On the consistency of the grid

Institute for Perception Research
P.O. Box 513 - 5600 MB Eindhoven

HvL/hvl 92/19
20.10.1992

Rapport no. 871

On the consistency of the grid

H.C. van Leeuwen

On the consistency of the grid

Hugo C. van Leeuwen

Abstract

A number of currently popular text-to-speech systems are built around a multi-level, synchronized data structure, called a *grid*. In such a structure, the different levels are synchronized with each other at strategic points by means of so-called sync marks. In order to make sense, in the grid the sync marks may not cross, or in other words, the grid must be consistent. As each change to the synchronization relations in the grid may affect its consistency, in principle for each change a consistency check must be performed. Therefore, for an acceptable run-time performance we need an algorithm that is sufficiently fast.

In this paper, the problems related to this issue are discussed. It is first addressed why consistency is needed, and then what part of consistency checking can be done during compile-time and what must be done at run-time. For efficient algorithms for both tasks, a proper internal representation of the grid is of essential importance; a straightforward representation results in a too slow run-time performance of the text-to-speech system. It is shown that by representing the grid as a graph, order N in time complexity can be saved, where N is the number of sync marks. Algorithms for both the compile-time and the run-time parts of consistency checking are given, as well as some statistical figures as to how often this is needed in a practical situation.

1 Introduction

This paper discusses a theoretical problem that is encountered in the application domain of text-to-speech systems that are built around a multi-level, synchronized data structure. Examples of such systems are *Speech Maker* (van Leeuwen & te Lindert, forthcoming), *Delta* (Hertz, Kadin & Karplus, 1985), and systems described by Boves (1991) and Lazzaretto & Nebbia (1987). This multi-level structure, which we have called a *grid*, consists of a number of *streams* (the levels) which are *synchronized* with each other at strategic points by means of *sync marks*. An informal description of the grid data structure can be found in van Leeuwen & te Lindert (forthcoming) and Hertz *et al.* (1985). Below, a formal description of the grid is given.

The theoretical problem that is encountered is that of keeping the grid consistent, and a directly related practical problem is the problem of keeping the run-time performance of the text-to-speech system within acceptable limits. Consistency means that the sync marks may not 'cross', and in the text-to-speech setting one wants the grid to be consistent. Ensuring the grid's consistency turns out to be a non-trivial problem and as it must be done often, it may well become a substantial part of the run-time performance of the system. In this paper we will show that the choice of a proper internal representation of the grid is essential; a straightforward implementation of the grid does not suffice. The problem is especially relevant if one wishes to manipulate the grid by means of rules, a typical demand in the text-to-speech setting (van Leeuwen, submitted).

We start by giving a formal description of the grid and a definition of consistency. Then we discuss why the grid should be consistent and how consistency can be maintained. It is shown

that if one manipulates the grid by means of rules, one cannot fully guarantee consistency at compile-time, and thus, one has to do some consistency checking at run-time. On the other hand, quite a lot of work can be done at compile-time. To minimize the execution time, there are two points of interest: (a) how to maximize the compile-time contribution and (b) how to optimize the run-time consistency checks. For both aspects explicit algorithms are given. Finally, we also give some practical figures: given M rules, how many run-time checks are performed, and given N sync marks, how long a run-time consistency check takes, and what typical values of M and N are.

2 The grid from a mathematical point of view

In this section a mathematical description of the grid is given. The grid consists of a number of streams. Each stream consists of a sequence of sync marks, between which linguistic data can be stored. The sync marks serve to synchronize streams with each other at strategic points. Thus, there are two aspects to the grid, a horizontal and a vertical: horizontally the relations between sync marks are ordinal (one can speak of relations such as 'to the left of', 'to the right of', or 'equal to'), vertically they are nominal (one can only speak of 'equal to' or 'not equal to').

2.1 Horizontal relations

For the present discussion, a stream may be viewed as an ordinal arrangement of sync marks. Denote each sync mark in the grid by S_{in}, where i is the stream index (and thus constant within a stream) and n is the rank order of the sync mark within the stream. Here, we express the ordinal arrangement of the sync marks in a stream by means of the operators '$<$' and '$>$'. They denote the relation 'left of' and 'right of', respectively. We define:

$$\forall n \in S_i : \quad S_{in} < S_{i[n+1]} \tag{1}$$

Further, we define

$$S_{im} < S_{in} \quad \Leftrightarrow \quad S_{in} > S_{im} \tag{2}$$

The operator is transitive, which means

$$\left.\begin{array}{l} a < b \\ b < c \end{array}\right\} \Rightarrow a < c \tag{3}$$

Thus, one can derive:

$$\forall i : m < n \quad \Rightarrow \quad S_{im} < S_{in} \tag{4}$$

2.2 Vertical relations

Between streams, synchronization may exist. This is a nominal relation, viz. it either exists or does not exist at a certain place. Thus, with respect to synchronization, we can only speak of equality or inequality. We define two streams, say i and j, to be synchronized with each other at point m for S_i and point n for S_j if

$$S_{im} = S_{jn} \quad \text{where } i \neq j \tag{5}$$

Equality, too, is transitive:

$$\left.\begin{array}{l} a = b \\ b = c \end{array}\right\} \Rightarrow a = c \tag{6}$$

