THE ARCHITECTURE OF THE FESTIVAL SPEECH SYNTHESIS SYSTEM

Paul Taylor *Alan W Black* *Richard Caley*

Centre for Speech Technology Research,
University of Edinburgh, 80,South Bridge, Edinburgh EH1 1HN, UK
{pault,awb,rjc}@cstr.ed.ac.uk

ABSTRACT

We describe a new formalism for storing linguistic data in a text to speech system. Linguistic entities such as words and phones are stored as *feature structures* in a general object called an linguistic *item*. Items are configurable at run time and via the feature structure can contain arbitrary information. Linguistic *relations* are used to store the relationship between items of the same linguistic type. Relations can take any graph structure but are commonly trees or lists. Utterance structures contain all the items and relations contained in a single utterance. We first describe the design goals when building a synthesis architecture, and then describe some problems with previous architectures. We then discuss our new formalism in general along with the implementation details and consequences of our approach.

1. INTRODUCTION

Speech synthesis systems require ways of storing the various types of linguistic information produced in the process of converting the input format (e.g. text) into speech. In this paper we present a new formalism for representing arbitrary linguistic data and show how this helps in building a speech synthesis system.

The are a number of design considerations which builders of synthesis architectures must take into account. We have listed these as follows:

- Simple linguistic objects such as words, phones, syllables and phrases need to be represented.
- A co-indexing mechanism is needed whereby given a word, one can find the phones that comprise this word.
- Changes should be localized. For example, a change in the syllabification algorithm should only require changing those parts of the program directly involved with syllabification - other modules should not be affected.
- There should be no redundancy or duplication of information in the system. For instance, it is common to want to know the start and end times of linguistic entities. The end time of a word will be the same as the end time of the last phone in that word. If this information is stored separately for the phone and word, problems will occur if one value is changed as the other will then become out of date.
- By their very nature, multi-lingual systems must support a wide variety of linguistic theories. Hence the architecture should not be tied to any particular linguistic theory or formalism.
- The architecture should be fast and efficient as the synthesizer is intended for real-time use.

Most importantly, the real purpose of the architecture is to allow speech synthesis algorithms to be written as easily as possible. It is therefore important that the architecture should be unobtrusive and provide the sorts of structures and information that synthesis algorithms need. All programs must deal with infrastructure issues such as data storage, file i/o, memory allocation etc. If left unchecked, algorithms can easily become bogged down with this sort of code, which becomes intertwined with the actual algorithm itself. A good architecture will abstract the infrastructure to such an extent that these aspects are hidden, so that synthesis algorithms can be easily written and read without other issues getting in the way. The interface should be easy to use, and make the writing of synthesis algorithms easier and quicker than by purely ad-hoc methods.

2. BACKGROUND

In this section we review some previous types of architecture.

2.1. String processing

Many early synthesis systems used what has been referred to as a *string re-writing* mechanism as their central data structure. In this formalism, the linguistic representation of an utterance is stored as a string. Initially, the string contains text, which is then re-written or embellished with extra symbols as processing takes place. Systems such as MITalk [1] and the CSTR Alvey synthesizer [5] used this method.

There are many short comings in this formalism which have been previously recognized ([6], [3], [4], [7]). The main problem with the string formalism is that it soon becomes unwieldy for anything apart from the most trivial of tasks. Often the string becomes very complex with words, phrase symbols, stress symbols, phones etc all mixed in together. There are two main ways in which modules process such strings. Modules can work on this string directly, but the interpretation of the symbols often gets in the way of the algorithm itself. Alternatively, a module can parse the string into an internal format and let the algorithms use that. Although this may simplify the writing of the algorithms themselves, this is a very unwieldy approach as it means the string has to be parsed every time a module is called. Moreover, this often leads to each module having an individual internal data structure, which is unattractive from a programming point of view as new structures and techniques have to be learnt to understand the workings of any new module.

