TU/e EINDHOVEN UNIVERSITY OF TECHNOLOGY

Implementation of second synthesis module in Spraakmaker

Citation for published version (APA):
van Leeuwen, H. C. (1991). *Implementation of second synthesis module in Spraakmaker*. (IPO rapport; Vol. 833). Instituut voor Perceptie Onderzoek (IPO).

Document status and date:
Published: 08/11/1991

Document Version:
Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:

• A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the "Taverne" license above, please follow below link for the End User Agreement:

www.tue.nl/taverne

Take down policy
If you believe that this document breaches copyright please contact us at:

openaccess@tue.nl

providing details and we will investigate your claim.

Download date: 16. Nov. 2023

Institute for Perception Research
PO Box 513, 5600 MB Eindhoven

HvL/hvl 91/17
08.11.1991

Rapport no. 833

Implementation of second synthesis module in Spraakmaker

H.C. van Leeuwen

Implementation of second synthesis module in Spraakmaker

Hugo C. van Leeuwen

1 Introduction

This report describes the implementation of the second synthesis module in Spraakmaker. As the word 'second' implies, at the time of writing there are two synthesis modules. The first is the old one, copied from the DS program, built around the data structures used in DS. Due to these data structures, in DS the processes are not functionally separated. For instance, the framebuffer (this is the buffer in which the AP-parameters are stored) is filled in the duration module, because the duration module needs initial durations of phonemes, which it derives from the diphones. It is not really necessary to fill the framebuffer at that moment, but in DS it does no harm since it should be filled anyway. However, in Spraakmaker we want the modules to be as independent as possible, and we do not want a common data structure separate from the grid. Therefore, we do not want to fill a framebuffer—we don't even want to *have* a framebuffer—before we are actually in the synthesis process. As a consequence, the strategy followed in the old synthesis module falls short for our current purposes.

Therefore, a second synthesis module has been developed, designed specifically to synthesize the symbolic data represented in the grid. This report is the technical documentation of that module.

2 Module structure

The synthesis module is written in PASCAL and consists of several files. Five of them are stored in the 'synthesis2' directory (this is: '`sm$root:[speech2_1.speech2_2]`'), and directly concern the second synthesis module. Some other files are also concerned, which deal with the passing of data from PROLOG to PASCAL. As this mechanism is structured in a manner common to all PROLOG-PASCAL couplings, it will not be described here, but in the general Spraakmaker documentation provided by Rick te Lindert. The five files directly concerning the second synthesis module are listed below:

- SPEECH2_ENV.PAS: defines the constants, types, variables and procedures global for the four files below
- FILL_FRAMES.PAS: contains the routines which fill the framebuffer and compute the relevant duration and pitch parameters.
- SPEECH2.PAS: contains the routines which receive the symbolic data from the PROLOG part of Spraakmaker, and the routine which is invoked by Spraakmaker to effectuate synthesis.
- AP2.PAS: contains an outer shell for (LPC) synthesis of a framebuffer.
- APOBUF2.PAS: contains the kernel of (LPC) synthesis of a framebuffer.

The files are structured as depicted in figure 1. In this report only SPEECH2_ENV.PAS and FILL_FRAMES.PAS will be discussed; SPEECH2.PAS is described in the general Spraakmaker documentation, AP2.PAS and APOBUF2.PAS are slight variations on algorithms of the LVS system (Allain, 1990).