2.3 Context and inequality relations

We can now combine the horizontal and vertical relations, and derive:

$$S_{im} = S_{jn} \rightarrow \begin{cases} p < m \rightarrow S_{ip} < S_{jn} \\ p > m \rightarrow S_{ip} > S_{jn} \end{cases} \tag{7}$$

Proof:

$$\left.\begin{array}{l} S_{im} = S_{jn} \\ p < m \end{array}\right\} S_{ip} \overset{(4)}{<} S_{im} = S_{jn} \quad \Rightarrow \quad S_{ip} < S_{jn} \tag{8}$$

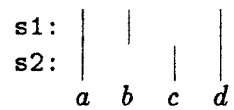
$$\left.\begin{array}{l} S_{im} = S_{jn} \\ p > m \end{array}\right\} S_{ip} \overset{(4)}{>} S_{im} = S_{jn} \quad \Rightarrow \quad S_{ip} > S_{jn} \tag{9}$$

Thus, we can extend the meaning of the $<$ and $>$ operators to describe the relation between sync marks that are not in the same stream. If $a < b$, where a and b are arbitrary sync marks in the grid, then we will say a is in the *left context* of b, or, as $a < b \Leftrightarrow b > a$, that b is in the *right context* of a.

However, if we have two arbitrary sync marks a and b, the three operators presently defined, equality and context relations ($a < b$, $a = b$ and $a > b$), do not suffice to describe all possible relations between a and b. Consider two streams, S_1 and S_2, and let

$$\begin{array}{l} S_{11} = S_{21} = a \\ S_{12} = b \neq S_{22} = c \\ S_{13} = S_{23} = d \end{array} \tag{10}$$

This grid looks as follows:



(11)

Considering definition (1) we have:

$$\begin{array}{l} S_1 : a < b < d \\ S_2 : a < c < d \end{array} \tag{12}$$

Now, if we consider the relation between S_{12} and S_{22}, we know that $S_{12} \neq S_{22}$, but neither can we prove $S_{12} < S_{22}$ or $S_{12} > S_{22}$. The sync mark are *unrelated*, which we denote as follows: $S_{12} \bowtie S_{22}$. We use the symbol '$\bowtie$', as we want to reserve a slash trough an operator as its boolean inversion (so $a \neq b$ does not mean the same thing as $a \bowtie b$). The relation $S_{im} \bowtie S_{jn}$ between two sync marks is true if and only if none of the other relations are true:

$$\left.\begin{array}{l} S_{im} \neq S_{jn} \\ S_{im} \not< S_{jn} \\ S_{im} \not> S_{jn} \end{array}\right\} \Rightarrow S_{im} \bowtie S_{jn} \tag{13}$$

Within a stream, sync mark are always related: a sync mark is either to the left or to the right of another or equal to itself. Only sync marks in different streams can be unrelated, as shown above. Thus, there are four basic relations that can exist between two arbitrary sync marks in the grid: $a < b$, $a = b$, $a > b$ and $a \bowtie b$.

Note that unrelatedness, in contrast to equality and context relations, is not transitive:

$$\left.\begin{array}{l} a \bowtie b \\ b \bowtie c \end{array}\right\} \not\Rightarrow a \bowtie c \tag{14}$$

This can be seen by taking for instance $a = c \neq b$. In fact, with $(a \bowtie b) \wedge (b \bowtie c)$, you still know nothing of the relation between a and c, it can be any of the four cases.

3 Consistency

We define a grid to be inconsistent if:

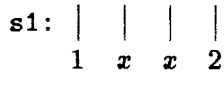
$$\exists i, n \mid S_{in} < S_{in} \tag{15}$$

A grid is consistent if it is not inconsistent:

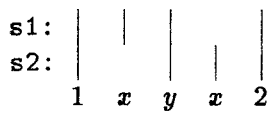
$$\neg\exists i, n \mid S_{in} < S_{in} \quad \Leftrightarrow \quad \forall i, n \; : \; \neg(S_{in} < S_{in}) \tag{16}$$

The definition of consistency, thus, is an indirect one, based on the one defining inconsistency. Inconsistency can be interpreted as a loop in the grid: shifting either to other sync marks only in a fixed direction or to the 'same' sync mark in another stream, there exists a sync mark for which one returns to the same sync mark after a finite number of steps. Often, one can also view this as sync marks that cross: one that is left to another in one stream, is to the right of that sync mark in another stream.

Thus, in order to to determine whether a grid is consistent or not, one has to examine all sync marks and all paths through the grid. Examples of inconsistent grids are:



(17)



(18)

(19)

Grid (19) is an example of a grid where none of the streams contains both x and y, or a and b. Although none of the sync marks x, i, a, y and b actually cross the grid is inconsistent; all can serve as the starting point of the loop.