To lessen these problems, information is often deleted from the string so as to keep only what is perceived as essential information. For instance, after the grapheme to phoneme conversion, the orthographic form of the word may be deleted from the string. This can have unfortunate consequences in that information which might potentially be of use to a module may have been deleted by an earlier module.

2.2. Multi-level data structures

In recognition of these shortcomings, many current systems have abandoned string based processing and now use *multi-level data structures* (MLDS). The most famous of these systems, known as Delta, was developed by Hertz [6], but many other systems, such as Chatr [3], the Bell labs system [7], Polyglot [4], and early versions of Festival [2] use similar formalisms. In the multi-layered formalism, different types of linguistic information are held in separate *streams* which are linear lists or arrays of linguistic items. For example we may have a word stream, a phone stream and a syllable stream. Some systems have a fixed set of these while others an allow arbitrary number.

Often algorithms need to know what phones are related to a given word, and hence the streams must be co-indexed. There are two main types of co-indexing. In Delta, streams are aligning by the edges of items. To find the phones in a word, one goes to the beginning of the word, traces the edge “down” to the phone stream, and they progresses along the phone stream until the edge relating to the end of the word is found. An alternative strategy is to align by the “centres” of items. In this case, a word contains a set of links to the phones that are related to it, and the phones in a word can be found by following these links.

While multi-level data structures are far preferable to string structures they still have serious drawbacks. The main drawback stems from the fact that this forces all information to be represented by linear structures: other types of structure, specifically trees, are very hard to represent. Partly because of this, the number of streams in a system can become considerable which leads to difficulties in co-indexing the items in streams. With the delta co-indexing, it is often the case that for a given item in one stream, there is no corresponding item in other streams. For example, a pause is represented by a item in the phone stream, but there is no equivalent item in the syllable or word streams. Thus a “hole” must be created in these streams to ensure the co-indexing works. With a large number of streams, the number of holes can become considerable which makes processing awkward. In the centre-linking paradigm, the hole problem is absent, but because each item must be explicitly linked to items in other streams, the number of links can become very large. Furthermore, if a new stream is added, one would have to link every existing item to the items in the new stream to ensure full connectivity. As this isn’t possible in practice, the streams are often left partially connected. This can cause confusion in writing module, as one may be unsure of what streams are linked to what.

Another more subtle problem occurs in multi-level structures due to a lack of clarity as to what an item really represents. Often (not always) each item has a single value, typically a name. While it is obvious that a suitable phone name could be something like /h/ or /e/ and a word “hello”, what should the name of a syllable be? Commonly, syllables are regarded as *organisational* units, which serve to group together phones. As such, they don’t have a distinct name. One could group together the names of the phones and make that the syllable name (e.g. “/h e l/” for the first syllable of “hello”), but this is redundant, somewhat artificial and likely to cause errors if the phone representation is changed.

3. A FORMALISM BASED ON INTERSECTING RELATIONS

The most significant difference between the Festival architecture and those using MLDS is that Festival does not constrain linguistic items to be in linear lists: any graph structure is allowed. Additionally, items can be contained in more than one structure, which leads to more efficient representations. We have also generalised the content of an item such that information of arbitrary complexity can be easily stored.

3.1. Relations

We used the term *relation* to represent our generalisation of the stream concept. A relation is a data structure which is used to organize linguistic items into linguistic structures such as trees and lists. A relation is a set of *named links* connecting a set of *nodes*. Lists (streams in the MLDS) are linear relations where each node has a previous and next link. Nodes in trees have up and down links also. Nodes are purely positional units, and contain no information apart from their links. In addition to having links to other nodes, each node has a single link to an *item*, which contains the linguistic information.