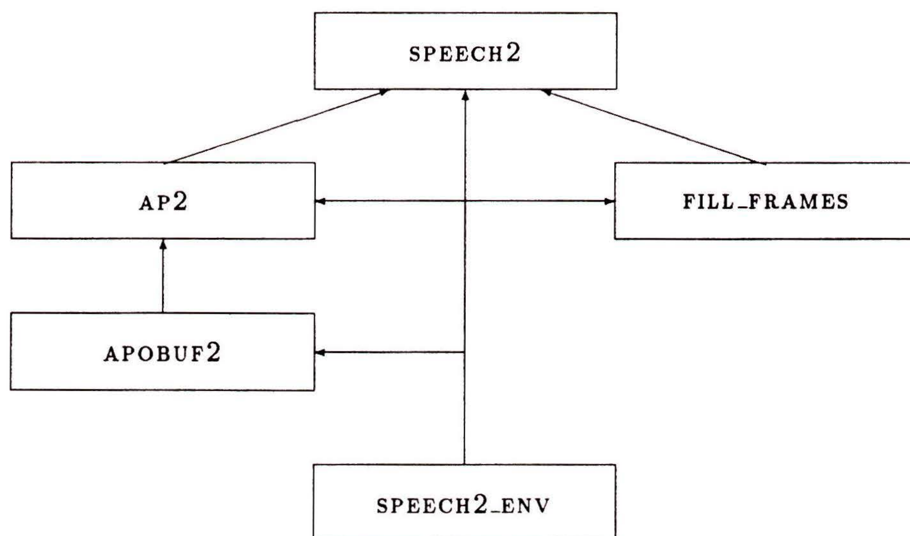

Figure 1: The structure of the five synthesis files.

3 Input/output

By the time the synthesis module is invoked, the Spraakmaker grid contains a host of symbolic data. However, not all data are relevant for the synthesis process. The synthesis module needs the diphones, the frame counts, the durations and pitch contour information. To be more specific, for each sentence in the grid, the `phon.dur`, the `frames.nr`, the `diph.d`, the `into.freq` and `.decli`, and the `pitch.anchor`, `.onset`, `.offset`, `.dur` and `.exc`, and timing information as to how the pitch movements are related to the diphones, is transferred to the synthesis module.

To visualize this a bit, in figure 2 a sample grid is given. Only the relevant streams are given (the phonemes are included for better understanding). The given values are not computed by Spraakmaker but generated by hand, and therefore do not necessarily represent realistic values.

The timing of the pitch movements with relation to the segmental structure (say, the phonemes) is defined by means of the sync marks. Some of the sync marks in the `pitch` stream will be related to sync marks in the phon stream. This will anchor the movement. The convention is: the *left* sync mark for 'vo' movements is anchored to the phon stream, and the *right* sync mark for 'eov' movements. This seems somewhat awkward, (why not always take the left sync mark?), but visually this gives better results. The 'eov' movements only occur at the end of an intonation domain, and if the left sync mark were anchored, it would appear visually as if the movement would take place after the intonation domain.

This convention still leaves a lot of sync marks unanchored in the pitch stream. This is indeed correct, as the moment an accent-lending movement (anchored at 'vo') changes into declination is defined by the `.dur` and `.onset/offset` fields rather than the place where the sync mark separating the two movements is anchored in the phon stream. Thus, in the given example, the timing of the first anchored movement should be read as follows. The movement, the type of which is '1', is anchored between the 'd' and the 'I' in the phon stream (which is indeed the vowel onset, 'vo'; however, 'vo' is a label which has been used in the derivation, and due to which indeed the left sync mark is attached to the phoneme stream,

sent:	declarative												
i_dom:		i_domain											
word.acc:		+			-		-			+			
phon.pho:	-	d	I	t	I	s	@	n	t	E	s	t	-
.dur:	31	46	69	46	69	79	54	62	46	69	79	46	31
frames.nr:	4	3 | 4	4 | 5	4 | 2	4 | 5	4 | 6	3 | 4	4 | 4	4 | 2	4 | 4	4 | 4	4 | 2	4
diph.d:	SI1D1	D1I1	I1T1	T1I1	I1S1	S1C1	C1N1	N1T1	T1E1	E1S1	S1T1	T1SI1	
into.freq:		93.50											
.decl:		-4.65											
pitch.type:		0	1	%						A	0		
.anc:		–	vo	–						vo	–		
.onset:										-10			
.offset:			50										
.dur:		var	120	var						60	var		
.exc:		0.0	6.0	-3.0						-3.0	0.0		

Figure 2: A sample grid containing the streams which are relevant for the synthesis module. The sentence is enclosed by 'gaps'; this is needed for the representation of the diphones. For each intonation domain (i_domain) a starting frequency and a declination must be given. Each intonation domain further consists of a number of pitch movements. Anchored movements ('vo' (vowel onset) and 'eov' (end of voicing)) must be alternated with unanchored movements ('none' or '–'). One other movement is also possible, a movement which is anchored to a previous anchored movement: 'prv'. The combination of an anchored and a 'prv' movement counts as one anchored movement in the alternation scheme.

but serves no further function in the synthesis module). The movement has a duration of 120 ms, and ends 50 ms after the anchor (`onset` defines when the movement starts relative to the anchor, `offset` defines when the movement ends, relative to the anchor. Only one of these two may be defined).

