ARTIST
An AI-based tool for the design of intelligent assistants for sound synthesis

Eduardo Reck Miranda,
AI/Music Group,
Faculty of Music and Dept. of Artificial Intelligence,
University of Edinburgh,
12, Nicolson Square,
Edinburgh, EH8 9DF,
Scotland, UK.
E-mail: miranda@music.ed.ac.uk

Abstract

In this paper we introduce the fundamentals of ARTIST (an acronym for Artificial Intelligence-aided Synthesis Tool). ARTIST is a tool for the design of intelligent assistants for sound synthesis that allow composition of sounds thought of in terms of qualitative descriptions (e.g. words in English) and intuitive operations rather than low level computer programming. Our research work is looking for (a) plausible strategies to map the composer's intuitive notion of sounds to the parametric control of electronic sound synthesis and (b) how to provide artificial intelligence (AI) to a synthesiser. In this paper we introduce how we attempted to approach the problem by means of a compilation of a few well known expert systems design techniques used in AI research. ARTIST is a prototype system which embodies the results of our investigation so far.

Keywords: AI-based synthesiser, knowledge-based systems, machine learning

Introduction

In the final quarter of the 20th century the invention of sound recording followed by sound processing and then sound synthesis have changed our view of what constitutes music. These recent developments have vastly expanded our knowledge of the nature of sounds. Nowadays, computer technology offers composers the most detailed control of the internal parameters of sound synthesis and signal processing.

Wanting the effective use of the new technology, composers become more ambitious, but the complexity also increases. The scale and nature of the compositional task changes, technically and aesthetically. Theoretically the computer can be programmed to generate any sound one can imagine. But, on the other hand, this can get composers into trouble. Quoting Barrière (1989, pp. 116),*"it is too easy to fail to take various consequences into account, to get technology side-tracked by a tool whose fascinating complexity can become a disastrous mirage".*

Even if the composer knows the role played by each single parameter for synthesising a sound, the traditional way of working with computer synthesis, tediously entering exact data at the terminal, is not particularly stimulating. We are convinced that higher processes of inventive creativity and musical abstraction are often prejudiced in such a situation. In this case we think that the computer is being used as a kind of word processor combined with player piano, and not as a creative tool. We have come to believe that this can be improved by means of an appropriate coupling between human imagination and artificial intelligence (AI).

In this paper, we introduce the fundamentals of ARTIST (an acronym for Artificial Intelligence-aided Synthesis Tool). ARTIST is a tool for the design of intelligent assistants for sound synthesis that allow composition of sounds thought of in terms of intuitive qualitative descriptions (e.g. words in English) rather than low level computer programming (Miranda et al., 1993a; 1993b; Miranda, 1994a; 1994b; 1994c). By an intelligent assistant we mean a system which works co-operatively with the user by providing useful levels of automated reasoning in order to support laborious and tedious tasks (such as working out an appropriate stream of synthesis parameters for each desired single sound), and to aid the user to explore possible alternatives when designing a sound. The desirable capabilities of such a system can be summarised as follows:

(a) The ability to operate the system by means of an intuitive vocabulary instead of sound synthesis numerical values,

(b) The ability to customise the system according to the user's particular needs, ranging from defining which synthesis technique(s) will be used to defining the vocabulary for communication,

(c) The encouragement of the use of the computer as a collaborator in the process of exploring ideas,

(d) The ability to aid the user in concept formation, such as the generalisation of common characteristics among sounds and their classification according to prominent attributes, and

(e) The ability of creating contexts which augments the chances of something unexpected and interesting happening, such as an unimagined sound out of an ill-defined requirement.

Apart from graphic workstations (such as the UPIC system (Xenakis, 1992; Marino et al., 1993; Lohner, 1986)) and medium level programming languages (see (Pennycook, 1985) for a survey), little research has been done towards a system for sound synthesis that responds to higher levels of sound description. An early attempt at the definition of a grammar for sound synthesis was made by Holtzman (1978) at Edinburgh University. Also, Slawson (1985) has proposed - not implemented on a machine though - a kind of vocabulary for sound composition based on his theory of sound colour which, we believe, he made up from Helmholtz's theory of vowel qualities of tones (Helmholtz, 1885). Lerdahl (1987) too has done some sketches towards a hierarchical perceptually-orientated description of timbres. Apart from these, it is worth mentioning that there have been a few attempts towards signal processing systems that understand natural language. The most successful ones are interfaces developed to function as a front end for systems which perform tasks to do with audio recording studio techniques such as, mixing, equalisation, and multitracking (e.g. CIMS (Schmidt, 1987) and Elthar (Garton, 1989)). More recently, Ethington and Punch (1994) proposed a software called SeaWave. SeaWave is an additive synthesiser (Dodge and Jerse, 1985) in which sounds can be produced by means of a vocabulary of descriptive terms. Although of a limited scope, SeaWave proffers an excellent insight and it seems to work well. Vertegal and Bonis (1994) also have been working towards a cognitive-orientated interface for synthesisers.

We begin the paper by introducing the problem from a musician's point of view. Then we introduce the signal processing of the synthesiser which will be used as an example study. After this, we indicate some methods for describing sounds by means of their attributes and suggest a technique for mapping those attributes onto the parameters of a synthesiser. Then we study how this technique works and present some examples. Here, we also study the utility and the functioning of machine learning in this kind of system. Finally, we propose a system architecture which embodies all the concepts discussed so far and introduce its functioning through examples. We end the paper with some final remarks and ongoing work.

