The Delta Rule Development System for Speech Synthesis from Text

SUSAN R. HERTZ, JAMES KADIN, AND KEVIN J. KARPLUS, MEMBER, IEEE

Invited paper

Progress in speech synthesis has been hampered by the lack of rule-writing tools of sufficient flexibility and power. This paper presents a new system, Delta, that gives linguists and programmers a versatile rule language and friendly debugging environment. Delta's central data structure is well-suited for representing a broad class of multi-level utterance structures. The Delta language has flexible pattern-matching expressions, control structures, and utterance manipulation statements. Its dictionary facilities provide elegant exception handling. The interactive symbolic debugger speeds rule development and tuning. Delta can not only accommodate existing synthesis models, but can also be used to develop new ones.

I. Introduction

With the continued trend toward friendlier man–machine interfaces, the use of synthetic speech is increasing rapidly. Limited-vocabulary devices, such as talking toys, cars, and clocks, have long been on the marketplace. With recent advances in computer technology, however, unlimited-vocabulary systems, capable of saying almost anything, are becoming more and more prevalent. They are being used in telephone message services and order systems, remote-access database services, automated factories, computerized dispatching systems, reading machines for the blind, speaking machines for the speech-impaired, and other practical applications. Unlimited vocabulary systems are also being used as research tools to increase our understanding about speech.

Because the number of possible utterances is so large, an unlimited-vocabulary system cannot store an encoded form of each utterance to be produced, as can a limited-vocabulary system. Rather, it must employ a set of *rules* that can be applied to any input text to produce the corresponding speech.

Manuscript received December 6, 1984; revised July 2, 1985.

S. R. Hertz is with Eloquent Technology, Ithaca, NY 14850, and with the Phonetics Laboratory, Cornell University, Ithaca, NY 14853, USA.

J. Kadin is with the Computer Science Department, Cornell University, Ithaca, NY 14853, and with Eloquent Technology, Ithaca, NY 14850, USA.

K. J. Karplus is with the Computer Science and Electrical Engineering Departments, Cornell University, Ithaca, NY 14853, and with Eloquent Technology, Ithaca, NY 14850, USA.

A. *Previous Rule Models*

Almost all previous text-to-speech systems have grouped their rules into two main sets. The first set converts the input text into a linear string of phonetic units (for example, phonemes). The second set uses the information in the phonetic string to produce values for a speech synthesizer.

Although existing text-to-speech systems all share these two basic rule components, the strategies used within the two components differ widely. They differ in the kinds of units on which the synthesis is based, in the balance maintained between rules and dictionary lookups, in the way the vocal tract is modeled (e.g., in articulatory or acoustic terms), in the kinds of information they predict (e.g., syntactic, morphological, or purely phonological), and more generally, in the linguistic premises on which they are based.

In the first rule component, systems for English differ in whether they predict a word's pronunciation on the basis of the entire word or on the basis of the word's component pieces (morphs). For example, both the MITalk System [1], [2] and the SRS synthesis rules for English [3], [4] break a word into morphs (e.g., *liking* into "like" + "ing"). The NRL rules [5], in contrast, perform no morphological analysis at all, attempting to convert text to phonemes by simple rewrite rules that generate phonemes for particular letters in particular contexts.

Within systems that analyze words into morphs, one again finds different strategies. For example, the MITalk system breaks words into morphs via an extensive morph dictionary with about 12 000 entries, and it generates pronunciations primarily on the basis of morph pronunciations extracted from the dictionary. The SRS rules for English, on the other hand, predict morphs with a set of about 200 context-dependent rules that insert morphological markers into the input text, and they generate pronunciations primarily by a second set of rules that apply to the annotated text string produced by the first set. Furthermore, the MITalk system respects etymological origins, distinguishing *briber* ("bribe" + "er") from *fiber* ("fiber"), while the SRS rules for English do not, generating such "spelling morphs" as "fibe" + "er" for *fiber.*

In the second rule component as well, synthesis al-

0018-9219/85/1100-1589$01.00 ©1985 IEEE

gorithms vary widely, ranging from those that produce synthesizer parameter values by concatenating pre-stored acoustic information to those that produce the values entirely by rule. The demisyllable scheme, for example, concatenates the acoustic information pre-stored for the various demisyllables ("half syllables"), using rules only for smoothing between demisyllables and for adjusting the duration and pitch patterns [6] – [8]. Most phoneme-based schemes, on the other hand, generate the synthesizer values entirely by rule, by interpolating each synthesizer parameter between target values set for each phoneme segment [1], [9] – [11]. The diphone scheme is an intermediate approach between these two extremes that uses pre-stored transitions between adjacent phoneme-sized segments [12], [13].

A primary factor in determining the choice of synthesis strategy is the rule-writer's linguistic convictions. For example, the proponents of an approach based on demisyllables claim that many of the influences of adjacent sounds on each other are automatically present in the demisyllables and cannot easily be captured with rules that operate on smaller units. The opponents of this approach counter that the degree to which sounds influence each other depends on the overall context (e.g., stressed versus unstressed), and therefore, a particular demisyllable in one context is not necessarily appropriate for another context.

Practical considerations, such as memory size, flexibility, and ease of implementation also play a role in choosing the synthesis strategy. Concatenative approaches based on pre-stored units generally take more space than approaches based strictly on rules. Furthermore, such concatenative schemes are less flexible than rule-based ones, because the acoustic information in the concatenative units is originally extracted (in a tedious process) from a single speaker, making it difficult to change voice qualities and to notice generalizations that may apply from language to language. A concatenative approach, on the other hand, frees the rule-writer from having to predict the spectral information that is already present in the pre-stored units.

B. *Previous Rule Development Systems*

Although there are many synthesis models to choose from, the lack of appropriate rule-writing tools has made developing good rule sets difficult. Most text-to-speech programs are written in general-purpose programming languages, burying the rules in the form of tables and code and thereby making the rules difficult to test and modify [1], [8] – [13].

Two systems, the one by Carlson and Granström [14], and the one by Hertz (SRS) [15], have attempted to overcome this inflexibility by using explicit rules, expressed in an easy-to-read and easy-to-modify linguistic rule notation. SRS has the added advantage of being highly interactive, making rule entry, testing, comparison, and modification especially easy. A major advantage of these systems is the speed with which rules can be developed with them. The high-quality SRS synthesis rules for Japanese [16], [17], for example, are the result of only six months of work.

Despite their considerable flexibility compared to other synthesis systems, these two rule development systems are still much too restrictive to warrant their reputation as universal rule-writing tools. Although neither dictates what the actual rules should do, each does dictate a particular rule framework that may not be equally appropriate for describing all languages or even for describing all phenomena in a particular language.

For example, SRS forces rule-writers to work with four kinds of rules — text-modification, conversion, feature-modification, and parameter rules — that must always apply in that order. Within each rule level, the kinds of statements that can be made and the order of rule application is also predetermined. For example, at the parameter rule level, rule-writers are forced to adopt a target-and-transition model (in which target positions, associated values, and intervening transition shapes are defined for the various synthesizer parameters). The parameter rules are always tested top to bottom through the rule set and from left to right across the utterance.

Perhaps the most serious weakness of the two rule-development systems, however, is that they are restricted to operating on linear utterance representations. The systems cannot manipulate larger units like phrases or syllables as single units in parallel with smaller units like phonemes. Likewise, they cannot manipulate as independent units pieces that are smaller than phonemes, such as the aspiration portion of a stop.

C. *The Delta System*

The Delta System (so-named because its central data structure consists of connected streams much like the streams at the mouth of a river) is a synthesis rule-writing tool that overcomes many of the limitations of previous synthesis systems. It builds on nine years of experience synthesizing a variety of languages with SRS [18], providing the facilities that have been deemed necessary on the basis of that experience.