Note that the absolute frame number of the anchor of a movement is not present in the grid. Although the timing relations are transferred to the synthesis module by means of a frame number, the user cannot directly access this value, nor can he/she use it to alter the timing relations; the values are first computed by Spraakmaker when the data are sent to the synthesis module. If a user wants to alter the timing of a pitch movement, one should either relate the anchoring sync mark to another sync mark in the phoneme stream, or adjust the onset or offset fields of the movement. The first option is useful for big shifts (e.g. a different syllable is to receive accent), the second for small shifts (the timing within the syllable). Of course, if one wants, one can screw things up by choosing extreme values for the onset/offset, but one is advised not to or else suffer the consequences. For anchored movements either the onset or the offset should be specified. (In the current implementation, specification of both will lead to selecting the onset value, specification of neither will result in an offset of 0 ms). Onset, offset and duration are assumed to be given in milliseconds. Excursion is assumed to be given in semitones. The type of the pitch movement is transferred to the synthesis module, but only for feedback purposes (e.g. in error messages it may help to locate the source of trouble more quickly).

3.1 The input

The synthesis module expects the following data as input, to be given by Spraakmaker:

- A diphone array containing the diphones to be synthesized, and the number of diphones in that array:

  `dip$_arr` : `diphone_array`; This is an array containing the diphones (and triphones if they occur). The array ranges from 1 to `max_char` (currently 512), and each element of the array is a string of (currently maximal) 30 characters, each denoting a diphone, e.g. 'H1AA1'.

  `nr$_of_diphones` : `integer`; This is the number indicating the actual number of diphones transferred. Call this value n.

- A duration array containing the segmental durations of the phonemes, and the number of durations in that array:

  `dur$_arr` : `duration_array`; This is an array containing the durations. The array ranges from 1 to `max_char` (currently 512), and each element of the array is an integer denoting the duration of a phoneme in ms. Note that the phonemes themselves are not transferred to the synthesis module.

  `nr$_of_durations` : `integer`; This is the number indicating the actual number of durations transferred. Typically, this has the value $n + 1$, unless triphones are involved.

- A 'frames' array containing the number of frames of each half diphone/phoneme, and the number of frame_numbers. Note that this is *not* the frame buffer 'frame_buf'.

  `fra$_arr` : `frames_array`; This is an array containing the frame numbers. The array ranges from 1 to `2*max_char` (1024), and each element of the array is an integer denoting the number of frames in half a diphone.

  `nr$_of_framenrs` : `integer`; This is the number indicating the actual number of frame numbers transferred. Typically, this has the value $2 * n$, unless triphones are involved.

- A specification for the pitch contour. This is a dynamic structure, to be accessed by one pointer, the '`pitch_spec_pointer`' (see section 4, SPEECH2_ENV.PAS). Each sentence contains a number of intonation domains. An intonation domain is either empty or filled. A filled intonation domain corresponds to the utterances, an empty intonation domain corresponds to the pauses between them. Each domain (both empty and filled) has a duration (`.dur`) [ms], a starting frequency (`.freq`) [Hz], a declination (`.decl`) [st/s] (= semitones per second), a number of moves (`.move`), and a next field (`.next`) pointing to the specification for the next domain. The values of these fields vary for empty and filled domains. For filled domains, all fields are relevant. For empty domains, only the duration field must be specified, and the `.move` field should be NIL.

  Each move contains the following fields:

`taip` : Indicates the symbolic representation of the movement (e.g., '1', 'a', 'fl', etc. This is the type that is given in the grid. It serves for error and warning messages only, to be able to locate the source of trouble more quickly.

`anchor` : Indicates the type of anchor. This must be one of the enumerated type '`anchor_taip`' = (vo,eov,prv,none). Here, 'vo' denotes vowel onset, 'eov' denotes end of voicing, 'prv' (=previous) denotes that movement directly follows previous movement, and 'none' denotes no anchor.

`frame` : Indicates the frame number to which the movement is anchored. Only anchored movements should receive a frame number. The 'vo' movements should contain the frame number associated with the *left* sync mark, the 'eov' movements the frame number associated with the *right* sync mark. The 'prv' and 'none' movements should contain the (integer) value zero.