An example study synthesiser

Assume that we wish a synthesiser which is able to produce human voice-like sounds. It is worth mentioning that to produce a perfect simulation of the human vocal tract is out of the scope of this paper. Thus, rather than making a description of the fundamental aspects of the phenomenon by means of a set of equations (e.g. (Woodhouse, 1992; Keefe, 1992)), we opted for observing it by means of a more traditional formant modelling technique which uses subtractive synthesis (Flanagan, 1984; Klatt, 1990; Sundberg, 1991; Miranda, 1992; 1993). We come to believe that this level of description (see also (Eckel, 1993) for a brief discussion about this business) suffices at this moment. The signal processing diagram of our example study synthesiser is shown in Figure 1.

Each block of the diagram (except the *Envelope*) is composed of several signal processing units (SPU). A composition of SPU's form sub-blocks within a block. Sub-blocks in turn may constitute sub-sub-blocks, and so forth. The *Voicing Source,* for example, has two sub-blocks: one, the *Vibrato sub-block,* contains an oscillator unit, and the other, the *Pulse Generator sub-block,* contains a pulse generator unit (Figure 2).

Each SPU needs parameter values for functioning. We say that, in order to produce a certain

Figure 1: The example study architecture.

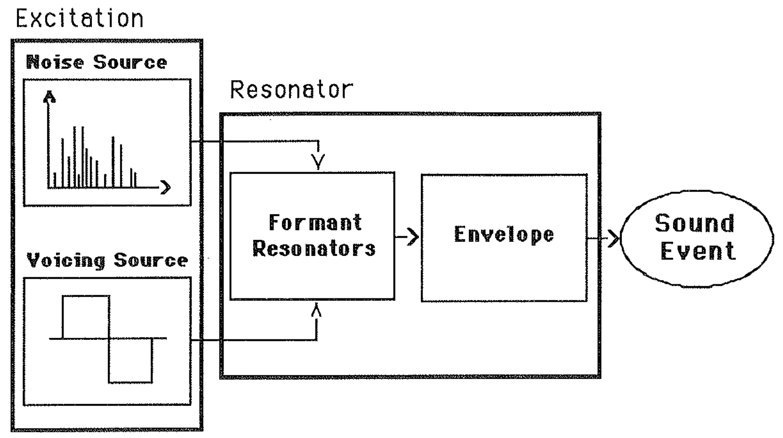
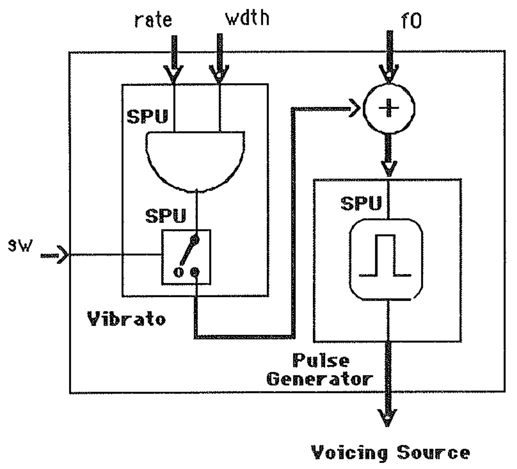


Figure 2: The voicing source block.



Describing sound by means of their attributes

There have been several studies defining a framework to systematically describe sounds by means of their attributes ((Schaeffer, 1966; von Bismark, 1971; 1974a; 1974b; Cogan, 1984; Giomi & Ligabue, 1992; Carpenter, 1990; Terhardt, 1974) to cite but a few). They are derived mainly from work in the fields of both psychoacoustics and musical analysis. We classify these studies in two approaches: on the one hand, the *device-orientated* approach and, on the other hand, the *perceptually-orientated* approach. As it is not our aim to survey all these, we have selected one example of a

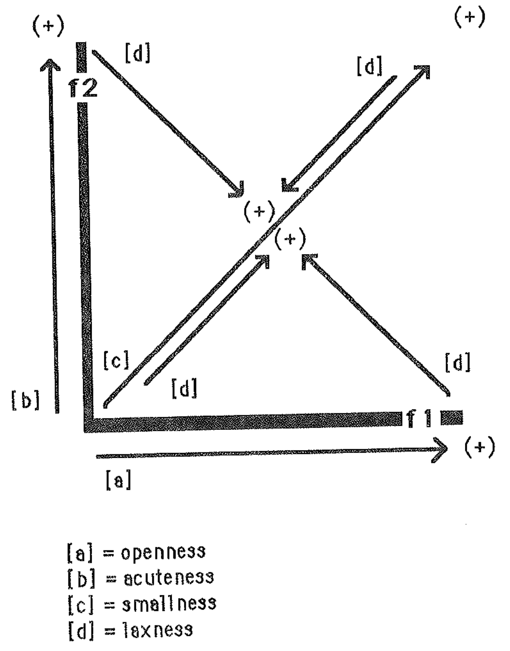
The source-filter model: a device-orientated approach

The source-filter model formulates that the characteristic of a sound is determined by its spectrum envelope's pattern. This pattern is composed of multiple hills called formants. Each formant has a centre frequency peak and a bandwidth. According to this model, the lowest two formants are the most significant determinants of sound quality.

The pattern of the spectrum envelope of formant frequencies is thought of as the result of a complex filter through which a source sound passes.

We can define here a two-dimensional space whose axes are the first (**f(1)**) and the second (**f(2)**) centre formant frequencies respectively. Then, four perceptual attributes, namely *openness*, *acuteness*, *smallness*, and *laxness* (after Slawson, 1985; 1987), can be specified as categories of equal-values contours in this space. The attribute *openness* varies with **f(1)**, *acuteness* with **f(2)**, *smallness* with the sum of **f(1) + f(2)**, and *laxness* varies towards a neutral position in the middle of the space (Figure 3).

Figure 3: Two-dimensional sound space.



