uses both of these optimisations.

Notice that the intersection algorithm does not "compile in" the effect of the daisywheel. This is because the semantics of a set of parallel FSTS includes the daisywheel and so the composite FST need not have its effect "compiled in". Furthermore, the intersection algorithm must not build in the daisywheel because the composite FST would have the wrong parallel semantics

and could not be correctly used as input to the intersection algorithm. (I.e., we cannot eliminate = or phoneme classes from any transition pairs.)

4. The Serial Composition Algorithm

Just as parallel FSTS can be intersected, a cascade of FSTS may be composed into one FST. Such a cascade is most useful for representing ordered sequences of rules. For example, a theory which orders assimilation processes before morphophonemic processes could be modelled by a cascade of two parallel sequences of transducers where the first parallel sequence models the assimilation processes and the second models the morphophonemic processes. As is the case with parallel intersection, the number of states of a composed FST is the product of the number of states of the composed FSTS in the worst case. Again, the number of states in the composed FST is usually much smaller in practice.

Serial composition is different in several ways from the parallel intersection problem. First, each FST in the composition must have the parallel semantics of §2 "compiled in" before it is composed. This means that type intersection as defined for parallel intersection is irrelevant for composition. On the other hand, we must include the effect of the daisywheel before composition on any transition pair $<\phi^{\lambda},\phi^{\rho}>$ where both ϕ^{λ} and ϕ^{ρ} are phoneme classes. As a result, we can replace all such transitions with one or more transitions $<\phi^{\lambda}, \phi^{\rho\prime}>$ where ϕ^{λ} and $\phi^{\rho\prime}$ are both phonemes. This simplifies the composition algorithm considerably. However, we must still check that the type of each transition pair in each FST to be composed is an element of TYP_{ω}. (In particular, users may encode illegal transitions.) Also, although serial composition is associative, unlike parallel intersection, it is not commutative. So, a cascade of FSTS must be composed in the same order as they appear in the cascade.

The composition of a sequence of FSTS $*<M_1 \cdots M_n>$ is defined by

$$*<M_1 \cdots M_n> = \begin{cases} M_n & \text{if } n=1 \\ *<M_1 \cdots M_{n-1}> * M_n & \text{if } n>1 \end{cases}$$

The composition $M_1 * M_2$ of two FSTS

$$M_1 = <N_1^{\alpha},\Sigma_1> \text{ and } M_2 = <N_2^{\alpha},\Sigma_2>$$

is their cross product

$$<N_1^{\alpha} \times N_2^{\alpha}, \Sigma_1 \times \Sigma_2>$$

The composition $\Sigma_{<1,2>} = \Sigma_1 * \Sigma_2$ of two states

$$\Sigma_1 = <N_1,T_1,A_1> \text{ and } \Sigma_2 = <N_2,T_2,A_2>$$

is

$$\Sigma_{<1,2>} = <<N_1,N_2>, T_1 \times T_2, A_1 \Lambda A_2>$$

I.e., the name of the composition is the pair of the names of the two composed states. The composition is an accepting state if both of the composed states are accepting states and is a nonaccepting state otherwise.

The cross product of two transition sequences T_1 and T_2 is a sequence $T_1 \times T_2 = <T',\leq>$ where T' is defined below and $\leq$ is a *total ordering*.

$$T' = \{T_k \mid T_i \in T_1 \text{ and } T_j \in T_2 \text{ and } T_k = T_i * T_j \text{ is defined}\}.$$

$\leq$ must satisfy the same partial ordering as that given for parallel intersection (modulo the substitution of * for $\cap$). Again, we use the ordering given in §3.

If $\Sigma_i = <N_i',T_1,A_i>$ and $\Sigma_j = <N_j',T_2,A_j>$ and $T_i \in T_1$ and $T_j \in T_2$ then the composition $T_i * T_j$ of two transitions $T_i = <\Phi_i,N_i>$ and $T_j = <\Phi_j,N_j>$ is defined by

$$T_i * T_j = \begin{cases} <<=,=>,<N_i,N_j>> & \text{if } \Phi_i = <=,=> \text{ and } \Phi_j = <=,=> \\ <<\alpha,\beta>,<N_i,N_j>> & \text{if } \Phi_i = <=,=> \text{ and } \Phi_j = <\alpha,\beta> \text{ and } <<\alpha;\alpha>,N_k> \notin T_1 \ni k<i \\ <<\alpha,\beta>,<N_i,N_j>> & \text{if } \Phi_i = <\alpha,\beta> \text{ and } \Phi_j = <=,=> \text{ and } <<\beta,\beta>,N_k> \notin T_2 \ni k<j \\ <<\alpha,\varepsilon>,<N_i,N_j'>> & \text{if } \Phi_i = <\alpha,\varepsilon> \\ <<\varepsilon,\beta>,<N_i',N_j>> & \text{if } \Phi_j = <\varepsilon,\beta> \\ <<\alpha,\delta>,<N_i,N_j>> & \text{if } \Phi_i = <\alpha,\beta> \text{ and } \Phi_j = <\beta,\delta> \\ \textit{undefined} & \textit{otherwise} \end{cases}$$

(The fourth and fifth clauses are due to Martin Kay (Kay 1983).)