`time` : Indicates the absolute time at which the movement is to start. When FILL_FRAMES is entered, this should always contain the (real) value zero. In the routine COMPUTE_ABS_TIME the frame numbers given in the '`frame`' field will be translated to an absolute time, indicating the starting time of the movement (not the time of the anchor), which are stored in this field.

`onset` : Indicates the onset of the movement, if any. If the onset is specified, the offset field must be unspecified. Since any number, including zero, is a valid value, this parameter is specified by a string. The string '-' means unspecified, all other strings should be ASCII representations of real numbers.

`offset` : Indicates the offset of the movement, if any. If the offset is specified, the onset field must be unspecified. Since any number, including zero, is a valid value, this parameter is specified by a string. The string '-' means unspecified, all other strings should be ASCII representations of real numbers.

`dur` : Indicates the duration in ms of the movement, specified as an integer. Variable durations must be coded as the (integer) value 0.

`exc` : Indicates the excursion in semitones of the movement, specified as a real.

`next` : Pointer to the next movement. Should be NIL if no movement follows the current one.

3.2 The output

The output of the synthesis module is speech. However, since in this report only the first part (viz. FILL_FRAMES) of the synthesis module is described, the output of FILL_FRAMES (which thus is input to AP2) is a frame buffer such as is common in the LVS environment. See for more information on frame buffers Allain (1990).

4 The implementation of SPEECH2_ENV.PAS

The global declarations are given in the file SPEECH2_ENV.PAS. Most of them are straightforward, so not everything will be discussed.

The dollar sign, '`$`', in some of the constants or variables denotes that these are global. On encountering them in FILL_FRAMES one can see that they are not defined locally, but

```
TYPE
  anchor_taip = (vo,eov,prv,none);

  pitch_move_pointer = ^pitch_move;  {     ----------            }
  pitch_move = record                {     | freq    |           }
      taip   : str;                  { --> | decl    |--> pitch_spec_ }
      anchor : anchor_taip;          {     | dur     |    pointer}  }
      frame  : integer;              {     | move    |           }
      time   : real;                 {     ---|------            }
      onset  : str;                  {        V                  }
      offset : str;                  {     ----------            }
      dur    : integer;              {     | taip    |           }
      exc    : real;                 {     | anchor  |           }
      next   : pitch_move_pointer;   {     | frame   |           }
    end;                             {     | time    |           }
                                     {     | onset   |           }
  pitch_spec_pointer = ^pitch_spec;  {     | offset  |           }
  pitch_spec = record                {     | dur     |           }
      freq : real;                   {     | exc     |           }
      decl : real;                   {     | next    |           }
      dur  : integer;                {     ---|------            }
      move : pitch_move_pointer;     {        V                  }
      next : pitch_spec_pointer;     {     pitch_move_pointer    }
    end;
```

Figure 3: The pitch_spec_pointer data structure

globally in SPEECH2_ENV.PAS. I defined the first four constants because I didn't want to work with meaningless numbers, although this may be awkward to someone who is used to working in the LVS environment (in which case the numbers would not be meaningless). Now, I hope to achieve one has a somewhat better notion of the meaning of the parameter which is being altered.

The 'pitch_spec_pointer' deserves some discussion, since it is the input data structure for FILL_FRAMES. It is given in figure 3. Each pitch_spec_pointer corresponds with an intonation domain, be it a filled or an empty one. If it is an empty one, the dur field should contain the duration of the pause, and the move field should be NIL. The other fields may contain arbitrary values, but preferably are set to zero.

For filled intonation domains all fields must contain correct values. The fields freq and decl must contain the starting frequency and the declination of the filled intonation domain, respectively. The move field must now point at a list containing the movements of the intonation domain. The respective movements are specified in the pitch_move record, the various aspects of which are stored in the various fields. This is discussed in more detail in section 3.1.

The various '...$_...' (dip$_fil to frame$_buf) arrays and variables are filled with information in SPEECH2 (by means of various RECEIVE_... routines), and used in FILL_FRAMES.