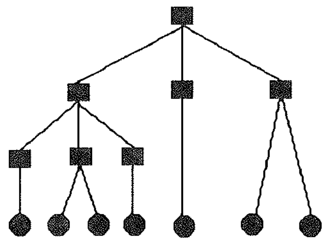
The notion of Abstract Sound Schema(ASS)

The Abstract Sound Schema (ASS) is the representation scheme we designed for describing a sound in terms of its perceptual *components* and the *relations* between them. The ASS scheme is constituted of: *nodes, slots,* and *links*. Nodes and slots are the components, and the links correspond to the relations between them. The links are labelled.

The ASS is, in fact, a tree-like abstract data structure whose ultimate nodes (the leaves) are slots. Each slot has a name and accommodates a sound synthesis datum.

Slots are grouped bottom up into higher level nodes, which in turn are grouped into higher

Figure 4: The ASS representation scheme.



Implementing a sound event by means of the schema

We have seen before (Figure 1) that the synthesiser is composed of several connected blocks (Voicing Source, Noise Source, etc.), one of each responsible for a certain sound attribute. We can now define a compound *sound event* by means of the ASS scheme. Each component of the *sound event* is responsible for a certain aspect of the sound quality.

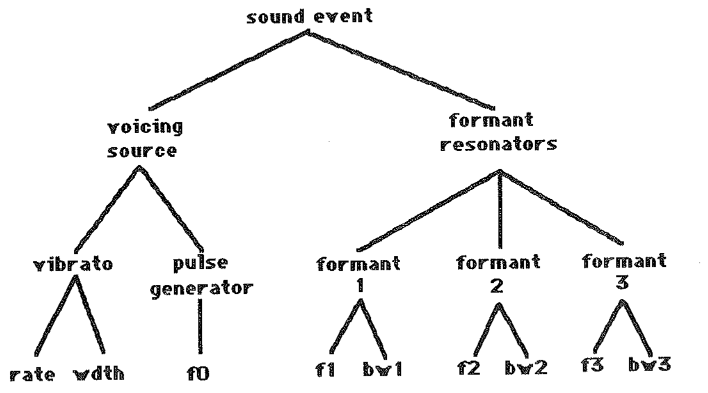
The leaves of the *sound event* are slots corresponding to the several sound synthesis parameters. Slots are grouped into nodes of a higher level layer, which in turn are grouped into nodes of a higher level, and so forth, up to the root of the tree (the *sound event*).

Figure 5 shows a partial definition of a *sound event* of the synthesiser shown in Figure 1. Although not shown in this figure, the links among the *sound-event*'s components are labelled *has_component*. They represent the offspring relation among nodes.

The partial *sound event* definition shown in Figure 5 can be implemented in Prolog as shown below. Each clause represents a *has_component* relationship between two atoms. The first clause, for example, is read: *'a sound event has a component called voicing source'*. A interpretation of the whole layer 1, for example, is: *'the sound event has two components named voicing source and formant resonators'*.

```
%%% layer 1
%%%
has_component( sound_event, voicing_source ).
has_component( sound_event, formant_resonators ).
%%%
%%% layer 2
%%%
has_component( voicing_source, vibrato ).
has_component( voicing_source, pulse_generator ).
has_component( formant_resonators, formant(1) ).
has_component( formant_resonators, formant(2) ).
has_component( formant_resonators, formant(3) ).
%%%
%%% layer 3
%%%
has_component( vibrato, rate ).                % vibrato rate
has_component( vibrato, wdth ).                % vibrato width
has_component( pulse_generator, f(0) ).        % fundamental frequency
has_component( formant(1), f(1) ).             % 1st formant frequency
has_component( formant(1), bw(1) ).            % 1st formant bandwidth
has_component( formant(2), f(2) ).             % 2nd formant frequency
has_component( formant(2), b(w2) ).            % 2nd formant bandwidth
has_component( formant(3), f(3) ).             % 3rd formant frequency
has_component( formant(3), bw(3) ).            % 3rd formant bandwidth
```

Figure 5: Partial sound event definition.



All the slots of the ASS must be filled in order to completely specify a sound. We say that a completely specified sound is an *assemblage.* For each different sound there is a particular assemblage. Thinking of this synthesiser as a (rough) model of the vocal tract mechanism, an assemblage would correspond to a certain position of the vocal tract in order to produce a sound.

Sound hierarchy and the inheritance mechanism

Recapitulating, we have defined a general abstract scheme for representing a sound. Then we defined and implemented the notion of the *sound event* by means of this scheme. We also introduced the idea of assemblage. It was explained that an assemblage occurs when all the slots of the scheme are properly filled. In this case, each assemblage corresponds to a particular sound.

In practice, sounds are represented in a knowledge base as a collection of slot values. In other words, the knowledge for the assemblage of a particular sound is clustered around a collection of slot values. An assemblage engine is then responsible for taking the appropriate slot values and 'assembling' the desired sound.

The following Prolog facts correspond to an example knowledge base which contains slot values for the (partial) *sound event* definition shown in Figure 5. Each clause represents a *slot*. It has two atoms: the first is a reference name and the second is a tuple. The reference name is an atom which identifies the affiliation of the slot, i.e. which cluster it belongs to. The first element of the tuple is the name of the slot and the second element is the value of the slot. This value can be either a number, a word, or a formula for calculating its value (these will be dealt later). This example knowledge base contains information about three sounds, namely *back vowel, front vowel,* and *vowel /a/.*