The Delta System provides a powerful, high-level programming language called Delta for expressing synthesis rules. With Delta, rule-writers can easily define and manipulate a broad class of utterance representations, ranging from linear strings to multi-level structures. They can supplement their rules wherever necessary with pre-stored information of any kind (for example — morphological, phonetic, symbolic, acoustic, phoneme-based, and demisyllable-based). They can combine the most favorable aspects of different synthesis strategies, not being forced into the "all-or-none" frameworks of other systems. Finally, they can concentrate on linguistic issues without being bogged down in the details of general-purpose programming languages.

The Delta System can produce compact, efficient, and portable rule sets, by compiling Delta programs (rules) into instructions for a special pseudo-machine called a *Delta Machine*. (In contrast, SRS rules are interpreted, and Carlson and Granström's rules are compiled into instructions for a particular computer.)

Fig. 1 shows a block diagram of the Delta System components, all of which are written in C.[1] The compiler operates on the Delta source code, producing assembly language

[1]Computational linguists often develop systems in programming languages built on top of languages like LISP and PROLOG. For example, the relatively new synthesis rule development system SYNTHEX [19], is implemented in LISLOG. (We do not yet know enough about this system to evaluate it in this paper.) We chose C for speed, portability, and compactness.

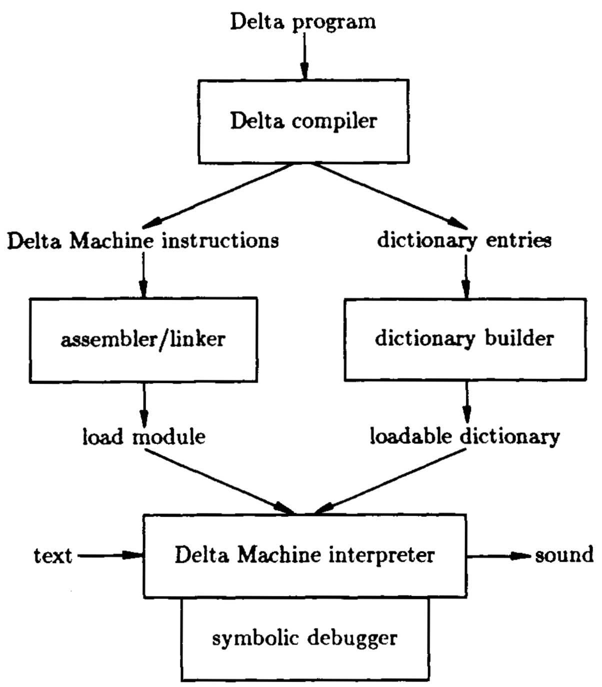

Fig. 1. Overview of the Delta System.

instructions for the Delta Machine and a separate set of dictionary entries.

The assembler/linker takes the assembly language instructions as input and produces a load module (an executable program). In addition to performing the usual linking functions, the linker can "patch" new or modified rules into existing load modules, so that rule changes can be tested quickly and easily.

The dictionary builder takes the dictionary entries and produces a loadable dictionary (one that can be searched at execution time by the dictionary instructions in the load module).

The interpreter/debugger executes the load module. The debugger is an interactive facility for testing and controlling Delta programs. Among other things, the debugger allows rule-writers to display selected portions of the utterance data structure, to trace the execution of selected rules, to display selected variables, and to temporarily stop execution of a program at almost any point.

In general, the Delta System has been designed to accommodate a broad range of synthesis schemes, to be equally suitable for synthesizing any human language, to provide for easy rule development and testing, and to produce compact, fast, and portable rules.

The remainder of this paper presents the current version of the Delta System from the user's point of view, focusing on the flexible source language and the powerful debugger. A final section describes briefly the numeric capabilities being designed for the next version of the system.

II. The Delta Language

A. *The Delta Data Structure*

The Delta language has been designed to create, test, and manipulate a data structure called a *delta*. A delta consists of one or more user-defined *streams* that are synchronized with each other at strategic points. For example, a Delta program might build up a delta for the word *bathed* that has the structure shown in Fig. 2.

`%morph:`	root					suffix
`%letter:`	b	a	t	h	e	d
`%phoneme:`	b	GAP	dh		GAP	d

Fig. 2. Delta fragment for *bathed.*

This delta has three streams named **%morph, %text,** and **%phoneme.** (To simplify parsing, stream names always begin with a percent sign.) Each stream contains a sequence of *tokens*. For example, the morph stream has two tokens named **root** and **suffix,** and the text stream has six tokens named **b, a, t, h, e,** and **d**. The **GAP** tokens in the phoneme stream function as placeholders for phonemes to be assigned by later rules.

The vertical bars that separate the tokens in each stream are called *sync marks*, and are used to synchronize tokens across streams. In the delta for *bathed*, for example, the sync marks synchronize the letters **b,a,t,h,e** in the text stream with a **root** token in the morph stream, and the final letter **d** with a **suffix** token. Rules can test a single stream or look for a particular cross-stream synchronization equally easily (see Section II-B).

Each token in a stream is a collection of fields with particular values. All tokens have at least a name field, and can be defined to have other fields. All tokens in a particular stream have the same fields. For example, each token in the text stream might have a field called **character_type** with possible values ⟨**letter**⟩, ⟨**digit**⟩, ⟨**punct**⟩ ("punctuation"), or ⟨**space**⟩. Fields and their possible values are defined in a *stream definition*.

The text stream definition in Fig. 3, for example, first defines the name field, with letters, punctuation marks,

```
stream %text;
   name: a, b, c, d, e, f, g, h, i, j, k, l, m,
         n, o, p, q, r, s, t, u, v, w, x, y, z,
         '.', ',', '?', '!', ' ', '1', '2', '3',
         '4', '5', '6', '7', '8', '9', '0';
   character_type: letter, punct, space, digit;
   letter_type:    vowel;
                 :
end %text;
```
Fig. 3. Sample text stream definition.

spaces, and digits as possible values. Then it defines the field **character_type**, with possible values ⟨**letter**⟩, ⟨**punct**⟩, ⟨**space**⟩, and ⟨**digit**⟩. Finally, it defines the field **letter_type**. Since only one value, ⟨**vowel**⟩, is given, the opposite value, ⟨~**vowel**⟩, is defined implicitly.

The text stream definition illustrates the three kinds of fields that tokens can have: *name-valued, multi-valued,* and *binary.* A name-valued field, such as the field **name**, takes token names in some stream as values; a multi-valued field, such as **character_type**, is a non-name-valued field with more than two possible values; and a binary field, such as **letter_type**, is a non-name-valued field with exactly two possible values.

A field together with one of its possible values is called an *attribute*. A stream definition can associate a set of *initial attributes* with a particular token name. The initial attributes are the field-values that are automatically set whenever a token with that name is inserted into the delta. The phoneme stream definition in Fig. 4 shows two ways in which initial attributes can be defined.

```
stream %phoneme;
   name:
      b, p, d, t, g, k, dh, th, f, v, s, z, sh,
      zh, ch, jh, l, r, y, w, i, I, ae, ... , uh;
   voicing:
      voiced;
   phoneme_class:
      consonant(b-r), vowel(i-uh), glide(y,w);
   manner_of_articulation:
      stop, fricative, affricate, ... ;
   place_of_articulation:
      labial, interdental, alveolar, palatal, ... ;
              ⋮
   th has  manner_of_articulation: fricative,
           place_of_articulation:  interdental;
   dh like th except voicing: voiced;
   ch has  affricate, palatal;
   jh like ch except voiced;
              ⋮
end %phoneme;
```

Fig. 4. Sample phoneme stream definition.