3.1 What is wrong with an inconsistent grid?

The idea of the grid is that the sync marks, which separate the tokens from one another, have a qualitative relation to time: if a sync mark a is placed to the left of sync mark b, the part of the utterance corresponding with a is uttered earlier then the part corresponding with b. Now, if the grid is inconsistent due to a loop including a and b, the relation to time is violated: one can say $a < b$ just as well as $b < a$.

Mathematically spoken, when the grid is inconsistent due to a loop including a, the categories $x < a$ and $x > a$ are not disjunct any more. A inconsistent grid is a different type of mathematical object, of which very little interesting properties can be proven.

The above considerations are somewhat theoretical, after all, the fact that the relation to time is violated is something that may also occur in real life. In Dutch, for instance, we pronounce numbers backwards: '43' is literally pronounced as 'three and forty'; so in the pronunciation utterance the order of the '4' and the '3' is reversed. The mathematical consideration is more important; it is awkward when categories defined by $<$ and $>$ are not disjunct any more, although it *is* conceivable.

There is, however, a very practical objection to an inconsistent grid: SMF rules may become ambiguous. SMF stands for Speech Maker Formalism. This is a dedicated computer language with which one can manipulate a grid by means of rules. It is described in van Leeuwen (submitted). Here, we assume the reader is familiar with the formalism.

A grid can be inconsistent due to a loop or due to the fact that a sync mark occurs more than once in a stream (this is in fact the simplest form of inconsistency). The former can be transformed to latter by means of a simple SMF rule, and the latter gives rise to ambiguity in the interpretation of an SMF rule. This last point is clearly undesirable, since then the operation of the formalism becomes unpredictable and undependable.

For example, consider pattern (20):

```
i: | a |                                                    (20)
k: | c
   x   y
```

To the right of sync mark x a 'c' must be found in stream k. Now consider grid (21):

```
i: |  a | q |
k: |  c | d |                                               (21)
     x   xy
```

Here, sync mark x is present twice in stream k, and depending on the one you pick a 'c' or a 'd' is found, making the pattern match or fail, respectively.

A grid like (21) can easily be made once a grid is inconsistent. Consider, for instance, grid (22):

```
i: | p |  a  | q |
          \ /
j: |     | b | s |                                         (22)
k: |     |   |   |
```

then a valid rule like (23) will put sync mark x twice in stream k:

```
i: | a |                                                    (23)
j:     | b |  -->  k: | c |
  x       y           x   y
```

Thus, any inconsistent grid may be transformed into a grid with a stream that contains some sync mark twice, which makes the interpretation of an SMF rule ambiguous. Therefore, we do not want the grid to become inconsistent, and hence, we must ensure with each operation we perform on the grid that it remains consistent.

4 How to maintain consistency

Consistency is a matter that concerns the whole grid. As stated, to establish that a grid is consistent, one needs to consider all sync marks. We therefore approach the matter by means of induction: we assume the grid is consistent and consider how the grid can become inconsistent.

A consistent grid can only become inconsistent by means of an action that alters the synchronization relations. There are only two basic types of rules: *insertion* rules and *substitution* rules. An example of an insertion rule is rule (24), which aligns a 'b' in stream s2 with the 'a' in stream s1:

```
s1: | ... | a | ... | --> s2: | b |                        (24)
      ^     x   y      ^         x   y
```

A example of a substitution rule is rule (25), where sync mark x in stream s2 is replaced by sync mark y:

```
s1: | ... | a | ... |  --> s2: |==>|
    ^     x   y     ^           x   y
```
(25)

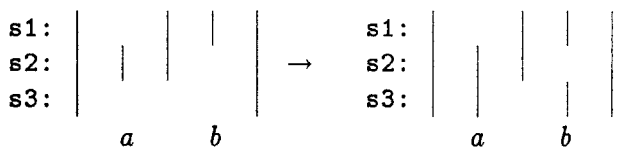
For both types of rules it will now be discussed how they can introduce inconsistency.

Insertion rules

The action part may consist of several lines, each of which specify a certain alteration to be made to the grid. Each such line is called an action line, or AL. An AL always concerns exactly one stream. Each AL must at least contain one bound sync mark, i.e., at the time the AL is executed at least one of the sync marks of the AL must be related to a sync mark in the grid; otherwise the rule interpreter does not know where to insert the data specified in the action part. If the bound sync mark is present in the stream, that is where the data will be inserted, if it is not, that sync marks will be inserted first in the stream at the appropriate place.

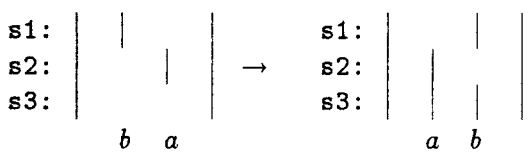
In any case, if the AL contains only one bound sync mark, the consistency of the grid cannot be disturbed. Only when two or more bound sync marks are involved can this occur. An example is given in section 5, where dependent of the grid a rule may be executed or not. The most simple insertion rule for which integrity problems can arise is a rule that contains an AL that involves two bound sync marks. The reason is that this AL explicitly introduces a possible new relation, viz. $a < b$. Now, in general there are four situations possible in the actual grid: $a < b$, $a \bowtie b$, $a = b$ and $a > b$ (see section 3).