Note that if $\Phi_i = <\alpha,\varepsilon>$ and $\Phi_j = <\varepsilon,\beta>$ then both $<<\alpha,\varepsilon>,<N_i,N_j'>>$ and $<<\varepsilon,\beta>,<N_i',N_j>>$ are defined. Their order relative to each other is irrelevant since the semantics is nondeterministic with respect to ε transitions. Also, note that the second and third clauses dealing with <=,=> transitions are further constrained to eliminate any "instantiation" of <=,=> which has lower precedence than a transition with the "instantiation" in the transition sequence which contains <=,=>. E.g., if $<<=,=>,N_j> \in T_1$ and $<<=,=>,N_j> * <<b,c>,N_k> = <<b,c>,<N_j,N_k>>$ and there is a transition $<<b,b>,N_i> \in T_1$ and i<j then <<b,b>,N_i> takes precedence over <<=,=>,N_j> and so the composition is undefined.

Finally, note that nondeterministic transition sequences may be defined. That is, two or more transitions with the same transition pair may be specified which have different new states. E.g., the composition of the transitions <<a,b>,s1> and <<b,c>,t1> is <<a,c>,<s1,t1>> but the composition of the transitions <<a,d>,s2> and <<d,c>,t2> is <<a,c>,<s2,t2>>. Both compositions have the transition pair <a,c> but the new state is the <s1,t1> for the first transition and <s2,t2> for the second transition. This form of nondeterminism is genuine and must be eliminated if the quasi-deterministic semantics that we have outlined is to be maintained.

5. The Determinisation Algorithm

As (Barton 1986b) points out, FSTS used as acceptors are finite-state machines (FSM) with an alphabet of pairs of characters. As such, an equivalent deterministic FST can be constructed for any nondeterministic FST used as an acceptor since a deterministic FSM can always be constructed that accepts exactly the same language as a nondeterministic FSM (Hopcroft and Ullman 1979). Because the serial composition algorithm may produce nondeterministic FSTS, a determinisation algorithm is required to produce equivalent deterministic FSTS.

The algorithm collapses all transitions in a transition sequence with common transition pairs but different *new states* into one transition with a complex *new state* name. This *new state* name is the name of a state which is the parallel intersection of all the *new states* of the transitions with the common transition pairs. The only fundamental difference between this type of parallel intersection and the definition presented in § 3 is that a state in the intersected FST is an accepting state if *any* of the intersected states is an accepting state.

Although it may not be obvious, the determinisation algorithm is guaranteed to terminate. The following argument shows why. The *new states* of simple states are always simple states so complex states are the intersection of only simple states. The number of simple states is finite. The number of transitions within a simple state is finite. It follows that the number of transitions in a transition sequence with common transition pairs is bounded, the number of possible complex states is bounded and the size of a complex state is bounded. Therefore, there is an upper bound on the size of the equivalent deterministic machine and so the determinisation algorithm is guaranteed to terminate.

6. Implementation

The second author designed the parallel semantics and implemented an interpreter for it in Interlisp-D on a Xerox 1186. The first author designed and implemented the parallel intersection, serial composition and determinisation algorithms in Lucid Common Lisp on a Masscomp MC5700. The programs exhibit reasonable performance (about ten minutes using compiled Lisp for composite FSTS with approximately 160 states).

7. Conclusions and Related Work

Although it has been reported in the literature that the algorithms described here have been implemented, we are unaware of the publication of any such algorithms to date. The algorithms themselves are of interest because they formalise the semantics of finite state transducers. Also, these algorithms are similar to graph unification algorithms. Specifically, the parallel intersection and determinisation algorithms can be viewed as cyclic graph unification and graph disjunction elimination algorithms respectively.

As Barton points out, a determinisation algorithm like the one presented here will not work on transducers used for generation and recognition (as opposed to simple acceptance). He claims that many FSTS are not determinisable at all. The current work provides a formal basis on which to investigate the class of determinisable transducers used for generation and recognition.

8. Acknowledgments

This research was supported by the Alvey Speech Input Word Processor and Workstation Large Scale Demonstrator project, ESRC Grants D/29611, D/29628 and D/29604. The first author has been supported during the writing of this paper by the EEC Esprit Project 393 *ACORD*: the Construction and Interrogation of Knowledge Bases using Natural Language Text and Graphics.

9. References

Barton, G. E. (1986) Constraint Propagation in Kimmo Systems. In *Proceedings of the 24th Annual Meeting of the Association for Computational Linguistics*, Columbia University, New York, N.Y., June, 1986, pp45-52.

Barton, G. E. (1986) Computational Complexity in Two-Level Morphology. In *Proceedings of the 24th Annual Meeting of the Association for Computational Linguistics*, Columbia University, New York, N.Y., June, 1986, pp53-59.

Hopcroft, J. and Ullman, J. D. (1979) *Introduction to Automata Theory, Languages and Computation*. Reading, Mass.: Addison-Wesley.

Kaplan, R. and Kay, M. (1985) Phonological rules and finite-state transducers.

Karttunen, L. (1983) KIMMO: A general morphological processor. *Texas Linguistic Forum*, **22**, 165-186.

Karttunen, L. and Wittenburg, K. (1983) A two-level morphological analysis of English. *Texas Linguistic Forum*, **22**, 217-228.

Karttunen, L., Koskenniemi, K. and Kaplan, R. (1987) A Compiler for Two-level Phonological Rules. Technical Report, Center for the Study of Language and Information, Stanford University, 1987.

Kay, M. (1982) When meta-rules are not meta-rules. In Sparck-Jones, K. and Wilks, Y. (eds.) *Automatic Natural Language Parsing*, pp74-97. Chichester: Ellis Horwood. Also in M Barlow, D Flickinger and I A Sag (eds.) *Developments in Generalized Phrase Structure Grammar: Stanford Working Papers in Grammatical Theory*, Volume 2, pp69-91. Bloomington: Indiana University Linguistics Club.

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