For the field **phoneme_class**, each field-value is followed by a parenthesized list of token names that have that value. For other fields, the initial attributes are assigned to each token name by a sequence of **has**- and **like**-statements that follow the definitions of the fields themselves. The first **has**-statement, for example, assigns the initial attributes for **th**. The first **like**-statement then gives **dh** the same initial attributes as **th**, except that the field **voicing** gets the value ⟨**voiced**⟩. (The token name **th** receives the attribute ⟨**˜voiced**⟩ by default.) The token names **jh** and **ch** are treated similarly, except that the field names for the attributes are not given. An attribute can always be used without the field name when the field-value alone is unambiguous.

The selection of stream types and attributes is left entirely to the rule-writer. For example, rule-writers could use a demisyllable stream, a phoneme stream, or both. Similarly, they could use streams representing articulatory parameters such as the tongue tip movement and degree of labial constriction, or they could use streams representing acoustic parameters such as formants.

Multi-Leveled Deltas versus Linear Structures: Multi-leveled deltas have several advantages over the linear representations central to previous synthesis systems. The first and most obvious advantage is clarity. Consider the delta for the word *discovers* shown in Fig. 5.[2] This representation not only makes the utterance structure clear (for example, the different prefix and first-syllable boundaries), but allows the relevant units to be tested and manipulated straightforwardly. For example, a Delta program can easily find the irregular pronunciation for the root *cover*, by looking up in the dictionary the letters synchronized with the root token. Similarly, a Delta pattern can easily determine that the phoneme [k] of the root *cover* does not begin a syllable and should therefore not be aspirated [20]. (Compare the [k] of *discomfort*, which for most speakers begins a syllable and is aspirated. See Section II-C for a discussion of how the different syllable boundaries in words like *discover* and *discomfort* can be determined.)

%morph:	prefix			root					suffix
%text:	d	i	s	c	o	v	e	r	s
%phoneme:	d	I	s	k	uh	v	E	r	z
%syllable:	syl		syl			syl			

Fig. 5. Delta fragment for *discovers*.

The clarity of the Delta notation is especially evident when contrasted with the cluttered way in which the same information would be represented in other systems. For example, after SRS text modification (the first stage of SRS text-to-phoneme conversion), the word *discovers* would be written as follows:

$$+ + \mathrm{d\,I\,.\,s} - + \mathrm{c\,o\,v\,e\,r} + - \mathrm{s} + +$$

(Only unpredictable syllable boundaries — in this case, the first one — are marked at the text-modification level.) In other kinds of words, stress-related information would be interspersed as well, and in sentences, phrasal information would be included. The abundance of special markers not only obscures the utterance structure, but complicates the rules, since they cannot test and manipulate the higher level units directly.

A second advantage of multi-leveled deltas over linear representations is that they can synchronize noncontiguous tokens at one level with a single token at another level. Consider Semitic languages, where the consonants without the intervening vowels comprise the root. For example, the Hebrew word *kotev* is the present-tense (masculine) form of "write," whereas *katuv* is the passive participle "written." The word *kotev* can be represented in delta form as shown in Fig. 6. The word is represented from right to left in accordance with Hebrew writing convention. (Rules can scan the delta equally well in either direction.)

The consonant stream has two adjacent sync marks that have no intervening token; such sync marks act like a single sync mark. The tokens between them in other streams — in this case **o** and **e** in the vowel stream — can be thought of as being "invisible" in the stream with the adjacent sync

GAP	e	GAP	o	GAP	**:%vowel**
v		t		k	**:%consonant**
		root			**:%morph**

Fig. 6. Delta fragment for Hebrew *kotev* ("write").

[2]This delta represents an intermediate stage of rule application. It is assumed that the phonemes **I** and **E** would be reduced to the appropriate unstressed vowels by later rules.
Many linguists would argue that the **v** of *discover* actually belongs to two syllables. Ambisyllabicity can be represented with the delta data structure, but is not illustrated in this paper, since the additional complexity of the resulting structure would only obscure the points being made.

marks. This representation permits straightforward manipulation of the three consonants as a unit. For example, to look them up in the dictionary, the program would simply look up the consonant sequence synchronized with the root token in the morph stream.

A third advantage of multi-level deltas over linear structures is that the derivational history of all units is available to the rules. Having earlier stages available eliminates the need to carry special markers or attributes through a derivation when reference must be made back to those stages. For example, in some English dialects, the stop [t] is inserted between a tautosyllabic [n] and [s]. The words *sense* and *prince* would be pronounced with the same sequence of segments as the words *cents* and *prints*. However, according to a recent study [21], the inserted stop of a word like *sense* is not phonetically identical to the inherent stop of a word like *cents*; its closure duration is shorter.

With deltas in which letters are synchronized with phonemes, the conditioning factors for the closure durations of inserted versus inherent stops can be stated directly, as shown by the deltas in Fig. 7 for *sense* and *cents*.[3]

```
sense:
        %text:    | s | e | n |   | s | e |
        %phoneme: | s | E | n | t | s |   |
cents:
        %text:    | c | e | n | t | s |
        %phoneme: | s | E | n | t | s |
```

Fig. 7. Delta fragments for *sense* and *cents*.

In the delta for *sense*, the inserted stop is not synchronized with a letter, and can therefore be easily differentiated from the inherent stop of *cents*, which is synchronized with a letter. (Note that the inserted stop is invisible — that is, not synchronized with a gap — in the text stream, so that a test for the letter sequence **ns** would succeed despite the intrusion of the stop in the phoneme stream.)

A fourth advantage of multi-level deltas over linear structures is that the basic synthesis units (say, phonemes) can be easily subdivided into smaller pieces. Such sub-pieces simplify the description of complex segments, such as pre-nasalized and post-nasalized stops ([mb] and [bm]) [22], diphthongs ([ay] in *chide*), and affricates ([ch] in *chide*), which behave as single units for some purposes and as two units for others.

Consider the delta shown in Fig. 8 for the word *chide*. The phoneme-sized units **ch, ay,** and **d** might be used to predict durations. Their subparts, on the other hand, might be used to predict formant transitions, since the formant transitions from the preceding segment into [ay] will be much like those into the monophthong [a], but those from [ay] into the following sound, will be more like those from [y].[4]

```
%phoneme: |   ch   |  ay   | d |
%sub_phon: | t | sh | a | y | d |
```

Fig. 8. Delta fragment for *chide*.

Multi-level deltas can also represent low-level acoustic phenomena that do not belong to any single higher level segment. For example, there has been debate over whether aspiration is best treated for synthesis purposes as part of the stop or as part of the following vowel. Linear representations force it to be treated as one or the other. Delta allows a third (perhaps phonetically more accurate) alternative — namely, to associate it exclusively with neither, as shown in the delta in Fig. 9 for the word *pie*.

```
%letter:          |    p    |      i     |    e    |
%phoneme:         |    p    |           ay         |
%amplitude_type:  | silence | aspiration | voicing |
```

Fig. 9. Delta fragment for *pie*.

B. Rules

Synthesis rules (and linguistic rules in general) usually test for the occurrence of a *pattern* and perform an *action* if it occurs. For example, a phoneme-predicting rule for English might test for a letter *a* preceding a consonant and a root-final letter *e* (as in *shave* and *wade*), and generate the phoneme [e] for it. The Delta language is specifically designed to test patterns against a delta and perform actions on the delta when the patterns match.

The syntax of Delta's rules borrows as heavily from existing computer programming languages as it does from the notations of linguists. Although multi-level utterance representations are central to current research in phonological theory [23], linguists have not yet developed a formalism for testing and manipulating these structures that can be implemented easily on a computer.

Patterns: Delta patterns can test for the occurrence of particular sync marks and particular tokens in relation to those sync marks. Sync marks are referred to in the rules by means of *pointer variables.*

Assume, for example, that a pointer named ^1 has been set at the sync mark before the letter **a** in the sample delta for *bathed*, as shown in Fig. 10. The sync mark can then be

```
%morph:   |             root              | suffix |
%letter:  | b |  a  | t | h |  e   |   d    |
%phoneme: | b | GAP |  dh   | GAP  |   d    |
             ^1
```

Fig. 10. Delta fragment for *bathed* with pointer.