1. $a < b$. The new relation is in accordance with the existing situation. The specified AL may be executed, and there is no essential change in the set of relations that are described by the grid. Example:



(26)

2. $a \bowtie b$. The new relation does not conflict with the existing situation. The specified AL may be executed, but in this case there is an essential change in the set of relations that are described by the grid, viz. $a < b$ is added. Example:



(27)

3. $a = b$. The new relation conflicts with the existing situation. Normally a and b will represent different sync marks, but it is possible in SMF to have them refer to one and the same sync mark. One can prove that in this case an inconsistency will occur:

$$\left.\begin{array}{l} S_{im} = a \\ S_{i[m+1]} = b \\ a = b \end{array}\right\} \Rightarrow S_{im} \overset{(1)}{<} S_{i[m+1]} = b = a = S_{im} \tag{28}$$

4. $a > b$. The new relation conflicts even more clearly with the existing situation. This is a case that can occur, for example in grid (29):

(29)

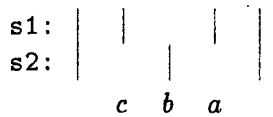
One can prove that an inconsistency will occur in an analogous way:

$$\left.\begin{array}{l} S_{im} = a \\ S_{i[m+1]} = b \\ a > b \end{array}\right\} \Rightarrow S_{im} \overset{(1)}{<} S_{i[m+1]} = b < a = S_{im} \tag{30}$$

As a result, when an AL introduces a new relation $a < b$, we must be able to establish that in the existing situation $a \not\geq b$, which is equivalent to proving that either case 1 or 2 is present, viz. $(a < b) \vee (a \bowtie b)$.

Of course, the action part can consist of more than one AL, and each can define more than one new relation. However, this can be viewed as the repeated execution of a number of basic actions (a basic action introduces only one new relation in the grid). This means, however, that in determining whether the insertion of a bound sync mark may be executed, the state (the set of sync mark relations) of the grid *at that moment* should be taken, in which (new) relations due to previous insertions of the *current* rule are included.

For instance, consider grid (31) and the rules (32) and (33):



(31)

```
... -> s3: | | |
           a b c
```

(32)

```
... -> s3: | |
      s4:   | |
           a b c
```

(33)

In grid (31), $a \bowtie b$ and $b \bowtie c$, but it is incorrect to conclude that thus $a \bowtie c$; unrelatedness is not transitive, as was shown in (14). And note that $a < c$ is an implicit relation of $a < b < c$ that is raised by the ALs. Both rules (32) and (33) can be seen as composed of two parts, one which defines the relation $a < b$ and a second which defines $b < c$. The first part does not conflict with the existing situation, neither does the second. But once $a < b$ is included, the second relation conflicts with the then present situation, which is: $c < a < b$.

Substitution rules

For the substitution rule the only thing that happens with respect to synchronization is that in one stream, one sync mark, a, is replaced by another, b. The only relations between sync marks in the grid that can change are the relations within that stream. Thus, the only way to make the grid inconsistent is that b (which is the new value of S_{im}) conflicts with its neighboring sync marks $S_{i[m-1]}$ and $S_{i[m+1]}$. The requirements to keep the grid consistent can be expressed mathematically as follows:

$$\begin{cases} b \not\leq S_{i[m-1]} \\ b \not\geq S_{i[m+1]} \end{cases} \tag{34}$$

Note that this is not equivalent to $S_{i[m-1]} < b < S_{i[m+1]}$ as e.g. $b \not\leq S_{i[m-1]}$ means that $(b > S_{i[m-1]}) \vee (b \bowtie S_{i[m-1]})$.

5 compile-time versus run-time checking

In this section we will first show by means of an example that one cannot guarantee the consistency of the grid during compile-time, and thus that one cannot circumvent the necessity to determine context relations at run-time. Then, we will discuss qualitatively what can be done compile-time.

5.1 Consistency-checking is a run-time issue

Consider rule (35), which is a perfectly valid SMF rule:

```
s1: | a |  b |                                                    (35)
s2: | c  | d | --> s3: | e | f | g |
    ^   xy   ^          ^   x   y   ^
```

The pattern part of rule (35) matches on the following three grids, (36), (37) and (38). All grid are perfectly valid, that is, any of them could be the actual state when the rule is matched against it.

```
s1: | a |  b |
s4:
s2:   c  | d |                                                    (36)
s3:
```

```
s1: | a |  b |
s4:   |    |
s2:   c  | d |                                                    (37)
s3:
```

```
s1:   a  | b |
s4:   |  |   |
s2: | c  | d |                                                    (38)
s3:
```

In the first two cases, the execution of rule (35) will result in consistent grids. In the third case, however, an inconsistent grid will result, and thus the rule should not be allowed to be executed. Of course, at compile-time we cannot determine which of the three grids will be actual when rule (35) is executed. Hence, at compile-time, we cannot *guarantee* that the grid will remain consistent, and hence, we must do *some* consistency-checking during run-time.