The other variables are mainly used in 'speech2' (I do not know why they are put here, in SPEECH2_ENV, rather than in SPEECH2 itself). The same goes for the procedures, they are not used in FILL_FRAMES. Probably, this is due to its historically determined origin.

Thus, it might be well possible that a somewhat better structure could be achieved (with variables placed where they belong), but currently the priorities are such that these minor improvements will not be implemented.

5 The implementation of FILL_FRAMES.PAS

The module FILL_FRAMES consists of two parts. One part fills the framebuffer with diphone data and sets the duration parameter for each frame, the second part sets the pitch parameter for each frame. The first part is realized by routine FILL_FRAME_BUF, the second by routine DETERMINE_PITCH.

5.1 FILL_FRAME_BUF

The routine FILL_FRAME_BUF determines the value of the duration parameter (`'$dur' = 17`) for each frame. It essentially loops only once over all diphones (see the source code, line 646). The variable `'dip$_index'` is incremented by the routine NEXT_DIPHONE, which returns the name of the next diphone. The routine TREAT_DIPHONE reads the diphone into the framebuffer and does some smoothing if allowed.

Then, the routine PROCESS_NEXT_PHONEME sets the duration for the 'current' phoneme. This is somewhat intricate. If a diphone is read, say the second one in the example, 'D1I1', it completes the first phoneme, this is /d/ (or the second if you want to count /SI/ as a phoneme). First then the duration of this phoneme can be adjusted (the /d/). The first parameter of this routine indicates out of how many parts the phoneme is built up. Generally, this is two, but the first and the last phonemes (viz. the /SI/ phonemes) consist only of one part, and triphones of three.

Inside PROCESS_NEXT_PHONEME, the 'nr_of_parts' determines the total number of frames out of which the phoneme is built up. Together with the NEXT_DURATION, the duration per frame is determined in ADJUST_DURATION (simply by means of a division). However, the way to store it in the framebuffer is not in absolute milliseconds, but in a percentage of the global frame duration, which is expressed in samples per frame. Therefore, some arithmetic juggling takes place in SET_DURATION. See the comments in the source code for further explanation.

As one can see in the source code of routine FILL_FRAME_BUF, inside the **while** loop PROCESS_NEXT_PHONEME is called with 'nr_of_parts' having the value 2 (corresponding with normal phonemes), and before and after the loop it is called with value 1 (corresponding with the boundary phonemes). Inside the loop, however, as an exception, the routine can also be called an extra time with value 1. This corresponds with a triphone.

A triphone (only occurring in the ZELLE diphone set for /h/ phonemes) is represented in the grid as in (1):

phon.segm:	a		h	a	
phon.dur:	120		50	120	
frames.nr:	4	10	3	5	9
diph:		AA1H1AA1			

(1)

Suppose that TREAT_DIPHONE has just read the 'AA1H1AA1' 'diphone'. Then the first call of PROCESS_NEXT_PHONEME with value 2 will set the duration of the first /a/ (processing the frame counts 4 and 10). Then, since 'AA1H1AA1' is a triphone, PROCESS_NEXT_PHONEME is called once again with value 1, thus setting the duration for the /h/ and processing only

one frame count, viz. 3, which is exactly what we want, since here the phoneme /h/ consists only of one part!

In FILL_FRAME_BUF, after each call to PROCESS_NEXT_PHONEME a check is performed in order to see if the actual number of frames read in the framebuffer corresponds with the number specified by Spraakmaker. If it does not, an error message is issued. When this occurs in a session in which the sentence has just been typed, this error should be taken seriously, i.e., someone should dive into the source code. If it occurs when a grid has been undumped, the error indicates that a different diphone file has been used to compute the frame counts than to synthesize the speech, or that one has edited in the grid such that the integrity of the data has been lost.

5.2 DETERMINE_PITCH

The routine DETERMINE_PITCH determines the value of the pitch parameter ('`$pitch`' = 1) for each frame. It consists of three parts, COMPUTE_ABS_TIME, which transforms the frame numbers into an absolute time in milliseconds, GENERATE_CONTOUR, which generates a straight-line representation of the pitch contour (a list of slopes and durations), and COMPUTE_HERTZES, which samples the straight-line representation at the moments a frame begins.