Figure 1 shows an example utterance with a syntax relation and a word relation. The word relation contains nodes which have **next** and **previous** connections, whereas the syntax relation, has **up** and **down** connections in addition to next and previous. Each node in the syntax tree is linked to a item, each of which has a feature, **CAT**, giving its syntactic category. Terminal nodes in a syntax tree are words, and so an additional feature **name** is used here. The nodes in the word relation, which is a linear list, are

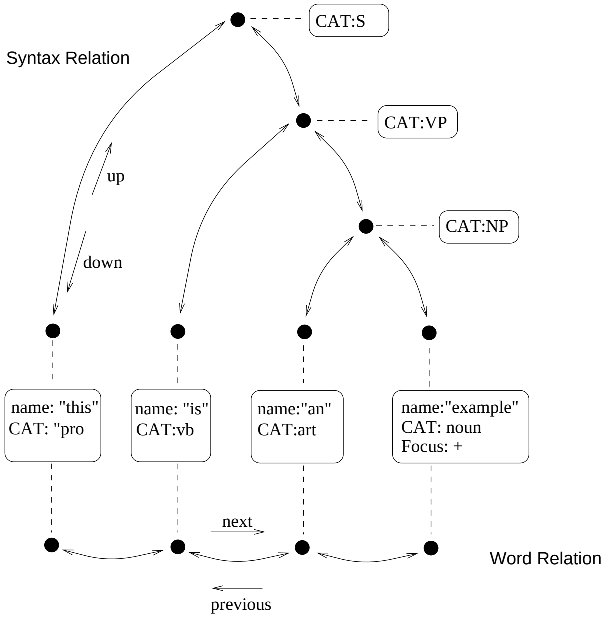

Figure 1: An example representation of an utterance structure. This example shows the word relation and the syntax relation. The syntax relation (shown on top) is a tree with links connecting the nodes, shown as black circles. The word relation (shown on the bottom) is a list. The items contain the actual linguistic information and are shown in the rounded boxes. The dotted lines show the connections between the nodes and items.

also linked to the items that are linked to the terminal nodes in the syntax tree.

In this way, node structures of arbitrary complexity can be constructed, and they can be intertwined in a natural way by having links from different nodes to the same item.

3.2. Items and Features

Items consist of a bundle of features and a set of named links to nodes in relations. Items can be linked into any number of relations - in the above example words were linked into 2 relations, but in principle any number of relations is possible.

The information in items themselves are represented by *features* which are stored as a list of key-value pairs. Feature values are commonly numbers or strings, but may also take complex objects as values if necessary. A item can have arbitrarily many features, including zero: syllable items often have no features at all, for example.

As well as simple values, features can also take *functions* as values. Function features are a powerful facility which can help to greatly reduce the amount of redundant information in a utterance structure.

Consider the case of a simple phone segmentation of an utterance, which comprises a contiguous list of named segments each with timing information. If the phones are properly contiguous, only one timing value for each phone is needed to fully represent all the timing information of the segmentation. If the end point is used, the start time of an item will be equal to the end point of the previous item and the duration will be equal to the start subtracted from the end. However, it is often useful to have access to the start times and duration of the phone also. Without the use of function features, there are two choices. Either the end information alone can be stored and calculations can be done on the fly to compute start times and durations, or else the start and durations can be written in as additional features to the item. The first solution is unattractive as this can make algorithm writing unwieldy and overly complicated. While the second solution will may make algorithm writing easier, it involves effectively copying information, which can lead to out of date information being present.

Function features provide a neat solution to this problem. In the current example, a *start* function is written (in Festival this can be written in C++ or scheme) which looks at the previous item and returns its end. This function is then assigned as the value of a key named “start” in the items in question. When one accesses the start feature a time value is returned as if it were actually stored. A duration function returns the value of the start feature subtracted from the end feature. The duration function simply evaluates the feature named “start” - it doesn’t need to know if it is a simple feature or a function feature.

In our current usage of the Festival architecture, we only keep time positions in items linked to the segment relation: all other times are calculated by functions. Different start functions can be written for different purposes. For example, the end function assigned to the non-terminal nodes in the syntax tree descends from the current node until a terminal node is encountered and that node’s item’s end value is returned. Of course the terminal node’s item, being a word, has its end feature as a function also - a common operation is for the word’s end function to return the time of the last syllable it is related to, which returns the time of the last phone etc. In each case a separate end function is used, but all are assigned to the key “end” in the item. This means that algorithms can evaluate the end feature on any item, and be sure of a legitimate value being returned without having to worry about the details of calculation.