5.2 *Context relations*

The question now is: which are the pairs of sync marks of which we must determine the context relations during run-time, in other words, of which pairs are we unable to determine this during compile-time.

An SMF rule consists of a pattern part (that is matched to the grid) and and action part (that alters the grid if the pattern part matches). Just as the action part introduces possibly new relations in the grid, the pattern part checks for existing relations. As the action part will only be executed if the pattern part matches, in determining which of those relations are to be checked run-time, we may assume the relations in the pattern part to be valid. For instance, consider pattern (39):

```
|  |     |
|     |  |
a  b  c  d
```
(39)

If this pattern matches the grid, we know that $a < b < d$ and $a < c < d$. So, in the case of rule (35), we only have to check during run-time that $x < y$ does not conflict with the existing relations in the grid.

In general, one can say that each line of the pattern part gives us a number of relations that we may assume to be valid. Each AL gives a number of relations that must be met by the grid, in order to guarantee its consistency. Thus, all relations that are not already met by the pattern relation must be checked during run-time. If p_{ij} are the sync marks in the pattern part and a_{ij} the sync marks in the action part, then the left side of (40) gives us all the valid relations and the right side gives us all the relations that are to be established. Those relations of the action part that cannot be established by the pattern part are precisely those relations that must be determined at run-time. All other relations can be checked at compile-time.

$$\begin{array}{lll} \text{Pattern}: & p_{11} < p_{12} < \ldots < p_{1n} & \text{Action}: \quad a_{11} < a_{12} < \ldots < a_{1s} \\ & p_{21} < p_{22} < \ldots < p_{2o} & \qquad\quad a_{21} < a_{22} < \ldots < a_{2t} \\ & \vdots & \qquad\quad \vdots \\ & p_{m1} < p_{m2} < \ldots < p_{mp} & \qquad\quad a_{r1} < a_{r2} < \ldots < a_{ru} \end{array} \tag{40}$$

6 Algorithms for consistency checking

All in, we need two algorithms: one that checks a particular relation at run-time, and one that determines at compile-time which are the relations to be checked at run-time. The first algorithm will be called *In_context* as it basically checks whether a is in the context of b. The second will be called *Det_runtime_checks*. The routine *In_context* must be fast, say order N where N is the number of actual sync marks, otherwise we may expect to have severe problems with the run-time performance.

As it will appear, the two routines are closely related. The run-time routine, that determines whether one sync mark is in the context of another, is also needed for the compile-time task of determining which pairs of sync mark must be checked at run-time. Therefore, we first discuss the run-time routine, and then the compile-time routine.

For the remainder of the paper we assume a sync mark to be contained by its own context. Strictly spoken this is in conflict with the earlier given definition, which makes a distinction between $a < b$ and $a = b$. But in order to prevent having to talk about something like context$^+$ all the time we make a small compromise to exactness.

6.1 The internal representation of the grid

The key to an efficient algorithm is to translate the grid to a graph (Even, 1979), where a set of sync marks that are synchronized with each other are represented by a single vertex, and the left/right relations between the sync mark are represented by edges between the vertices. Thus, an edge $a \rightarrow b$ represents the relation $a < b$. Note that in the mathematical description each sync mark of a stream is a separate object (the value of S_{im} may be equal to S_{jn}, but they are different objects), where in a graph sync marks in different streams that are synchronized with one another are a single object. The value of the sync mark becomes the relevant parameter and gives direct access to all relevant information of the sync mark.

There are three possible graphs as an internal representation of the grid which come to mind fairly easy. For distinction, let us call these the *minimal*, the *maximal* or the *direct* graph. The minimal graph is without any redundancy: only unique relations are inserted as edges (see fig. 1a): if $a \rightarrow b \rightarrow d$ then $a \rightarrow d$ would not be coded as it can be derived from $a \rightarrow b$ and $b \rightarrow d$. The maximal graph is with maximal redundancy, all relations between all sync marks are inserted as edges (see fig. 1c): if $a \rightarrow b$ is already present and $b \rightarrow d$ is added, also $a \rightarrow d$ must be added to have all relations between all sync marks available directly. The direct graph is somewhere in between: precisely those relations are coded that are present in the grid (see fig. 1b). Thus, $a \rightarrow b$ is the representation of the fact that there exists a stream in the grid where a is a direct left neighbour of b. This graph may be redundant (some relations will be coded despite the fact that they could be deduced from other relations) but is probably not maximal redundant (not all deducible relations may be present in the grid).

The minimal and maximal graphs are very suited for the context search: in the minimal case there are very few relations (order N) to be considered, in the maximal case we only have to consider the outgoing relations from a given vertex, which is also order N. The direct graph, however, does not benefit from either of the properties of the minimal or the maximal graph: in principle we will have to check all relations (order M) for all vertices (order N), where M is the number of streams, which leads to order M^{N-1} (worst case) if no efficiency measures are taken.