[3]The term "phoneme" is used loosely in this and other examples to mean "phoneme-sized segment." An epenthetic (inserted) stop is not a phoneme in strict linguistic usage of the term. A more complicated, but linguistically more appealing way to handle the epenthetic stop would be to insert it at a level (stream) below the phoneme stream. The duration rules could then distinguish the inserted stop from an inherent one by testing whether it is synchronized with (i.e., is derived from) a phoneme.

[4]Even segments not thought of by most linguists as complex segments often have asymmetrical properties that could be handled straightforwardly by subdividing the segments. For example, the [k]s of words like *pokey* and *okay* could be divided into a velar portion and a palatal portion in order to generate the velar-like formant transitions into the [k]s and the palatal-like transitions out of them.

referred to in patterns such as the following:

[%text _^1 a]

The square brackets enclose a stream identification (here, the text stream) and a pattern to be tested in that stream. The underscore followed by a pointer name is the pattern's *anchor*, and specifies the sync mark at which testing of the pattern begins. The simple pattern above starts at the sync mark ^1 and tests whether the following token in the text stream is named **a**.

Rather then testing for the letter **a** alone, a pattern could test for an occurrence of one of the letters **a**, **e**, **i**, **o**, or **u**, as shown in the following pattern:

[%text _^1 {a | e | i | o | u}]

Alternatively, if the letters in question all have the attribute ⟨vowel⟩, this test could be expressed in the following, simpler form (which can also be tested more quickly):

[%text _^1 ⟨vowel⟩]

Patterns are not restricted to testing the text stream. The following pattern tests the phoneme stream for the phoneme **dh** after the sync mark pointed to by ^1:

[%phoneme _^1 dh]

Patterns can also switch freely from one stream to another, wherever there is a sync mark that is in both streams. For example, the pattern

[%text _^1 a [%phoneme ⟨consonant⟩] e]

matches if after ^1 there is the letter **a**, a sync mark spanning the text and phoneme streams (implicitly indicated by the square bracket), a phoneme with the attribute ⟨consonant⟩, another sync mark spanning the text and phoneme streams, and the letter **e**. It would match, for example, the sample delta shown in Fig. 10.

Because it would be tedious in writing synthesis rules to have to keep repeating stream identifications, such as **%text** or **%phoneme**, Delta allows the rule-writer to specify two default streams for pattern matching — a *primary default stream* for portions of patterns outside square brackets and a *secondary default stream* for portions in square brackets that have no explicit stream specification. The rest of this paper assumes that the rule-writer has made the text stream the primary default stream, and the phoneme stream the secondary default stream. With these defaults, the above pattern can be simplified to

{ _^1 a [⟨consonant⟩] e }

Curly braces, rather than square brackets, surround the pattern, since square brackets would test the phoneme stream, rather than the text stream. Curly braces can surround patterns and most subpatterns, without changing the meaning of what they enclose.

In order to test pointer variables, such as ^1 in the pattern just shown, it is necessary first to set them. Initially, two built-in pointers are set: ^**left**, which points to the leftmost sync mark of the delta, and ^**right**, which points to the rightmost sync mark. Other pointers are set as side-effects of pattern matches, by preceding their names with an exclamation point. For example, the following pattern is the same as the previous one except that if the entire pattern matches, ^2 will point to the sync mark after the letter **a**:

{ _^1 a !^2 [⟨consonant⟩] e }

Any side-effects of testing a pattern are undone if the pattern fails to match, so that failed tests will not affect the delta or any variables. If the above pattern failed to match after ^2 had been set (because the delta contained two consonants at that point, for example) the setting of ^2 would be undone.

Patterns can be quite complex. For example, the following, more realistic pattern for English, adds a test to the above pattern to determine whether the letter **e** is root-final (in order to predict the pronunciation of the vowel phoneme corresponding to the letter **a** in words like *make*, *date*, *dating* ("date" + "ing"), and so forth):

{ { _^1 a !^2 [⟨consonant⟩] e ! ^end}
& [%morph root _^end] }

The **&** operator connects two independent tests that must both succeed for the pattern to match. The left conjunct is always tested before the right, which ensures, in this case, that ^**end** is set by the first half of the expression before it is checked by the second half.

Not only can patterns test whether two tokens in different streams (such as **e** and **root** above) are bounded by the same sync mark, but they can also test whether a token in one stream is "contained in" a token in another stream (regardless of whether the tokens are bounded by the same sync mark). For example, given a delta with syllable tokens and associated stress attributes, the rule in English that reduces vowels in most unstressed syllables could easily test whether each token with the attribute ⟨vowel⟩ is in an unstressed syllable.

Assume that a vowel-reduction rule is being applied to the word *discover* and that ^1 is pointing after the phoneme **E**, as shown in Fig. 11. (Ignore the \\^1 in the figure for the moment.) The following pattern, which tests whether the vowel preceding ^1 is in an unstressed syllable, would match the phoneme **E**:

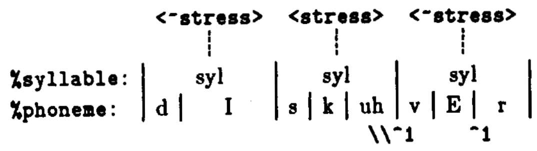

Fig. 11. Delta fragment for *discover*.

{ [⟨vowel⟩ _^1] & [%syllable_ \\^1 ⟨~stress⟩] }

This pattern uses the *left context operator* \\ to specify the closest sync mark in the syllable stream to the left of ^1 as the anchor for the second test, as shown in Fig. 11. Since a syllable token with attribute ⟨~stress⟩ follows this sync mark, the pattern succeeds. The same pattern would match the phoneme **I** of the first syllable if ^1 were positioned at the sync mark following the **I**.

Actions: A pattern is usually coupled with an action to be performed if the pattern matches. For example, the pattern for words like *bathe* would be coupled with an action that inserts the phoneme **e** (the pronunciation for the letter **a**) into the phoneme stream between pointers ^1 and ^2 (replacing the gap synchronized with the letter **a** in Fig. 10), as shown below:

```
{ { _^1 a !^2 [⟨consonant⟩] e !^end}
                    & [%morph root _^end] }
    -> insert [e]  ^1...^2;
```

This rule applies not only to the root **bathe**, but to any root containing the letter **a** followed by a consonantal phoneme and a root-final letter **e**—for example, *bake*, *fate*, and *lathe*. Fig. 12 shows the sample delta for *bathe* after applying the rule.

```
%morph:   |           root          | suffix |
%letter:  | b | a | t | h |    e    |   d    |
%phoneme: | b | e |  dh   |   GAP   |   d    |
               ^1  ^2
```

Fig. 12. Delta fragment for *bathe* after e-insertion.

The rule just developed can be expanded to handle any vowel, not just **a**, in the same context:

```
{ {_^1 ⟨vowel⟩ !^2 [⟨consonant⟩] e !^end}
                        & [%morph root _^end] }
    -> strong_vowel (^1, ^2);
```

The only change to the pattern is a test for a letter with the attribute ⟨vowel⟩, rather than for the letter **a** alone. The action of the rule invokes a procedure named **strong_vowel**, which assigns the appropriate phoneme for the particular vowel matched.

A definition for the strong_vowel procedure is shown in Fig. 13. The body of the procedure is an if-statement. In an

```
proc strong_vowel(^before, ^after);
   if
     {_^before a} -> insert [e]  ^before...^after;
     {_^before e} -> insert [i]  ^before...^after;
     {_^before i} -> insert [ay] ^before...^after;
     {_^before o} -> insert [o]  ^before...^after;
     {_^before u} -> insert [u]  ^before...^after;
   fi;
end strong_vowel;
```

Fig. 13. Strong vowel procedure.

if-statement, the patterns to the left of the arrows are tested in succession until one of them succeeds, and the action specified to the right of the arrow for that pattern is executed.