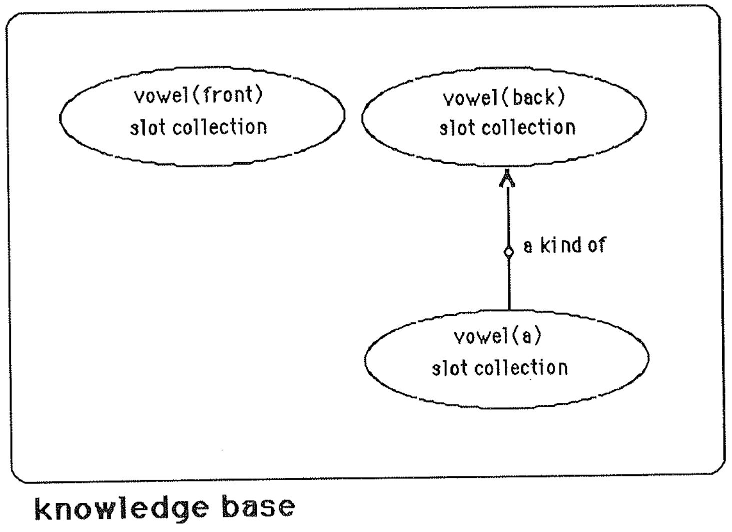
```
% %% back vowel
%%%
slot( vowel(back), [ rate, 5.2 ] ).
slot( vowel(back), [ wdth, 0.06 ] ).
slot( vowel(back), [ f(0), 155.56 ] ).
slot( vowel(back), [ f(1), 622.25 ] ).
slot( vowel(back), [ f(2), 1244.5 ] ).
slot( vowel(back), [ f(3), 2637 ] ).
slot( vowel(back), [ bw(1), 74.65 ] ).
slot( vowel(back), [ bw(2), 56 ] ).
slot( vowel(back), [ bw(3), 131.85 ] ).
%%%
%%% front vowel
%%%
slot( vowel(front), [ rate, 5.5 ] ) .
slot( vowel(front), [ wdth, 0.06 ] ).
```

```
slot( vowel(front), [ f(1), 559.37 ] ).
slot( vowel,front), [ f(2), 1108.7 ] ).
slot( vowel(front), [ f(3), 2637 ] ).
slot( vowel(front), [ bw(1), 78.3 ] ).
slot( vowel(front), [ bw(2), 110.8 ] ).
slot( vowel(front), [ bw(3), 131.85 ] ).
%%%
%%% vowel /a/
%%%
slot( vowel(a), [ a_kind_of, vowel(back) ).
slot( vowel(a), [ f(0), 103.83 ] ).
```

Note that the representation of the sound **vowel(a)** is different from the other two: it is incomplete (i.e. there are no slot values for the **vibrato** nor for the **formant resonators**). On the other hand, there is new information in it. The new information, called *a_kind_of,* is not a simple *sound event* slot, as it might appear to be, but it is a link (see Figure 6). This is a link which associates one collection of slots with other collection of slots.

The link *a_kind_of* allows a hierarchical organisation of the knowledge. The ability to represent the relationship between slot collections hierarchically is useful for inheritance relation. Inheritance is a relation by which an individual assumes the properties of its class and by which properties of a class are passed on to its subclass. Thus, when a slot collection for a sound is attached to another slot collection at a higher level, the former inherits properties of the latter. The first fact of the third cluster of slots listed above states that a **vowel(a)** is *a_kind_of* **vowel(back)**. This is to say that slots not defined for **vowel(a)** will be filled with slot values taken from **vowel(back)** (Figure 6). In practice, the assembler engine has to 'know' that the missing slots in one level are inherited from a higher level.

Figure 6: The example knowledge base has information about three sounds. Each sound is represented as a collection of slot values. Note that **vowel(a)** *inherits* slots from **vowel(back)**.



The notion of partial assemblage

We ought to make the assembler engine flexible so that it also may assemble single internal nodes of the schema. In other words, besides the assemblage of the whole scheme there might be (partial) assemblages of only certain nodes.

Let us observe again the example shown in Figure 5. It has a branch of filters which constitute three formant resonators. Taking as an example only the node **formant(1)** , we say that it needs only its affiliated slots (namely **f(1)** and **bw(1)**) for assemblage.

The advantage of being able to think in terms of assemblages of single nodes, as an alternative to the solely ASS root assemblage, is that now one can attach non-numerical attribute values (i.e. words in English) to partial assemblages too. For instance, one could refer to the node **formant(1)** as **low and wide** if it has **f(1) = 250 Hz** and **bw(1) = 200 Hz.** This is also represented in the knowledge base as a cluster of slots. Example:

```
slot( [ formant(1), low_and_wide ],  [ f(1),  250 ] ).
slot( [ formant(1), low_and_wide ],  [ bw(1),  200 ] ).
```

Now, for each node of the schema one can define a set of possible non-numerical attribute values. Back to figure 5, the slots **rate**, and **wdth** constitute a node called **vibrato** which in turn, with the node **pulse generator,** forms the higher level node **voicing source**. One could establish here that the possible attribute values for **vibrato** are **none, uniform,** and **too slow.** Each of these attributes will then correspond to either a numerical value or to a range of values within a certain interval. For example, one could say that **vibrato** is **none** if **rate = 0 Hz**, and **wdth = 0 %**. The node **voicing source** could be similarly defined: one could establish that **voicing source** is **steady low** if **vibrato = none** and **pulse generator = 55 Hz**, for example.

Hypothetically considering only this left part of the example schema shown in Figure 4, a sound, say **sound(a)**, could be described as *having steady low voicing source and none vibrato.* See example below:

```
slot( [ vibrato, none ] , [ rate, 0 ] ).
slot( [ vibrato, none ] , [ wdth, 0 ] ).
slot( [ voicing_source,  steady_low ],  [ vibrato,  none ] ).
slot( [ voicing_source, steady_low ], [ pulse_generator,  55 ] ).
slot( [ sound(a), [ voicing_source,  steady_low ] ),
slot( [ sound(a), [ vibrato,  none ] ).
```

The role of machine learning

In this section we will study the role played by two machine learning techniques in our proposed system, namely *inductive learning* and *supervised deductive learning.* Both are well known techniques which have been satisfactorily used in expert systems (see (Dietterich and Michalski, 1981; Quinlan, 1982; Winston, 1984; Bratko, 1990; Carbonell, 1990) for a survey).