However, there is another aspect to the internal representation of the grid by means of a graph: it must kept up to date. Whenever something changes in the grid (a sync mark is added or removed from a particular stream), this must be reflected in some way in the graph. For the minimal and maximal case it is a non-trivial matter to keep to graphs up to date; changes may have a severe consequences on the graph, which may well cost more than order N to bring about. And changes to the grid are to be expected relatively often: it is a dynamic structure that is meant to change. So, also these changes may not be too expensive.

Thus, for the minimal and maximal case reasonably sophisticated algorithms will have to be devised to keep the graphs up te date, which are not needed for the direct graph: changes to the grid are related directly to changes in the graph (order 1). And as to the context search, fortunately, we can reduce this to order $M \cdot N$ (worst case) by using *memo-ization*, a mechanism that is known from functional programming. Therefore, we will be using a direct graph to represent the grid internally, and use it as the data structure on which the context search will operate.

6.2 Algorithm In_context

Both for insertion and substitution rules one has to establish relations of the type: $a \not\geq b$, in other words, a may not be in the right context of b. The basic routine for establishing this relation is the context checking routine, which determines whether an arbitrary sync mark

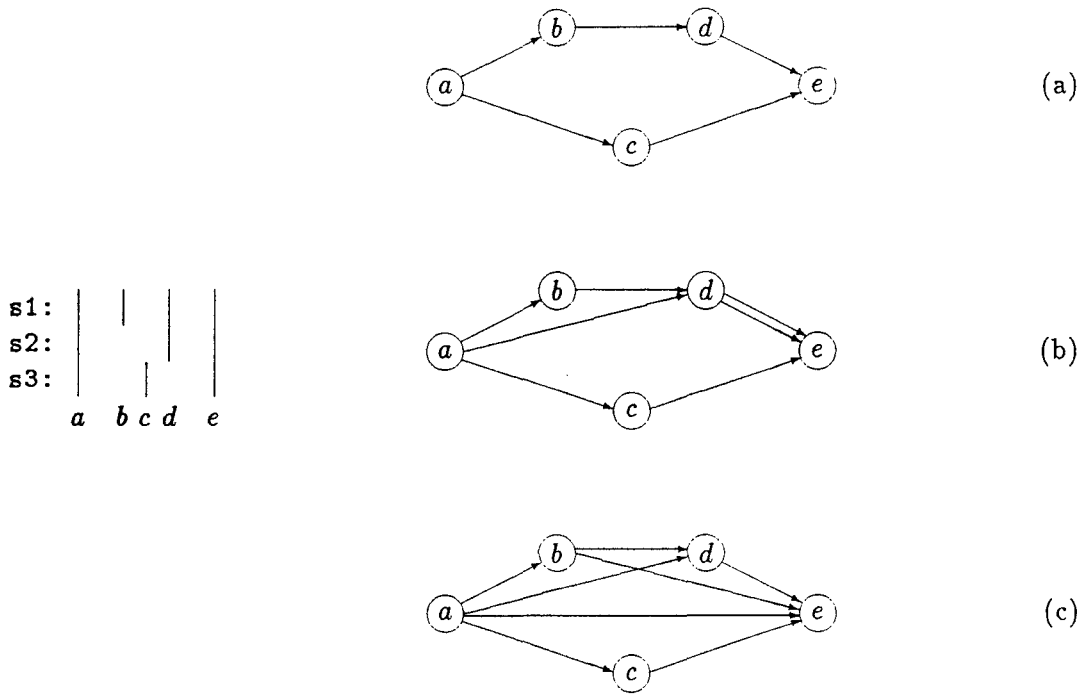


Figure 1: Possible graphs as internal representation. At the left a grid is shown. At the right side three graphs are shown, all representing the grid. At the top (a) the minimal graph is shown, in the center the direct graph (b), and at the bottom (c) the maximal graph is shown.

target is in the left or right context of another arbitrary sync mark *source*. The principle idea is: start searching at *source* in the indicated *direction* via all possible paths. Terminate with success (**true**) if the *target* is encountered, and with failure (**false**) if all paths eventually reach the *boundary* sync mark without having encountered the *target*. However, since each vertex (sync mark) can have maximal M edges to another vertex, the number of paths can be M^{N-1}. However, these paths can never all be distinct, since there are only N vertices. In other words: some of the paths will go via the same vertex. This is where memo-ization comes in. Once a vertex has been dealt with, i.e., once all paths for a sync mark are investigated, you know that via that vertex you can never reach the *target*. Thus, we mark the vertex, and whenever we hit a marked vertex, we stop the search via that path with failure. This reduces the worst case complexity to $M \cdot N$: each vertex (of which there are N) is invoked at most M times (the maximum number of streams a sync mark can be present in).

Closer inspection of the algorithm learns that it is of order E, where E is the number of edges between the vertices. As each vertex is marked as soon as it gets dealt with, each outgoing edge will only be investigated once. The sum of all outgoing edges of all vertices is exactly E, the total number of edges in the graph. In the worst case this is be $M \cdot N$, which represents that case that all sync marks are present in all streams. In practical grids, however, $E \ll M \cdot N$ (see section 7).