Delta's procedures are much like those of conventional programming languages. Parameters can be passed by value or by value-return. Passing by value means that a copy of the argument is given to the procedure, so that any changes made to the copy will not affect the variable in the calling program. An argument passed by value-return, in contrast, is copied for the procedure, and then copied back to the calling program if the procedure succeeds.

Delta is unusual in that the procedure call, rather than the definition of the procedure, specifies which parameter passing mechanism to use. The strong_vowel procedure call above has two arguments, both of which are passed by value. Preceding either argument with an exclamation point would indicate that the argument may be modified when the procedure returns.[5]

The next example shows how to insert the appropriate strong vowel without calling a procedure, by using a name-valued field. Assume that the definition for the text stream shown in Fig. 3 is expanded as shown in Fig. 14 to include the name-valued fields **strong_pronunc** and **default_pronunc**, whose possible values are the names of tokens in the phoneme stream. Following the field defini-

```
stream %text;
       :
    strong_pronunc:  names in %phoneme;
    default_pronunc: names in %phoneme;
    a has strong_pronunc: e,  default_pronunc: ae;
    e has strong_pronunc: i,  default_pronunc: E;
    i has strong_pronunc: ay, default_pronunc: I;
    o has strong_pronunc: o,  default_pronunc: a;
    u has strong_pronunc: u,  default_pronunc: uh;
       :
end %text;
```

Fig. 14. Expanded text stream definition.

tions is a list of letters and their values for these fields. For example, **a** is defined to have the phoneme name **e** as the value of the **strong_pronunc** field and the phoneme name **ae** for **default_pronunc**.

Given these definitions, the root-final **e** rule can now be written without a procedure as follows:

```
{ { _^1 ⟨vowel⟩ !$v !^2 [⟨consonant⟩] e !^end}
                              & [%morph root _^end] }
    -> insert [⟨name: $v.strong_pronunc⟩] ^1...^2;
```

The pattern is the same as before, except for the addition of the expression **!$v**, which saves a copy of the matched vowel (in the sample delta for *bathed*, the letter **a**) in the *token variable* **$v**. The action is a single insertion, rather than a call to a more complex procedure. It inserts the phoneme whose name is the value of the **strong_pronunc** field of the token in **$v**—in the case of **a**, the phoneme named **e**.

Our sample rules so far have shown alternative ways to synchronize one token (a phoneme) with another token (a letter). Delta also provides a way to insert a token between two others (for example, to insert the vowels in the Hebrew example in Fig. 6 between the consonants). Rule-writers

[5] The conventional choices for parameter passing (pass-by-value and pass-by-reference) are incompatible with each other, so a choice must be made for each parameter when the procedure is defined. Pass-by-value and pass-by-value-return, however, are compatible, so the choice can be made independently for each invocation of the procedure.

can specify whether new tokens should be visible or invisible in other streams (that is, whether a gap should be inserted in each other stream or not).

Insertion is not the only possible action for a rule. Rules can also delete tokens or set the values of fields. The following rule deletes the first of two identical phonemes not separated by a morph boundary (for example, the first **t** in *ditto*):

```
[ _^1 ⟨ ⟩ !$ph !^2 [~ [%morph] ~] $ph ]
     -> delete %phoneme ^1...^2;
```

The empty angle brackets ⟨⟩ match any single token in the phoneme stream. A copy of that token is stored in the token variable **$ph**, and ^2 is set at the sync mark following the token. The next part of the pattern checks that this sync mark does not extend into the morph stream — that is, that there is no intervening morph boundary. The outer negation brackets [~ ... ~] indicate that the enclosed change to the morph stream must fail for the pattern as a whole to succeed. The pattern then checks whether the next phoneme is the same as the phoneme stored in **$ph**. If so, the action deletes the first phoneme.

This deletion rule uses the special Delta negation brackets. Negation brackets are not restricted to surrounding stream changes. For example, the simple pattern

```
{ _^1 [~⟨vowel⟩ ~] }
```

tests for the absence of a letter token with the attribute ⟨vowel⟩ after the anchor. The pattern { _^1 ⟨ ~vowel⟩ }, on the other hand, would test for the presence of a token that does not have the attribute ⟨vowel⟩, and unlike the pattern with negation brackets, would not match when ^1 points to the rightmost sync mark in the delta.

Loops: All of the sample rules so far have been applied once to a particular place in the delta. Delta provides for repetitive rule application with its **loop** construct. For example, the **forall** loop in Fig. 15 breaks each word of an utterance into morphs.

```
loop forall {_^begin <letter>++ !^end} from ^left;
    break_into_morphs(^begin, ^end);
    continue from ^end;
pool;
```

Fig. 15. Forall loop for breaking words into morphs.

The **from**-option "**from ^left**" initializes the anchor **^begin** to ^left. The **forall** pattern matches words (uninterrupted sequences of letters terminated by a punctuation mark or white space). The ++ after ⟨letter⟩ indicates that the longest sequence of one or more letters should be matched. If the anchor **^begin** is not before a letter, it is advanced one sync mark in the text stream, and the pattern is tried again. This advancing is repeated until the **forall** pattern matches (in which case the body of the loop is executed), or until **^begin** reaches the right end of the delta and cannot be advanced (in which case the loop terminates).

For each word found, the body of the loop calls a procedure named **break_into_morphs** to insert **prefix, suffix,** and **root** tokens into the morph stream. The **continue** statement advances **^begin** to the sync mark pointed to by **^end** (the sync mark immediately following the matched word) and continues the scan for words.

This loop uses one of the **forall** options, the **from**-option. Other **forall** options can be used to specify in which direction to scan, in which stream, and which pointer to advance.

C. Dictionary Facilities

Writing rules for words that behave regularly is only a small part of the problem of text-to-speech conversion. Even more challenging is handling phenomena that cannot easily be captured in rules. Traditional exception dictionaries containing words and their pronunciations work well for very irregular spellings like *lasagna* and *solder*. Delta's more flexible dictionary can store a much broader class of information.

Delta's dictionary has two parts: the *action dictionary* and *sets*. The action dictionary contains token sequences (for example, text or phoneme sequences) and associated actions (for example, insertions into the phoneme stream). Sets contain token sequences, but no actions.

Delta's **find**-statement looks up token sequences in the dictionary. The expression

```
find {^beg...^end}
```

looks in the action dictionary for the sequence of text tokens between **^beg** and **^end**, returning success if it is found and failure if not. On success, the action specified for the entry in the dictionary is automatically performed. The expression

```
find {^beg...^end} in prepositions
```

looks for the text sequence between **^beg** and **^end** in the set named **prepositions** and simply returns success or failure.

Sets: Sets provide the rule-writer with a succinct way to group words that behave similarly. For example, the following statement adds the words *humbly, crumbly, nimbly,* and *assembly* to the small set of words in which the final *ly* does not function as a suffix. (If *ly* were stripped as a suffix, later rules would make the **b** silent, as in *dumbly* and *numbly,* where *ly* is a true suffix.)

```
set no_ly_strip contains {humbly}, {crumbly},
                         {nimbly}, {assembly};
```

The *ly*-stripping rule can now be expressed straightforwardly, as a simple pattern with an explicit set of exceptions:

```
{!^suff ly _^end}
    -> ~find {^beg...^end} in no_ly_strip
     -> insert [%morph suffix]  ^suff...^end;
```

First, the pattern checks that the word ends with **ly**.[6] If so, the word is looked up in the set **no_ly_strip**. The "~" in front of **find** is a negation operator. If the word is not found, the **suffix** token is inserted into the morph stream.