The target of inductive learning here is to induce general concept descriptions of sounds from a set of examples. A further aim is to allow the computer to use automatically induced concept descriptions in order to identify unknown sounds or possibly suggest missing attributes of an incomplete sound description. Our main reason for inducing rules about sounds is that the computer can then aid the user to explore among possible alternatives during the design a certain sound. Here the user would be able to ask the system to *'play something that sounds similar to a bell'* or even *'play a kind of dull sound',* for example. In these cases the system will consult induced rules in order to work out which attributes are relevant for synthesising a bell-like sound or a sound with dull colour attribute (Smaill et al. 1993).

An example rule, when looking for a description for, say **sound(c)**, on the basis of some examples, could be as follows:

```
sound(c) = { [ vibrato = fast ], [openness =high ] }
```

The interpretation of the above rule is as follows:

A sound is **sound(c)** if:
    it has **fast vibrato** and
    high openness.

No matter how many attributes **sound(c)** had in the training set, according to the above rule, the most relevant attributes for this sound are **vibrato = normal** and **openness = high**. *'Most relevant'* here means what is most important for distinguishing **sound(c)** form other sounds of the input training set. In this case, if the system is asked to synthesise a sound with **fast vibrato** and **high openness**, then it will produce **sound(c)**.

The target of supervised deductive learning in our system is to allow the computer to update its knowledge about attribute values throughout user interaction. We remind the reader the fact that the input requirement for producing a sound can contain either or both: attribute values (e.g. **vibrato = none**) or slot values (e.g. **f(0) = 55 Hz**). The aim of supervised deductive learning here is at allowing the computer to infer whether or not input slot values (in a requirement) match with known attribute values. If there is no matching, then the system automatically adds this yet unknown information to the knowledge base and asks the user to give a name for this novel deduced attribute value. Suppose that the system knows three values for the attribute **vibrato:**

vibrato = uniform	if { **rate = 5.2 Hz, wdth = 3 %** }
vibrato = too slow	if { **rate = 3.6 Hz, wdth = 3 %** }
vibrato = none	if { **rate = 0 Hz, wdth = 0 %** }

If the user requires a sound with (vibrato) **rate = 12 Hz**, for example, then the system will synthesise it and deduce that there is no attribute value for vibrato in the knowledge base whose **rate** is equal **12 Hz**. In this case the system adds this new information to the knowledge base, works out the other slot values needed to create this new attribute value, and ask the user to name it. Let us say, for example, that the user wishes to call it **tremolo**. Eventually the system will add the following information in its knowledge base:

vibrato = tremolo	if { **rate = 12 Hz, wdth = 3 %** }

Towards a system architecture

User configuration is one of the desirable capabilities of this system. Therefore rather than providing a closed architecture which reflects both a particular synthesiser and a particular vocabulary for sound description, we propose an architecture which provides open-ended modules (Figure 7).

This system architecture provides means for handling information about sound synthesis but it remains open-ended regarding what the information is about.

Engines and services provided by the system

The role of the *assembler engine* and the functioning of the *machine learning engine* modules have already been introduced.

The *machine learning engine* module performs the two kinds of learning: *inductive learning* and *supervised deductive learning*, just discussed above. The training set for the inductive learning mechanism is given either by the user or it is automatically produced by the system by consulting its own knowledge base. The input for the supervised deductive learning mechanism is provided partly by the system and partly by the user.

The *user interface* module provides means for communicating with the system. Here the user can activate the assembler engine in order to produce a sound, consult the status of the system (such as the content of the *induced rules* module and the content of the *knowledge base* module), and input any external information the system might need (such as the names for new sounds and attributes, and training sets).

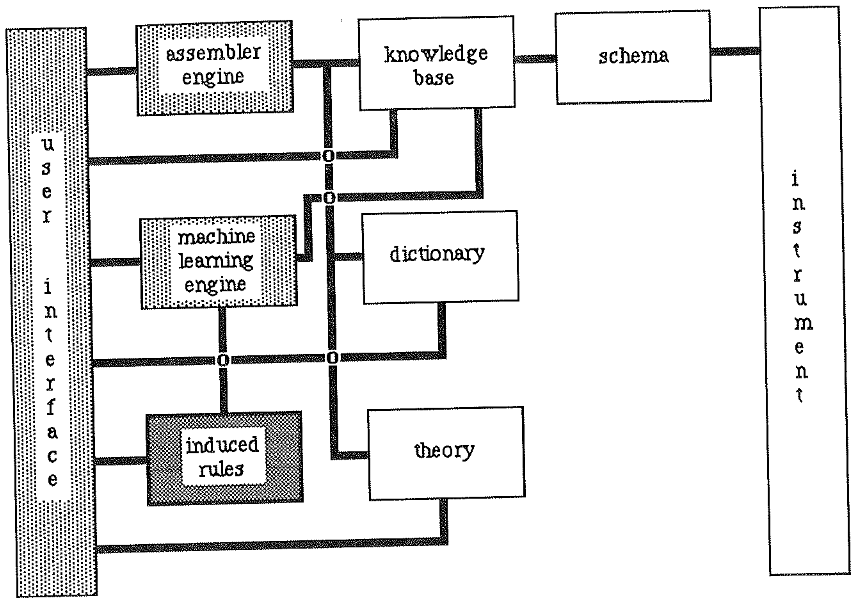
Information internally generated and administered by the system

The *inductive rules* module holds the information internally generated by the system as the result of the inductive learning. As its name suggests this module contains rules which were induced by the *machine learning engine* module.