Below, an algorithm using an array implementation of the graph is given. The basic data structure is a two-dimensional array, where the first index gives the *value* of the sync mark

and the second the stream. For each element of this array the basic relations are given: (a) whether this sync mark is present in this stream, and if so, (b) what is its left neighbour and (c) what is its right neighbour. This structure is called a *grid_impl* (in a real implementation also linguistic data will be stored somewhere, of course, but this is not relevant, here):

```
basic_relations = record
    present : boolean;
    left, right : sync_mark; end;
grid_impl = array [1..max_sync, 1..max_stream] of basic_relations;

var     {global for Speech Maker}
    grid : grid_impl;
```

The function *In_context* is the entry point from outside, which answers the question whether *target* can be reached from *source* in a certain *direction*. For the duration of the quest the values of *target*, *direction* and the *boundary* will not change. In this case (as opposed to the compile-time case), the *boundary* is either the leftmost or the rightmost sync mark that are always present in each stream. It seems useless loss of efficiency to transfer these values explicitly to *Search* (a function that is invoked recursively) each time by means of parameters. Therefore, they are stored in parameters that are global for *In_context* and *Search*. After fixation of these parameters the function *Search* is invoked, which directly gives the result. A side effect of *Search*, however, is that some of the vertices are marked, so this side effect has to be reset (for which we assume here that a procedure *Unmark* is available):

```
var     {global for In_context and Search}
    target : sync_mark;
    dir : (←, →);
    boundary : sync_mark;

function In_context(src : sync_mark;
                    trg : sync_mark;
                    drc : direction) : boolean;
begin
    target := trg;
    dir := drc;
    if (dir = ←)
        then boundary := leftmost
        else boundary := rightmost fi;
    in_context := Search(src);
    Unmark;
end;      {of In_context}
```

Function *Search* does the work. It starts by checking if the current sync mark is the *target* that is searched for. If so, it terminates with success. If not, it checks whether the current sync mark has already been dealt with, or if it is the terminating boundary. If so, it terminates with failure. If these filters are passed, the real work begins. First the sync mark (the vertex) receives a mark, so as to indicate that when it will be encountered via an alternative path the work should not be doubled. Then, all outgoing edges are investigated by means of a recursive call to *Search*. Of course, one may terminate the search as soon as one of the outgoing edges hits the *target*:

```
function Search(sm : sync) : boolean;
var
  target_found : boolean;
  stream : integer;
begin
  if (sm = target)
  then target_found := true    {terminate}
  else {Treat Non Target}
    if Marked(sm) or (sm = boundary)    {use memo-ization}
    then target_found := false
    else    {Treat New Sm}
      Mark(sm);    {memo-ize}
      target_found := false;
      stream := 1;
      while (stream <= nr_of_streams) and not target_found) do {treat all neighbors}
        if (grid[sm, stream].present) then
          if (dir = ←)
            then target_found := Search(grid[sm, stream].left)
            else target_found := Search(grid[sm, stream].right)
          fi
        fi
        stream := stream + 1
      od
    fi
  fi;
  search := target_found;
end;    {of Search}
```

6.3 Algorithm Det_runtime_checks

A similar approach can be taken with respect to the problem *which* are the pairs of sync mark for which it should be established during run-time that they do not conflict with existing synchronization. In establishing that a relation $a < b$ in the action part is guaranteed (or not) by the relations in the pattern part, we once again have the problem to determine whether one sync mark is in the context of the other. Thus, also for this problem we will use graphs to represent the sync mark relations. There are some subtle differences with a grid (for instance, the graph representations of the pattern and action part are generally not bounded by a leftmost and a rightmost sync mark), but these can be accounted for without too much effort.

The main strategy of *Det_runtime_checks*, therefore, is to build graphs of both the pattern and the action part, and use these to determine the which run-time checks are needed. We will not go into the process of building the graphs, but we assume they can be made, and that for the action part there is a pointer for each stream pointing to the first used sync mark in that AL. In this implementation, the graphs are appended with an extra sync mark (we use the value zero) where *graph*[0, *stream*].*right* points at the first sync mark of the AL.

The idea is then as follows. Treat the streams in the action part one by one. For each stream, consider each pair of sync marks encountered in the AL, for which both sync marks are present in the pattern graph (*Present*(*sm*)). Check whether, in the pattern graph, the

second sync mark is in the right context of the *first*. If so, no action is needed, as the pattern part already guarantees that the relation to be added will not conflict with existing relations at the time the rule is executed. If this cannot be guaranteed by the pattern part, check whether *second* is in the left context of *first*. If so, the order of sync marks in pattern and action part is reversed: an error message must be generated. If not, this is the case that we must determine the context relations during run-time: the pattern part says noting about the relations of *first* and *second*, and if the rule matches, *second* may not be in the left context of *first*:

```
procedure Det_runtime_checks;
var
  stream : integer;
  first, second : sync_mark;
  out : boolean;
begin
  for stream := 1 to nr_of_streams do
    if (act_graph[0, stream].right ≠ 0) then
      first := act_graph[0, stream].right;
      while not Present(first) do first := act_graph[first, stream].right;
      if (first ≠ 0) then    {first is present}
        out := false;
        while not out do
          second := act_graph[first, stream].right;
          while not Present(second) do second := act_graph[second, stream].right;
          if (second = 0)
          then out := true    {no more pairs; exit}
          else    {second is present}
            if not In_context(first, second, right) then
              if In_context(first, second, left)
              then Error_reversed(first, second)
              else Runtime_verify(first, second, left);
              fi
            fi
          fi
          first := second;
        od
      fi
    fi
  od
end;    {of Det_runtime_checks}
```