The set **no_ly_strip** in this example prevents an action from occurring. Sets are also useful for invoking actions. The example in Fig. 16 uses a set named **prefix_not_syl** to

[6]It is assumed that the rule would not be tested against one-syllable words such as *sly* and *fly*.

```
set prefix_not_syl contains
    {discover}, {discuss}, {disease}, {mistake};

find {^begw...^endw} in prefix_not_syl
    -> {!^ends <> _^endp}
       -> insert [%syllable syl] ^begw...^ends;
```

Fig. 16. Syllable insertion in words like *discover.*

insert the first syllable token in words like *discover*, where the prefix boundary (marked by ^**endp**) does not coincide with the syllable boundary (see Fig. 5). First, the set is defined, then the word surrounded by ^**begw** (begin word) and ^**endw** (end word) is looked up in the set.[7] If it is found, ^**ends** (end syllable) is placed one letter before the end of the prefix, and a **syl** token is inserted into the syllable stream. For example, in *discover*, the token **syl** would be synchronized with the letters **d** and **i**.

Using sets for exceptions is often clearer than using the action dictionary, and produces faster, more compact rules.

Insertions: The exception dictionaries of previous systems have been used mainly to store irregular pronunciations for text strings. The identical effect can be achieved in Delta by storing sequences of letters in the action dictionary with associated insert actions that put the pronunciations in the phoneme stream.

To insert tokens into the delta, the dictionary must know the sync marks between which the insertion is to occur. Pointer variables are therefore passed to the dictionary, in much the same way as they are passed to a procedure. In fact, a dictionary definition block (in which all action dictionary entries must be enclosed) looks much like an ordinary procedure definition:

```
dictionary (^beg, ^end, ^1, ^2);
        ⋮
end dictionary;
```

The dictionary's header specifies the names of the pointers delimiting the sequence being looked up (here, ^**beg** and ^**end**), and the names of two auxiliary pointers (^**1** and ^**2**) that can be used within the dictionary entries, as illustrated below.

Since simple insert actions are very common, Delta provides a shorthand notation for them. Instead of putting the general form

```
{of} -> insert [uh v] ^beg...^end;
```

inside the dictionary definition block, the rule-writer can use the shortened form

```
{of} => [uh v];
```

Just as the source language provides a special encoding for simple insert actions, the Delta Machine uses a special compact representation for them.

In many cases, items are placed in the dictionary because one letter (or a sequence of letters corresponding to one phoneme) has an irregular pronunciation. A dictionary entry can use the auxiliary pointers to insert a pronunciation for the irregular grapheme alone. The following dictionary entry, for example, surrounds the irregular letter **o** of *glove, shove,* and *love* with the pointers ^1 and ^2, and inserts the phoneme **uh** into the phoneme stream between these pointers:

```
{gl !^1 o !^2 ve},
{sh !^1 o !^2 ve},
{ l !^1 o !^2 ve} => [uh];
```

Whenever one of these roots is looked up in the dictionary, the phoneme **uh** is inserted. The rest of the pronunciation can be determined by the rules.

Other Actions: An especially powerful feature of the Delta dictionary is its ability to associate an arbitrary action (not just an insertion) with an entry, as illustrated in Fig. 17.

```
dictionary(^beg, ^end, ^1, ^2);
  {have} ->
       if { _^end ' ' to !^end [^ <letter> ^] }
            -> insert [h ae f t uh]  ^beg...^end;
       else -> insert [h ae v]       ^beg...^end;
       fi;
   ⋮
end dictionary;
```

Fig. 17. Dictionary actions for *have to* and *have.*

This example recognizes *have to* as a single phonetic word pronounced [h ae f t uh], and *have* in other contexts as [h ae v].[8] When the word *have* is looked up, ^**end** is set at the sync mark after the *e*. If the orthographic word *to* follows, the dictionary action moves ^**end** to the sync mark following the *to*, and inserts the corresponding pronunciation [h ae f t uh] into the phoneme stream. On the other hand, if *have* is not followed by *to*, ^**end** is not moved, and the pronunciation [h ae v] is inserted.

The dictionary's ability to move pointers keeps the general flow of control of the rule program simple, as illustrated by the **forall** loop in Fig. 18.

```
loop forall {_^beg <letter>++ !^end} from ^left;
    find {^beg...!^end};
    continue from ^end;
pool;
```

Fig. 18. Forall loop for looking up words in the dictionary.

The exclamation point before ^**end** in the **find**-statement indicates that the pointer's value may be changed by the dictionary (see the discussion of value-return parameters in Section II-B). If the loop is applied to the sentence *I have to go*, it would first look up *I*, then *have*. Recognizing that the following *to* is phonetically part of *have*, the dictionary would assign a pronunciation for the entire phonetic word,

[7]The set **prefix_not_syl** contains many more words than are shown in the example.

[8]The syntactic context determines whether *have to* functions as a single phonetic word. For example, in the sentence *I have to drink a lot of milk* it does, whereas in the sentence *Mary has more to eat than I have to drink*, it does not. The example in this paper is thus somewhat oversimplified, showing how *have to* might be handled in the absence of a proper syntactic analysis.

and would set ^end after the *to*, causing the forall loop to continue by looking up *go*.

Dictionary actions can do more than insert pronunciations. Any statements (including procedure calls) that are valid in the Delta program are valid in a dictionary action. The action in Fig. 19, for example, marks the final syllable of *police* as stressed and inserts the phoneme **i** for the letter **i**.

```
{pol !^1 i !^2 ce} ->
   do
      mark [%syllable <stressed>]
        (\\^1 in %syllable)...(//^2 in %syllable);
      insert [i] ^1...^2;
   od;
```

Fig. 19. Dictionary action for *police*.

The do and od keywords group the statements that comprise the action. The **mark**-statement marks the syllable token that "contains" the letter **i** as ⟨stressed⟩, using Delta's context operators \\ and // to delimit the syllable token. The insert-statement inserts the phoneme **i**.

Dictionary Definition: All dictionary entries other than the "set contains" declaration must be contained in a dictionary definition block. The block in Fig. 20 illustrates the various forms of dictionary entries.

```
dictionary(^beg, ^end, ^1, ^2);
  set prepositions contains
              {at}, {in}, {under}, {by};
  {coup}                     => [k u];
  {gl !^1 o !^2 ve}          => [uh];
  {have}                     ->
       if { _^end ' ' to !^end [^ <letter> ^] }
            -> insert [h ae f t uh] ^beg...^end;
       else -> insert [h ae v]      ^beg...^end;
       fi;
  {l !^1 o !^2 ve}           like {glove};
  {near} (in prepositions);
  {of}   (in prepositions) => [uh v];
       :
 end dictionary;
```

Fig. 20. Sample dictionary definition block.

The only expressions not yet described are the like-shorthand and the "(in *set-name*)" clause. The like-shorthand is used for the action of *love* to indicate that the action is identical to that of *glove*. Alternatively, *love* and *glove* could be listed together with a single action, as illustrated earlier. The like-shorthand is useful when the rule-writer wishes to keep the dictionary entries in alphabetical order, as in this example.

The "(in *set-name*)" clause is used for the preposition *near* as an alternative way to add it to the set **prepositions**. This clause is most useful when an entry both belongs to a set and has an associated action (for example, *of*).

Because it is tedious to type braces or square brackets around each dictionary entry, Delta provides an alternate syntax, illustrated in Fig. 21.

```
dictionary(^beg, ^end, ^1, ^2);
  set prepositions contains %text:
                 at, in, under, by;
  %text:
    coup          => [k u];
    gl !^1 o !^2 ve => [uh];
          :
end dictionary;
```

Fig. 21. Alternate syntax for dictionary entries.