User specified modules

These are the open-ended modules. They define the domain of the system, that is, the sonic world the system will deal with. Here the user implements the instrument (i.e. the synthesis algorithm), the schema on the top of it, the knowledge base whose information is used to 'play' it, a dictionary of slot values, and a theory for the instrument, using Prolog (Bratko, 1990).

Figure 7: The proposed system architecture.



The modules of the architecture are classified in three groups:

- □ User specified modules
- ▩ Information automatically generated by the system
- ▒ Engines and services provided by the system

Default libraries of such modules can be provided in case the user does not wish start from scratch. However, as these modules are to be user customised, it may not always be very useful to exchange highly customised libraries with other users.

Firstly, the user specifies the *instrument* module. This can be done by means of any suitable SWSS (Software for Sound Synthesis) package, such as CLM, (Schotstaedt, 1992), Csound (Vercoe, 1991), Mosaïc (Morrison and Waxman, 1991), or ISPW Max (Puckette et al., 1992), to name but a few. Having specified the instrument then the user implements the *schema* on the top of it. Secondly, the *knowledge base* module is specified. In this module the user creates clusters of slot values. As it was mentioned before, each cluster corresponds to an instantiation of either a whole sound event or an internal node of the schema, i.e. a sound attribute. Thirdly, the user builds the *dictionary* module. In this module the user specifies the meaning of the vocabulary for speaking about slots, i.e. about each parameter of the instrument. Each word of the vocabulary for slots may mean either a numerical synthesis parameter or a pointer to a formula for calculating it. Finally, the user specifies a *theory* for the instrument. A theory is a set of formulas for calculating slot values. These formulas can calculate values either based on other slot values or by the random choice of a value within a certain interval.

As the system is to start with a certain body of knowledge which will be expanded through user interaction, these specifications do not need to be exhaustive.

An example functioning

Let us study an example functioning of the architecture explained above. Assume that the following information can be used in order to assemble the scheme of Figure 5.

Knowledge base module:

```
slot( sound_event( sound(c) ), [ rate, fast] ).
slot( sound_event( sound(c) ), [ wdth, default ] ).
slot( sound_event( sound(c) ), [ f(0), low ] ).
slot( sound_event( sound(c) ), [ openness, high ] ).
slot( sound_event( sound(c) ), [ acuteness, low ] ).
slot( sound_event( sound(c) ), [ f(3), 2637 ] ).
slot( sound_event( sound(c) ), [ bw(3), 131.85 ] ).

slot( attribute( [ openness, low ] ), [ f(1), low ] ).
slot( attribute( [ openness, low ] ), [ bw(1), 74.65 ] ).

slot( attribute( [ openness, high ] ), [ f(1), high ] ).
slot( attribute( [ openness, high ] ), [ bw(1), 78.3 ] ).

slot( attribute( [ acuteness, low ] ), [ f(2), low ] ).
slot( attribute( [ acuteness, low ] ), [ bw(2), 110.8 ] ).

...
etc.
```

Dictionary module:

```
dict( slot( f(1) ), [  value( low,       290 ),
                       value( medium,    400 ),
                       value( high,      650 ) ] ).

dict( slot( f(2) ), [  value( low,       1028 ),
                       value( medium,    1700 ),
                       value( high,      1870 ) ] ).

dict( slot( f(0) ), [  value( low,       220 ),
                       value( medium,    rule( f(0), medium ) ),
                       value( high,      rule( f(0), high ) ) ] ).

...
etc.
```

Theory module:

```
instrument_theory( rule( f(0), medium ), F0 ):-
                    get_value( f(0), low, V ),
                    F0 is V * 2.
...
etc.
```

Suppose that a training set has been input and the system has already induced some rules, such as:

```
sound(a) = { [openness = low ] }
sound(b) = { [ f(0) = medium ] }
sound(c) = { [ rate = fast ], [ f(0) = low ] }

...
etc
```

Now, let us suppose two hypothetical queries and examine what ARTIST would do in order to compute them.

Example query 1:

Produce a sound with fast vibrato rate and low pitch.

ARTIST functioning 1:

*Firstly, the system consults the induced rules in order to find out if it knows any sound whose most prominent features are **rate = fast** and **f(0) = low**. In this case, there is a rule which tells that **sound(c)** matches this requirement. Thus, **sound(c)** will be produced. Before assembling the schema, the system consults the dictionary in order to compute the slots whose values are represented by a word (e.g. **f(0) = low** actually means **220 Hz**).*

Example query 2:

Produce a sound with medium pitch and high openness.

ARTIST functioning 2:

*In this case the system has no matching induced rules. Thus, this sound will be created from scratch. The system consults the dictionary in order to compute the values of **f(0) = medium** and **f(1) = high**, and automatically completes the missing slot data with default values. Note that instead of a value for **f(0) = medium**, the dictionary points to a rule. In this case, the system consults the theory module in order to calculate it. The theory says that this value corresponds to the double value of **f(0) = low**. Therefore, **f(0) = medium** here means **440 Hz**. The sound is then produced, the user is asked to name it, and a novel cluster of slot values is automatically created in the knowledge base for representing it.*

Conclusion and further work

In this paper we introduced the fundamentals of ARTIST: a tool for the design of intelligent assitants for sound synthesis.

ARTIST is provided with some degree of automated reasoning which supports the laborious and tedious task of writing down number sequences for generating a single sound on a computer. A 'synthesiser' implemented by means of ARTIST is provided with a certain knowledge about sound synthesis and it is able to infer the necessary parameters values for a sound from a quasi-natural language sound description.

Although the user has to specify the information of the knowledge base (i.e. the synthesis algorithm(s) and the vocabulary for sound description) beforehand, this does not necessarily need to be exhaustive. The system is to begin with a minimum amount of information about certain sounds and attributes, but it is able to automatically expand the scope of its knowledge by acquiring new information through user interaction.