7 Some practical figures

To get a final grip on the problem, it is important to know some practical figures: given M streams and N sync marks, how many edges E are there on average in actual grids, and given P rules, how often does it occur that one specifies rules that call for run-time checks?

The first questions concerns the run-time aspects. In our 1992-implementation of Spraakmaker 16 streams are used, a number that has not changed for quite some time now, and which is not expected to change drastically in the future, so M is typically of this order.

The number of sync marks depends quite linearly on the length of the sentence that is being analysed (which currently is the unit of processing). When all analysis modules of Spraakmaker have been invoked, the number of sync marks in the grid is on the average 2.35 times the number of characters in the sentence, with a remarkable low spread in this figure: 0.12 (measured from some twenty sentences with a variation in length from 27 to 179 characters). Further, also the average number of edges per sync mark is remarkably stable: 2.63 ± 0.05. So typically N will be ± 2.35 times the number of characters in a sentence, and E, the number of edges in the grid, ± 6.25 times the number of characters in a sentence.

As we saw, the number of actual edges, E, is indicative for the amount of work that the *In_context* routine will have to do. One could say, therefore, that the time-complexity of *In_context* routine is linear in the length of the sentence (we cannot expect to get much better without domain-specific constraints, such as: do not search further than two or three words). Since the length of sentences in characters does not really get large (more than thousand characters would be an extremely long sentence) there is nothing to worry about with the current algorithm. This would be different, of course, if the time complexity of the context routine would be quadratic with the sentence length, and this is probably the case for all algorithms that take a sync mark in a particular stream as single object rather than taking a sync mark over all streams as single object in a graph.

The second question concerns the compile-time aspect, how often is a run-time check needed? At present we do not have very much real-life applications to test upon, only three serious linguistic modules have been developed. One module determines the pronunciation of acronyms, another the pronunciation of numbers and the third the realization of intonation contours. These consist of 70, 59 and 37 rules respectively, and only for two rules run-time checking is needed. This is somewhat disappointing perhaps, but had we known it in advance we would still not have been discharged with finding a solution to the consistency problem. We cannot allow the grid to become inconsistent, and we cannot count on it that a rule writer will never write a more complicated rule that does call for run-time checking. For, although the simplest rule with this property that can be thought (e.g. rule (35)) of is not altogether an everyday rule, it is not of a type that would never be useful.

Thus, although consistency is a serious theoretical problem, it is not a very big practical problem. However, apart from the reasons for maintaining consistency, it is nevertheless advisable to implement the grid as a graph. After all, the sync marks are the anchors of the grid, one cannot shift from one stream to another without using them, and it seems ineffective to split them as objects over the streams. Any process with any linguistic relevance will need to know the relation between the sync marks. For instance, even establishing the equality of sync marks, i.e., establishing the presence of a sync mark in another stream (and this is a construct that is used very often in SMF rules: in 166 rules about 1000 times) becomes a process of order N in a data structure where the stream is the main entry, as opposed to the graph representation where it is order 1.

8 Conclusion

The conclusion of this paper is the following: use a graph as internal representation for multi-level, synchronized data structures such as a grid. This saves order N in the time complexity of the relevant algorithms, where N the length of the input text that is to be converted. Graphs are absolutely necessary if consistency checks must be done regularly, and they are very advantageous for other (linguistic) processes that are typically done in text-to-speech systems.

9 References

Boves, L. (1991). Considerations in the design of a multi-lingual text-to-speech system. *Journal of Phonetics* **19**, 25–36.

Hertz, S.R., Kadin, J. & Karplus, K. (1985). The Delta rule development system for speech synthesis from text. *Proceedings of the IEEE* **73** (11), 1589–1601.

Lazzaretto, S. & Nebbia, L. (1987). SCYLA: Speech Compiler for your Language. *Proceedings of the European Conference on Speech Technology 87* **1**, 381–384.

van Leeuwen, H.C. & te Lindert, E. (forthcoming). Speech Maker: a flexible and general framework for text-to-speech synthesis, and its application to Dutch. *Computer, Speech and Language*, 19 pp.

van Leeuwen, H.C. (submitted). Speech Maker Formalism: a rule formalism operating on a multi-level, synchronized data structure. Submitted to *Computer, Speech and Language*, 18 pp.

Even, S. (1979). *Graph Algorithms.* Computer Science Press, Inc. Rockville, Maryland.