III. The Delta Debugging Environment

The Delta language is compiled into instructions that are executed by the Delta Machine interpreter. The interpreter includes an interactive symbolic debugger for tracing and controlling program execution. The debugger is complemented by the Delta linker, which can "patch" changed procedures into existing load modules, so that compilation and linking delays are minimized.

Debugging aids like these are important for developing any complex program, and are especially important for developing synthesis rules, where the output (the synthesizer parameter values) is generally not well-specified, but is determined through extensive experimentation. A synthesis rule set undergoes constant revision as applications change or new knowledge is gained. Synthesis rule development is generally an iterative, trial-and-error process that involves several rounds of examining data such as word lists and spectrograms, formulating hypotheses, embodying the hypotheses in the rule program, and testing the program. Often an incorrect pronunciation is not revealed until many utterances have been tested, and it can stem from almost any portion of the rules — for example, from incorrect text-to-phoneme conversion or from incorrect synthesizer parameter value assignment.

A. The Delta Debugger

The Delta debugger provides a set of commands with which rule-writers can tailor their debugging to their particular programs. The debugger command language is flexible enough to accommodate any Delta program, yet powerful enough that rule-writers can produce complicated rule traces with only a handful of commands. (Since the Delta debugger cannot predict the organization of the program to be debugged, it cannot have the kind of built-in, predetermined rule-tracing facilities of a more restricted system like SRS.)

The debugger's power stems from its detailed knowledge of the Delta source language. It can display and modify the delta data structure, and it uses source language notation for procedure names, stream names, variable names, token names, field names, and so forth. Breakpoints can be set at arbitrary source locations to suspend execution at those points. Programs can be stepped through source line by source line, with execution stopping after each line. Whenever execution is suspended, commands can be issued to examine and alter the program variables and the delta, or to display selected source lines.

Programs can be debugged at the machine level as well. Breakpoints can be set at specific addresses, the program can be stepped instruction by instruction, and the machine instructions can be displayed in assembly-language format. Machine-level debugging, although more complicated than source-level debugging, can be useful on occasions. For example, at the machine level, the rule-writer can trace the execution of the instructions in a complicated pattern, regardless of how many source lines the pattern consumes, to determine what parts of the delta match what parts of the pattern.[9]

Conditional Breakpoints: Conditional breakpoints are the heart of the debugger. With them, the rule-writer can tell the debugger to suspend execution and accept debugging commands when certain conditions arise. Depending on the rule-writer's specifications, execution may stop in any of several circumstances: just before or just after a change is made to the delta or to specified streams in the delta; at a particular procedure or return from a procedure; at a particular source line number or machine address; upon failure or success of particular pattern-matching instructions; upon failure or success of entire patterns; at the beginning or end of a rule's action; at a particular kind of instruction (for example, at an insert instruction); or at arbitrary points in the program marked with specially named *tags*.

For example, the command

break on delta

instructs the pseudo-machine to halt whenever it is about to change the delta. The debugger then prints the change that is about to occur, and the line of the source program that is being executed. Whenever the machine stops, the rule-writer can issue debugging commands. Debugging commands can be issued to display the delta and various parts of the source program to determine whether a change is correct. Execution can then be restarted anywhere in the program — for example, at the instruction that was interrupted, or at the next instruction, or not at all.

Conditions can also be associated with *ranges* that limit the halting to particular parts of the program. For example, the command

break on delta in strip_prefix

would cause the machine to stop only before the delta is changed by procedure **strip_prefix**. Similarly, the command

break after delta in strip_prefix

would cause the machine to stop immediately *after* the delta is changed by **strip_prefix**.

Ranges may be smaller than entire procedures. For example, the command

break after delta strip_prefix: 3–20

restricts the condition to lines 3 through 20 of procedure **strip_prefix**.

The debugger supports multiple ranges for each condition, and the ranges for any condition are independent of those for other conditions.

[9]One problem with the SRS trace facility has been the inability to isolate which parts of complicated rule contexts succeed or fail in particular cases. This inability has been particularly troublesome in the case of negated expressions.

Automatic Commands: Rather than wait for commands to be entered by the rule-writer, the debugger can execute pre-specified commands automatically when a breakpoint occurs. For example, the command

**break after delta in strip_prefix (print delta \\
%text %morph ^begw...^endw; go)**

instructs the debugger to print the text and morph streams of the delta between **^begw** and **^endw** after any change to the delta in procedure **strip_prefix**, and then to continue execution (go) at the point where execution was interrupted. Thus with a single command the rule-writer can obtain a list of words that are modified by **strip_prefix**.

If the command list associated with a condition does not include a command like go to continue execution, the rule-writer is prompted for further debugging commands.

Debugging Variables: The Delta debugger includes built-in variables that rule-writers can use to mark positions in the delta or to hold tokens. The following command, which invokes the procedure **strip_prefix**, uses the built-in pointer variables ^^1 and ^^2 as value-return arguments to the procedure call:

call strip_prefix(!^^1, !^^2)

It is assumed that these pointers were set to specific sync marks in the delta by earlier debugging commands. The extra "^ " marks the pointers as built-in debugging variables rather than ordinary program variables.

Since rule-writers can surround any token sequences in the delta with built-in pointer variables, then pass them as arguments to a procedure, any procedure can be tested independently of others. Since debugging commands can also initialize the delta, such independent testing is a realistic way to test a procedure before the supporting and calling routines are written.

The debugger includes additional built-in variables to which it assigns values whenever a breakpoint occurs. These variables keep track of the position in the delta where a pattern is being matched, the stream in which the pattern is being (or was) matched, the left end of where a change in the delta will take (or took) place, the right end of such a change, the stream in which the change will take (or took) place, the number of the source line being executed, and the address of the instruction being executed.

B. The Delta "Patcher"

Once an error has been isolated with the debugger, the relevant code can be corrected and retested quickly by recompiling it and patching it into the existing load module with the linker. In this way, the frustrating compilation and linking delays with which most programmers are all too familiar are minimized.

For example, the command

patch strip_prefix rules

instructs the linker to add the procedure **strip_prefix** to the existing load module named **rules**, overwriting any previous **strip_prefix** procedure.

In a similar way, dictionary entries can be added or altered and be quickly patched into an existing dictionary.

IV. Delta in the Future

This paper has focused on the present version of the Delta System. To increase the power of Delta further, several extensions have been planned or considered. Most important, numeric capabilities will be added to give rule-writers the same flexibility in producing synthesizer values as the current version gives them in converting text to non-numeric phonetic units.[10]

The second version of Delta will include a built-in data structure, separate from the delta, that will be designed for efficient numeric operations, such as interpolations. This data structure will be "synchronizable" with the delta at selected points along the time continuum, and rules will be able to test easily for particular synchronizations of the two data structures. Delta will also include numeric attributes, so that tokens in the delta itself can contain numeric information (for example, default target values for particular synthesizer parameters).

Other data structures are being considered. For example, pointer attributes may be added, so that tokens can point to other tokens in any stream. With such pointer attributes, arbitrary graphs (for example, recursive tree structures) could be built and manipulated.

The second version of Delta, with full-fledged numeric capabilities, will give rule-writers a much wider choice of strategies for producing synthesizer values than have previous systems.

First, it will not force rule-writers to maintain the traditional division between rule components (purely symbolic manipulations followed by purely numeric ones). Instead, it will allow them to intermingle both kinds of manipulations.

Second, it will not force rule-writers to select between a concatenative approach, in which synthesizer values are extracted from a library of pre-stored units, and a rule-based approach, in which all synthesizer values are generated by rules. Instead, it will allow them to extract some values from the dictionary, and to generate other values by rule.

Third, it will not force rule-writers to generate synthesizer values on the basis of one particular kind of unit. Instead, it will allow them to generate values on the basis of whatever unit works best for the synthesizer parameter and the phenomenon in question (for example, demisyllables or diphones in some cases, phonemes in others).