5.2.1 COMPUTE_ABS_TIME

The routine COMPUTE_ABS_TIME only looks at anchored moves. For the anchored moves, 'vo' and 'eov', it stores the specified frame number in the variable '`target`', and it increments the actual frame number '`frame_count`' until '`target`' is reached. While doing so, it integrates the duration of the individual frames in the variable '`time`'. This results in '`time`' containing the time in milliseconds at which the frame '`target`' will be synthesized. The time at which the anchored movement actually starts can then be computed from the fields `onset`, `offset` and `dur`, as specified in the source code, and is stored in the `time` field.

5.2.2 GENERATE_CONTOUR

The routine GENERATE_CONTOUR expects as input a list of 'idom specifications', a list of pitch movements for each intonation domain. In practice, this list will consist of alternating 'empty' and 'filled' (see above) intonation domains. As its output, it computes a straight-line representation of the pitch contour. This consists of one starting frequency in Hertz, and a list of slopes and durations, where the slopes are given in semitones per second and the durations in milliseconds, which appear to be the natural units to use. The straight-line representation is stored in the data structure '`sent_contour`' given in figure 4. The straight-line representation represents the pitch as function of time for the *whole* sentence. So, the list of 'idom specifications' (each of which may contain a list of movements) which is put in, will be put out in *one* single list, representing both the empty and the filled intonation domains.

Although the first element of the straight-line representation will always concern an empty domain, lets first discuss the filled domains. All filled domains contain a specification of the starting frequency. The starting frequency of the first filled list will be the starting frequency of the straight-line representation. The starting frequencies of the succeeding intonation domains will be accounted for by adjusting the slopes in the straight-line representation for those parts that correspond with the 'empty' intonation domains. The task is thus to determine the slopes and durations.

```
type
  contour_pointer = ^contour_element;   {                    --------    }
  contour_element = record              {                    | freq |    }
      slope : real;                     { sent_contour :     | cont |    }
      dur   : real;                     {                    ---|----    }
      next  : contour_pointer;          {                       |        }
    end;                                {                       V        }
                                        {                    ---------   }
                                        {                    | slope |   }
  sent_contour = record                 {                    | dur   |   }
      freq : real;                      {                    | next  |   }
      cont : contour_pointer;           {                    ---|-----   }
    end;                                {                       V        }
                                        {              contour_pointer   }
```

Figure 4: The data structure used for the straight-line representation.

For the 'filled' intonation domain this is done in the inner **while** loop of the body of GENERATE_CONTOUR (line 1034). One can only determine the slopes and durations of unanchored movements if one knows where the enclosing anchored movements end and start. So, when encountering an anchored movement ('vo' or 'eov'), two elements (viz. the unanchored and the anchored, respectively) must be added to the list, for unanchored movements ('none'), none. Only for 'prv' movements exactly one element must be added to the list.

Thus, for unanchored movements only the extra amount of excursion must be stored, plus a check if one has not specified two adjacent unanchored movements. For a 'prv' movement (see PRV_CASE, line 984), all specifications can be directly found in the 'pm^' fields. By means of ADD_MOVE the movement is added to the list. Before it is actually added by means of NEW_SLOPE it is checked whether the added element will exceed the specified duration of the intonation domain. If so, the element is truncated, all further movements are ignored, and a warning message reporting this is issued. The amount of processed time is updated, and the excursion is reset. Should a 'prv' movement directly follow an unanchored movement, an error message is issued, but the program is continued and the unanchored movement will not be realized (since, how would *you* try to realize it?).

The anchored movement is certainly the most complicated case. In principle, it adds two elements to the list, the unanchored and the anchored movement. This is the 'normal case', which very much resembles the case of a 'prv' movement, except for the fact that in this case the preceding unanchored movement is added to the list first (see NORMAL_CASE, line 969). However, it is possible that the duration of the unanchored movement becomes negative, for instance because two anchored movements have a slight overlap. In that case, no element should be added to the list for the unanchored movement, and the element that is added to the list for the anchored movement should be altered a bit. This case, that the duration becomes negative, is further divided into two cases, the 'overlapping case' and the 'initial case'. In the overlapping case two anchored movements have a certain amount of overlap, in the initial case the first anchored movement starts before the 'starting_time' of that intonation domain.