Although the function feature implementation is very efficient in Festival, it can still sometimes be expense to constantly evaluate these functions in the middle of a time critical loop. A global evaluation facility is therefore provided which can evaluate all the feature functions in items in a relation and re-write the features with their evaluation. After the loop, these can be discarded and the feature functions used again, thus ensuring little chance of out of date information being present.

Function features can be used for a variety of purposes other than timing. Another common usage in Festival is to use functions for intonation. Typically, intonation accents are associated with syllables. Thus one can ask whether a syllable is accented or not. However it is also useful to know if words are accented also. In this case we define a function feature on word items which looks at that word's main stressed syllable and returns true if that syllable is accented.

3.3. Utterances

Utterance structures are collections of relations. The best way to visualize an utterance is to think of it containing a unordered set of items, each of which is made up from a set of features. Relations are then structures comprising of nodes, each of which indexes into an item. An utterance structure simply collects these together to form a single object.

4. IMPLEMENTATION

While we will not go into the actual low level details of our implementation here, it is useful to raise some points about how the nature of the programming language used affects the implementation.

Festival is implemented in two languages, C++ and scheme (a variant of lisp). While in principle it would be attractive to implement the system in a single language, practical reasons concerning the nature of programming languages necessitate the approach we have taken here.

In addition to being a research platform, Festival also operates as a run-time system and hence speed is vital. Because of this, it is necessary to have substantial amounts of the code written in a compiled low-level language such as C or C++. For particular types of operations, such as the array processing often used in signal processing, a language such as C/C++ is much faster than higher-level alternatives. However, it is too restrictive to use a system that is 100% compiled, as this prevents essential run-time configuration as the following examples demonstrate.

In developing an algorithm, it is often useful to try alternatives. With a completely compiled system, trying alternatives would involve changing the code and then re-compiling and running the program. While this may be an acceptable effort for two or three algorithm variations, it soon becomes impractical for larger numbers, especially when a large set of alternatives need to be tried as part of an experiment. With an interpreter, the changes can be made at run time, and it is often a simple matter to write a scheme script which can iterate through all the alternatives and produce a table of results.

Festival is used for synthesizing many languages and it would be impossible to re-configure the compiled code for each. In practical usage, Festival must therefore be flexible enough so that any algorithm in any language can be implemented without the re-compilation of existing code. Due to this requirement, structures such as features, items and relations have been implemented so that any number of each with any name can be used. Specifically the use of C structures, where the fields in the structure would correspond to entities such as “stress”, “end” or “part of speech” have been avoided as the addition of any new feature would require a re-compilation. Instead, features are represented by extensible key-value lists. Feature names are stored as strings and efficient functions are used to return the value of a feature given the string name. The relations in an utterance are also stored as an extensible list, with strings being used to provide access to a given relation. Hence there is nothing in the C++ architecture which dictates what relations, items or features should be called. All items, relations and features are of exactly the same type in C++ regardless of what linguistic information they carry. It is only at run time that their linguistic function is designated.

We have found from experience that when designing a complex architecture such as this, it is important to take into account the expectations of a programmer with regard to a language their are experienced with. In a previous architecture we developed ([3]) we achieved the some of the generality and run-time flexibility described above. However the C language constructs we used we often obscure, stretching the language to its very limits. Because of this, even experience C programmers found the system very difficult to program with simply because much of the code didn't look like recognizable C to them. In Festival we have paid much more attention to this problem as the current C++ interface seems fairly natural and unobtrusive.

File I/O functions have been written so that utterance structures be saved and loaded to and from disk at any time in the synthesis procedure. In fact, this facility has proved so useful that we now store many of our speech databases (which contained phone, word and intonation information) in this format.