At the moment we are developing a higher level interface for the user specified modules. We wish to enable the user to specify these modules by means of natural language-like statements, instead of Prolog. We also plan to devise a visual interface for the specification of some attributes, such as envelopes, for example.

ARTIST is being tested using synthesis by physical modelling technique (Roads, 1993). It seems that this technique matches many of the concepts developed in this paper, such as the representation of sound attributes and the mapping of these to synthesis parameters.

We are aware that ARTIST is still in its infancy. For the moment we regard it as a suggestive and plausible starting point only.

References

Barrière, J-B. (1989), Computer music as cognitive approach: Simulation, timbre and formal processes, in *Contemporary Music Review*, Vol. 4, pp. 117-130, Harwood Academic Publishers.
Bratko, I. (1990), Prolog programming for Artificial Intelligence, Addison-Wesley Publibshers.
Carbonell, J. (1990) (Editor.), *Machine Learning: paradigms and methods*, The MIT Press.
Carpenter, R. H. S. (1990), *Neurophysiology*, Physiological Principles of Medicine Series, Edward Arnold.
Cogan, R. (1984), *New Images of Musical Sound*, Harvard University Press.
Dodge, C. and **Jerse**,T. (1985), *Computer Music*, Schirmer Books.
Dietterich, T. and **Michalski**, R. (1981), Inductive Learning of Structural Descriptions, in *Artificial Intelligence*,

Eckel, G. (1993), La Maître de la Synthèse Sonore, in *La Synthèse Sonore*, Les cahiers de l'Ircam Nr. 2, pp. 97-106, Ircam - Centre Georges Pompidou.
Ethington, J. and **Punch**, D. (1994), SeaWave: A system for Musical Timbre Description, in *Computer Music Journal*, Vol. 18, Nr. 1, pp. 30-39, The MIT Press.
Flanagan, F. (1984), Voices of Men and Machines, in *Electronic Speech Synthesis*, Bristow, G. (Editor.), Granada.
Garton, B. (1989), The Elthar Program, in *Perspectives of New Music*, Vol. 27, Nr. 1, pp. 6-41.
Giomi, F. and **Ligabue**, M. (1992), *Analisi Assistita al Calcolatore della Musica Contemporanea*, Rapporto Interno C92-01, CNUCE/CNR, Conservatorio di Musica L. Cherubini (Italy).
Helmholtz, H. L. F. (1885), *On the sensations of tone as a physiological basis for the theory of music*, Longmans, Green and Co.
Holtzman, S. R. (1978), A description of an automated digital sound synthesis instrument, *DAI Research Report No. 59*, Dept. of AI, University of Edinburgh.
Klatt, D. H. (1980), Software for a cascade/parallel formant synthesiser, in *Journal of Acoustic Society of America*, Vol. 67, Nr. 3, pp. 971-995.
Keefe, D. H. (1992), Physical Modeling of Wind Instruments, in *Computer Music Journal*, Vol. 16, Nr. 4, pp. 57-73, The MIT Press.
Lerdhal, F. (1987), Timbral Hierarchies, in *Contemporary Music Review*, Vol. 2, pp. 135-160, Harwood Academic Pulbishers.
Lohner, H. (1986), The UPIC System: A User's Report, in *Computer Music Journal*, Vol. 10, Nr. 4, pp. 42-49, The MIT Press.
Luger, G. F. and Stubblefield, W. A. (1989), *Artificial Intelligence and the design of Expert Systems*, Benjamin/Cummings.
Marino, G., Serra, M-H., and Raczinski, J-M. (1993), The UPIC System: Origins and Innovations, in *Perspectives of New Music*, Vol. 31, Nr. 1, pp. 258-269.
Miranda, E. R., **Smaill**, A., and **Nelson**, P. (1993a), A Symbolic Approach for the design of Intelligent Musical Synthesisers, in Proceedsing of the X Reunión Nacional de Inteligencia Artificial in Mexico City, Megabyte/Noriega Editores.
Miranda, E. R., **Smaill**, A., and **Nelson**, P. (1993b), A Knowledge-based approach for the design of Intelligenct Musical Instruments, in *Proceedings of the X Sympósio Brasileiro de Inteligência Artificial in Porto Alegre*, pp. 181-196, SBC/UFRGS.
Miranda, E. R. (1992), *Towards an Acousmatic Singer*, Research Report Nr. 1, Faculty of Music, University of Edinburgh.
Miranda, E. R. (1993), Modelagem do Aparelho Fonador e suas Aplicações na Música, in *Acústica & Vibrações*, Journal of the Acoustic Society of Brazil (SOBRAC), Nr. 12, pp. 60-74.
Miranda, E. R. (1994a), From Symbols to Sound: Artificial Intelligence Investigation of Sound Synthesis, in *Contemporary Music Review* (in press), Harwood Academic Publishers.
Miranda, E. R. (1994b), The Role of Artificial Intelligence in Computer-aided Sound Composition, in *Journal of Electroacoustic Music* (in press), Sonic Arts Network.
Miranda, E. R. (1994c), Towards an Intelligent Assistant for Sound Design, in *Musical Praxis*, Vol. 1, Nr. 1, pp. 53-57, Faculty of Music, Edinburgh University.
Morrison, J. and **Waxman**, D. (1991), *Mosaïc 3.0 Reference Manual*, Ircam.
Pennycook, B. W. (1985), Computer-Music interfaces: A Survey, in *Computing Surveys*, Vol. 17, Nr. 2, pp. 267-289.
Puckette, M., **Lippe**, C., and **Waxman**, D. (1992), *ISPW Max Reference Manual*, Preliminary Release 0.17, Ircam.
Roads, C. (1993), Initiation à la Synthèse par Modèles Physiques, in *La Synthèse Sonore*, Les cahiers de l'Ircam Nr. 2, pp. 145-172, Ircam - Centre Georges Pompidou.
Quinlan, J. R. (1982), Semi-autonomous Acquisition of Pattern-based Knowledge, in *Introductory Reading in Expert Systems*, Michie, D. (Editor), Gordon&Breach.
Schaeffer, P. (1966), *Traité des objets musicaux*, Ed. du Seuil.
Schmidt, B. L. (1987), Natural Language Interface and their application to Music Systems, in *Proceedings of the 5th Audio Engineeting Society International Conference*, pp. 198-206.
Schottstaedt, W. (1992), *Common Lisp Music Documentation*, available via Internet ftp from the clm directory on the host machine ccrma-ftp.Stanford.edu.
Slawson, W. (1985), *Sound Color*, University of California Press.
Slawson, W. (1987), Sound-color Dynamics, in *Perspectives of New Music*, Vol. 25, No. 1&2, pp. 156-179.
Smaill, A., Wiggins, G. A., Miranda, E. R. (1994), Music Representation - between the Musician and the Computer, in *Music Education: An Artificial Intelligence Approach*, Smith, M. et al. (Editors.), Workshops in Computing Series, Springer-Verlag, pp. 108-119.
Spender, N. (1980), Psychology of music (I-III), in *The New Grove's Dictionary of Music and Musicians*, Sadie, S. (Editor), Vol. 15, pp. 388-427, Macmillan Publishers.
Sundberg, J. (1991), Synthesising Singing, in *Representation of Musical Signals*, De Poli et al. (Editors), The MIT Press.
Terhardt, B. (1974), On the Perception of Periodic Sound Fluctuation (Roughness), in *Acustica*, Vol. 30, pp. 201-213.
Vercoe, B. (1991), *Csound Manual*, available via Internet ftp from the music directory on the host machine