Fourth, it will not force rule-writers to synthesize in terms of a particular type of synthesizer parameter. Instead, it will allow them to synthesize in terms of articulatory parameters, acoustic parameters, or both (provided, of course, that they have written or have been supplied with a program to drive the synthesizer to be used).

Fifth, it will not force rule-writers to assign synthesizer parameter patterns on a unit-by-unit basis. Instead, it will allow them to generate patterns that extend across domains of arbitrary size. For example, the system will be able to accommodate a variety of models of intonation, including those that generate contours on the basis of phonological targets linked by low-level transitions [24]–[27] and those that treat contours as the result of two or more interacting mathematical functions [28].

[10]Users can synthesize with the current version of Delta by interfacing their own procedures to generate synthesizer values on the basis of the information present in the Delta.

In general, the full-fledged numeric version of Delta will let rule-writers combine the favorable aspects of different synthesis techniques, selecting the technique that works best for the language and phenomenon being modeled.

V. Conclusion

The Delta System provides a powerful framework for expressing knowledge about speech synthesis. Rule development within this framework should enhance productivity and speed progress, since the pace of developments in speech synthesis depends largely on the ease of creating, testing, and discarding rules, and on the ease of transferring the knowledge gained to new practitioners.

Delta's flexibility should satisfy almost all synthesis rule developers, whether linguistic researchers or the developers of talking products. The practicality of the resulting rule sets will make it more than just a research tool.

Acknowledgment

The authors thank R. D. Ladd and G. N. Clements for their helpful comments on an earlier draft of this paper, and M. Beckman and O. Fujimura for their many helpful suggestions at various stages of the paper. They also acknowledge T. Räuchle, who programmed the dictionary builder, J. Lynn, who programmed the assembler/linker and portions of the compiler, and J. Dill, who programmed portions of the compiler.

References

[1] J. Allen, "Synthesis of speech from unrestricted text," *Proc. IEEE*, vol. 64, pp. 422–433, 1976.

[2] J. Allen, S. Hunnicutt, R. Carlson, and B. Granström, "MITalk-79: The 1979 MIT text-to-speech system," in *ASA-50 Speech Communication Papers*, J. J. Wolf and D. H. Klatt, Eds., New York: Acoust. Soc. Amer., 1979, pp. 507–510.

[3] S. R. Hertz, "SRS text-to-phoneme rules: A three-level rule strategy," in *Proc. IEEE Int. Conf. Acoust., Speech, Signal Processing*, pp. 102–105, 1981.

[4] _____, "The 'morphology' of English spelling: A look at the SRS text-modification rules for English," in *Working Papers of the Cornell Phonetics Laboratory*, no. 1, pp. 17–28, Dec. 1983.

[5] H. S. Elovitz, R. Johnson, A. McHugh, and J.. E. Shore, "Letter-to-sound rules for automatic translation of English text to phonetics," *IEEE Trans. Acoust., Speech, Signal Processing*, vol. ASSP-24, no. 6, pp. 446–473, Dec. 1976.

[6] O. Fujimura, M. J. Macchi, and J. B. Lovins, "Demisyllables and affixes for speech synthesis" (Abstract), in *Contributed Papers, vol. 1, 9th Int. Congr. on Acoustics* (Madrid, Spain, July 4–9). Madrid: Spanish Acoust. Soc., 1977, p. 515.

[7] C. P. Browman, "Rules for demisyllable synthesis using Lingua, a language interpreter," in *Proc. IEEE Int. Conf. Acoust., Speech, Signal Processing*, pp. 561–564, 1980.

[8] H. Dettweiler, "An approach to demisyllable speech synthesis of German words," in *Proc. IEEE Int. Conf. Acoust., Speech, Signal Processing*, pp. 110–113, 1981.

[9] J. Holmes, I. Mattingly, and J. Shearme, "Speech synthesis by rule," *Language and Speech*, vol. 7, pp. 127–143, 1964.

[10] D. H. Klatt, "The KLATTalk text-to-speech conversion system," in *Proc. IEEE Int. Conf. Acoust., Speech, Signal Processing, pp. 1589–1592, 1982.*

[11] C. H. Coker, "A Model of articulatory dynamics and control," *Proc. IEEE* (Special Issue on Man–Machine Communication by Voice), vol. 64, pp. 452–459, Apr. 1976.

[12] J. P. Olive, "Speech synthesis by rule," in *Speech Communi-*

cation, vol. 2, G. Fant, Ed. New York: Halsted, 1974, pp. 255 – 260.

[13] H. E. Wolf, "Control of prosodic parameters for a formant synthesizer based on diphone concatenation," in *Proc. IEEE Int. Conf. Acoust., Speech, Signal Processing*, pp. 106 – 109, 1981.

[14] R. Carlson, and B. Granström, "A text-to-speech system based entirely on rules," in *Proc. IEEE Int. Conf. Acoust., Speech, Signal Processing*, pp. 686 – 688, 1976.

[15] S. R. Hertz, "From text to speech with SRS," *J. Acoust. Soc. Amer.*, vol. 72, no. 4, pp. 1155 – 1170, 1982.

[16] M. E. Beckman, S. R. Hertz, and O. Fujimura, "SRS pitch rules for Japanese," *Working Papers of the Cornell Phonetics Laboratory*, no. 1, pp. 1 – 16, 1983.

[17] S. R. Hertz, and M. Beckman, "A look at the SRS synthesis rules for Japanese," in *Proc. IEEE Int. Conf. Acoust., Speech, Signal Processing*, pp. 1336 – 1339, 1983.

[18] S. R. Hertz, "Multi-language speech synthesis: A search for synthesis universals" (abstract), *J. Acoust. Soc. Amer.*, vol. 67, suppl. 1, 1980.

[19] A. Aggoun, C. Sorin, F. Emerard, M. Stella, "Prosodic knowledge in the rule-based Synthex expert system for speech synthesis," in *New Systems and Architectures for Automatic Speech Recognition and Synthesis* (Proc. of the NATO Advanced Study Institute held at Bonas, France, July 2 – 14, 1984).

[20] N. Davidsen-Nielsen, "Syllabification in English words with medial *sp, st, sk,*" *J. Phonetics*, vol. 2, pp. 15 – 45, 1974.

[21] D. A. Dinnsen, "A Re-examination of phonological neutralization," in *Research in Phonetics*," Rep. 4, Dept. of Linguistics, Indiana University, pp. 59 – 92, June 1984.

[22] S. R. Anderson, "Nasal consonants and the internal structure of segments," *Language*, vol. 52, pp. 326 – 344, 1976.

[23] H. van der Hulst and N. Smith, "Autosegmental and metrical phonology," in *The Structure of Phonological Representations*, pt. I. Dordrecht, Holland ≤Cinnaminson, NJ: Foris Publ., 1982.

[24] J. Pierrehumbert, "Synthesizing intonation," *J. Acoust. Soc. Amer.*, vol. 70, pp.. 985 – 995, 1981.

[25] M. D. Anderson, J. B. Pierrehumbert, and M. Y. Liberman, "Synthesis by rule of English intonation patterns," in *Proc. IEEE Int. Conf. on Acoustics, Speech, and Signal Processing*, pp. 2.8.1.– 2.8.4, 1984.

[26] G. Bruce, *Swedish Word Accents in Sentence Perspective*. Lund, Sweden: Gleerup, 1977.

[27] D. R. Ladd, "Phonological features of intonational peaks," *Language, vol. 59, pp. 721 – 759, 1983.*

[28] H. Fujisaki and K. Hirose, "Modeling and dynamic characteristics of voice fundamental frequency with applications to analysis and synthesis of intonation," in *Preprints of Papers, Working Group on Intonation*, XIIIth Int. Congr. of Linguists, Tokyo, Japan, 1982.