4.1. Core System and Modules

In Festival we make a firm distinction between the core system and the modules which actually perform speech synthesis tasks. The core system, which includes the architecture is written completely in C++ and doesn't change. Modules on the other hand can be written in C++ or scheme and can be added or taken away form the system with minimal disruption (adding or taking away a C++ module requires a re-linking of course, but not a major re-compilation). The decision of which language to write a module in is largely a matter of choice. Because of the interpreter aspect, it is usually easiest to develop modules in scheme and maybe for efficiency re-write them in C++ after they are stable. Some types of programming (e.g. arrays in C++, recursion in scheme) are more natural in one language than another. Finally, depending on personal experience, some programmers simply prefer one to the other.

As far as possible we have provided identical interfaces to the architecture in both languages which makes switching between each relatively easy.

4.2. Other Languages

Nothing in the design of the system architecture is specific to the languages we have used in our implementation. C++ was used to facilitate better data abstraction in the classes/structures of the architecture, but C could have also been used. Scheme was used firstly because it is a small and clean scripting language and secondly because the lisp s-expression (bracketed string) data structure allows a generic way to store complex data structures.

Other languages would be also be suitable, and we have made some progress towards providing alternative scripting languages in the style of perl/unix shell and Java etc. Ideally the compiled part and interpreted part would be in the same language. Until recently no available language has this degree of flexibility, but perhaps a future generation of Festival could accomplish this by using Java.

5. CONCLUSION

In summary, we feel that the following features of the current Festival architecture go a substantial way to reaching the design goals mentioned in the introduction.

- **Complex relations:** allows trees, lists and other linguistic structures to be represented in the same formalism. Intersecting relations allow an item to be in more than one relation which saves on redundancy.
- **Feature structures in items:** allows arbitrary amounts and types of information to be used to describe linguistic objects. Due to run-time configurability, no re-compilation is needed for new types of information to be stored.
- **Function features:** allows useful information to be calculated on the fly, reducing redundancy.
- **Speed:** efficient implementation ensures that the expressive power of the architecture does not impose a prohibitive speed cost.
- **Natural interface:** quick to learn and hence programmers are not intimidated when learning to write modules for Festival.

That said, we acknowledge that architecture design is never a solved problem. As the standard of the architecture increases, so do the demands and expectations of the programmers using it, and hence new design features are always required. However, we feel that the features listed above are a substantial improvement on previous architectures and go along way to facilitating quick and easy speech synthesis programming and algorithm implementation.

Acknowledgements

We gratefully acknowledge the support of the UK Engineering and Physical Science Research Council (grants GR/L53250 and GR/K54229) and Sun Microsystems.

6. REFERENCES

1. J. Allen, S. Hunnicut, and D. Klatt. *From Text to Speech: the MITalk System*. Cambridge University Press, 1987.
2. A. W. Black and P. Taylor. The Festival Speech Synthesis System: system documentation. Technical Report HCRC/TR-83, Human Communciation Research Centre, University of Edinburgh, Scotland, UK, 1997. Avaliable at http://www.cstr.ed.ac.uk/projects/festival.html.
3. Alan W. Black and Paul A. Taylor. CHATR: A generic speech synthesis system. In *COLING '94, Kyoto, Japan*, 1994.
4. Louis Boves. Considerations in the design of a multi-lingual text-to-speech system. *Journal of Phonetics*, 19(1):309–327, 1991.
5. W. N. Campbell, S. D. Isard, A. I. C. Monaghan, and J. Verhoven. Duration, pitch and diphones in the CSTR TTS system. In *International Conference on Speech and Language Processing '90, Kobe, Japan*, 1990.
6. Susan R. Hertz. The delta programming language: an integrated approach to non-linear phonology, phonetics and speech synthesis. In John Kingston and Mary E. Beckman, editors, *Papers in Laboratory Phonology 1*. Cambridge University Press, 1990.
7. Richard Sproat and Joseph Olive. A modular architecture for multilingual text-to-speech. In *Second ESCA/IEEE Workshop on Speech Synthesis, New York*, 1994.