Vertegal, R. and **Bonis**, E. (1994), ISEE: An Intuitive Sound Editing Environment, in *Computer Music Journal,* Vol. 18, Nr. 2, The MIT Press.
von Bismark, G. (1971), Timbre of Steady Sounds: Scaling of Sharpness, in *Proceedings of the* 7th *Internacional Congress on Acoustics in Budapest*, Vol. 3, pp. 637-640.
von Bismark, G. (1974a), Timbre of Steady Sounds: A Factorial Investigation of its Verbal Attributes, in *Acustica*, Vol. 30, pp. 146-158.
von Bismark, G. (1974b), Sharpness as an Attribute of the Timbre of Steady Sounds, in *Acustica*, Vol. 30, pp. 159-172.
Winston, P. (1984), *Artificial Intelligence*, (2nd ed.), Addison-Wesley.
Woodhouse, J. (1992), Physical Modeling of Bowed Strings, in *Computer Music Journal,* Vol. 16, Nr. 4, pp. 43-56, The MIT Press.
Xenakis, I. (1963), Musiques Formelles, in *La Revue Musicalle,* double issue Nrs. 253-254, Editions Richard-Masse.
Xenakis, I. (1971), *Formalized Music: Thought and Mathematics in Music Composition,* Indiana University Press.
Xenakis, I. (1992), *Formalized Music: Thought and Mathematics in Music*, Pendragon Press, revised edition.

Representing Musicians' Actions for Simulating Improvisation in Jazz

Geber Ramalho

LAFORIA-IBP-CNRS
Université Paris VI
4, Place Jussieu
75252 Paris Cedex 05 - FRANCE
Tel. (33-1) 44.27.37.27
Fax. (33-1) 44.27.70.00
e-mail: ramalho@laforia.ibp.fr

Abstract

This paper considers the problem of simulating Jazz improvisation and accompaniment. Unlike most current approaches, we try to model the musicians' behavior by taking into account their experience and how they use it with respect to the evolving contexts of live performance. To represent this experience we introduce the notion of *Musical Memory*, which exploits the principles of Case-Based Reasoning (Schank & Riesbeck 1989). To produce live music using this *Musical Memory* we propose a problem solving method based on the notion of PACTs *(Potential ACTions)* (Ramalho & Ganascia 1994b). These PACTs are a generic framework for representing the musical actions that are activated according to the context and then combined in order to produce notes.

1 - Introduction

This paper considers the problem of simulating the behavior of a bass player in the context of Jazz live performance. We have chosen to work on Jazz improvisation and accompaniment because of their spontaneity, in contrast to the formal aesthetic of contemporary classical music composition. From an AI point of view, modeling Jazz performance raises interesting problems since performance requires both theoretical knowledge and great skill. In addition, Jazz musicians are encouraged to develop their musical abilities by listening and practicing rather than studying in *conservatoires* (Baker 1980).

In Section 2 we present briefly the problems of modeling musical creativity in Jazz performance. We show the relevance of taking into account the fact that musicians integrate rules and memories dynamically according to the context. In Section 3 we introduce the notion of PACTs, the basic element of our model. In Section 4, we give a general description of our model and show particularly how the composition module integrates the two above-mentioned notions to create music. In the last section we discuss our current work and directions for further developments.

2 - Modeling Musical Creativity

2.1 - The Problem and the Current Approaches

The tasks of improvisation and accompaniment consist in playing notes (melodies and/or chords) according to guidelines laid down in a given chord grid (sequence of chords underlying the song). Musicians cannot justify all the local choices they make (typically at note-level) even if they have consciously applied some strategies in the performance. This is the greatest problem of modeling the knowledge used to fill the large gap referred to above (Ramalho & Pachet 1994). To face this problem, the first approach is to make random-oriented choices from a library of musical patterns weighted according to their frequency of use (Ames & Domino 1992). The second approach focuses on very detailed descriptions so as to obtain a complete explanation of musical choices in terms of rules or grammars (Steedman 1984). Regardless of its musical results, the random-based approach