The 'overlapping case' (see OVERLAPPING_MOVEMENTS, line 946) is the case which is actually confronted with conflicting information: two anchored movements overlap. It tries

to make the best of it, depending on the type of movements. If the movements have a different absolute slope they are assumed to be of 'equal strength' and both movements are shortened equally. If the movements are of the same absolute slope, the last movement is shortened by the full amount of overlap.

An example of the first case are an 'a' and a '2' that overlap because the syllable is too short (this will be the most common case), or a '1' and an 'a' (this will hardly occur). In this case the duration of the two movements is equally shortened and the slope is retained. If the movements have equal (absolute) excursions, the resulting pitch will be the same as in the case that there is no overlap. Of course, if the absolute excursions are not equal then this is not true. In any case, only the durations are adjusted and the slopes are retained.

An example of the second case are an 'a' and a 'fl' that overlap because the syllable is too short. In this case the 'a' is fully realized, and the 'fl' is shortened by the amount of overlap. Since both movements have a negative slope this is better than adjusting the duration of both. This ad-hoc convention has been discussed with Jacques Terken and awaits a better proposal. It is based on the assumption that the first movement is perceptually dominant (since it often concerns an accent-lending movement, whereas the second does not; the second one is probably anchored at 'eov', otherwise there would not be an overlap).

It can happen, of course, that the amount of overlap is too big, and that for instance the previous movement is fully 'eaten' by the current; the duration of the previous movement does not suffice to give half of the overlap (case one, see INFORM_USER_ABOUT_OVERLAP, line 864) or the current movement does not suffice to give the full overlap (case two, see INFORM_USER_ABOUT_SHORTENING, line 846). In such a case an error message is issued, reporting that the pitch contour is ill-specified. The program continues, but the pitch contour that will be generated may be totally unacceptable. In the other case, that there is overlap but the movements are not 'eaten', a warning message is issued so as to inform the user or rule developer that the synthesis module has autonomously removed the overlap. In this case the resulting contour should sound acceptable.

The 'initial case' is the case that the first movement starts before the starting time of the intonation domain. This is typically the case if the first syllable is to be accentuated. In this case not only must the unanchored movement be omitted from the list, also the starting frequency must be adapted. That is to say, if it concerns the first filled intonation domain. The straight-line representation is suspended at the starting frequency, and normally this is the frequency given in the first intonation domain of the current sentence. Note that in the grid each intonation domain has a starting frequency, whereas the straight-line representation only has one. The starting frequencies of the other intonation domains are used to determine the slopes in the straight-line representation corresponding with the empty domains.

The initial case also has two subdivisions. One is if the first anchored movement starts at $t < t_0$ (t_0 is the starting time of the intonation domain, '`start_time`') but ends $t > t_0$, i.e., the movement partly falls before t_0. The second is if the first anchored movement also ends at $t < t_0$, i.e., it *fully* falls before t_0. The first case is well possible in normal situations, i.e., could well follow from a normal specification, the last case should not occur in normal situations, however, one cannot exclude the possibility that for some reason it has been specified.

In the first case, the case that the movement partly falls before t_0, two actions are taken (see MOVEMENT_PARTLY_BEFORE, line 928). The first is the computation of the new starting frequency. If it concerns the first intonation domain (in this case '`i_contour = curr_element`'), the new starting frequency is stored in the appropriate field. If not, the slope of the element in the straight-line representation corresponding with the 'empty' intonation domain must be adjusted such that the new starting frequency is reached

at t_0 (see PROCESS_NEW_FREQ, line 883). The second action is that the duration of the anchored movement is adjusted (viz. to become the length of the part that falls in $t > t_0$) and that the element is added to the straight-line representation (as in the overlapping case). In the second case, the case that the movement fully falls before t_0, only the first action is taken: computation and installation of the new starting frequency (see MOVEMENT_FULLY_BEFORE, line 916). In this case, not only the slope of the anchored movement is needed, also the amount of extra declination is taken into account.

With this, the 'filled' intonation domains have been discussed. Now, it is easy to understand how the 'empty' intonation domains are treated (see EMPTY_INTO_CASE, line 783). Actually, only the duration of the empty intonation domain is of importance. The frequency at the end of a filled intonation domain results from the specified movements, and the starting frequency of the next filled intonation domain is given in the grid. The duration of the empty domain is used to 'connect' these two. One element, corresponding to the empty domain, is added to the straight-line representation, such that the starting frequency given in the grid is reached.

If the empty domain lies between two filled intonation domains, this is straightforward: compute the slope by means of the routine ST_SLOPE and add it to the list. There is no need to check whether it exceed the end of the intonation domain, since it is specified to exactly reach it. However, each sentence specification always begins and ends with an empty domain, so these are not enclosed by filled domains (and thus have given frequencies). In these cases a slope of zero semitones/second is installed. This is achieved by making the beginning and end frequencies of the slope equal.

With this, most of the specifics of GENERATE_CONTOUR have been discussed. Two minor points remain. One is that the duration of the last element in the straight-line representation is prolonged with $1/1000^{th}$ of a millisecond to ensure that the length of the straight-line representation is (somewhat) longer than the length of the framebuffer. Since we are computing with reals, due to rounding errors one cannot guarantee that the length is exactly the same.

The second is that in the implementation use is made of separate small routines without parameter passing. Although this is known to be dirty, it has been applied all the same. The reason for this is that quite a lot of parameters should be passed to quite a deep level. If they would be passed in the regular way, to my feeling this would obscure the source code more than is the case now. Those who disagree are invited to rewrite my code.

5.2.3 COMPUTE_HERTZES

The task of COMPUTE_HERTZES is to sample the straight-line representation (which is a continuous representation of the pitch contour as a function of time) at the starting moments of each frame (the framebuffer is a discrete data structure). For each frame in the framebuffer the pitch parameter ('`$pitch`' = 1) must be set. Since each frame starts at a distinct moment, the pitch of that moment can be computed from the straight-line representation. This sampling is done in the **for** loop of the routine.

Since the duration and the pitch are not directly stored in milliseconds and hertzes in a frame, some arithmetic juggling is necessary to store and retrieve the right values (see Allain (1990)).

The invariant of the loop is that the frequency has been computed up to the time of the current frame (`frame_nr`). Thus, in the first statement of the loop, the current frequency is stored in the current frame.

Next, the duration of the current frame is computed. Probably, the **if** statement to compute this in the default case is not necessary, since in this implementation the relative

duration always has a value unequal to zero, but the statement adds to the robustness of the routine.

Then, the new frequency belonging to t = (current time + duration current frame) (which is the time the next frame starts) is computed, in order to preserve the precondition. This is done by decrementing the duration of the current slope in the straight-line representation by the duration of the current frame while accounting for the slope: the new frequency is computed given the old frequency, the slope and the duration. These are the last two statements of the loop. Every once in a while, however, the duration of the current slope of the straight-line representation is shorter than the duration of the frame (since, for each frame the duration of the current element in the straight-line representation is decremented). In that case, first the last part of the current element is accounted for (the **while** loop), and then the first part of the next element is taken (with adjustment of the duration which is to be accounted for). A **while** loop is chosen (rather than an **if** statement) since it is conceivable that the duration of one frame spans more than one element in the straight-line representation (although this will generally not be the case). Thus the precondition is restored.

When all frames have been treated the work is done. Some checks are added after the **for** loop to ensure there has been no error in the line of thought.

5.3 Other units

Throughout the routine GENERATE_CONTOUR the computation of the frequencies is done via separate routines, ST_FRQ and ST_FREQ. The 'ST' in these routines stands for 'semitones', which currently is the unit used to express excursions and declination in the grid. Computation of frequencies via a separate routine is done in order to enable an easy shift to other units (e.g. ERBs, (=equivalent rectangular bandwidths)). To shift to other units, only these routines have to be adjusted. Currently, the routine ST_FRQ computes a new frequency (in Hz) given the old one (in Hz) and an excursion (in semitones). The routine ST_FREQ computes a new frequency given the old one, a slope (in semitones/second) and a duration (in milliseconds).

A third 'ST' routine is also available, ST_SLOPE. This routine computes the slope (in semitones/second) of an element in the straight-line representation, given the beginning and end frequency (in Hz) and the duration (in milliseconds). It takes some mathematical insight to see that this routine will also suffice to adjust a previously computed slope (in PROCESS_NEW_FREQ) in the 'initial case', but in fact one may always summon slopes like this.

6 References

Allain, P. (1990). LSV programmers manual, Manual no. 107, *Institute for Perception Research*, 120 pages.