# A FRAMEWORK FOR REPRESENTING KNOWLEDGE ABOUT SYNTHESIZER PROGRAMMING

Pierre-Yves Rolland, François Pachet
LAFORIA-IBP, Université Paris 6
Boîte 169, 4, Place Jussieu, 75252 Paris Cedex
E-mail: rolland/pachet@laforia.ibp.fr

**Abstract**
We aim at capturing the know-how of expert patch programmers to build more productive human interfaces for commercial synthesizers. We propose a framework to represent this superficial layer of knowledge, that can be used to help musicians program commercial synthesizers patches. Our key idea is to classify sounds according to transformations experts can apply to them. We propose a dual representation of sounds combining object-oriented programming with classification mechanisms. We illustrate our framework with a prototype system that helps program Korg-05R/W synthesizers.

**Key Words:** Computer-Aided Synthesis, Desription Logics, Object-Oriented programming, Smalltalk

## From Computer Aided Synthesis to Computer Aided Synthesizer Programming

Our framework stems from the following remark. In an intensive care unit, a typical nurse knows perfectly well how to use an infusion pump. However, the nurse may be an "expert" in infusion pump manipulation without necessarily having any theoretical knowledge of the device nor of medical issues related to diagnosis, treatment of patients, and so on. She may even be more expert than the doctor himself. But she knows what she needs to know: some kind of superficial knowledge about how to use the machine productively and safely. By analogy, we compare 1) expert *patch programmers*, who know how to effectively program synthesizers, and 2) experts in *sound synthesis*, who know everything about Fourier transforms and digital processing, but who would be hard pressed to program, say, a "fatter" sound on a commercial synthesizer (such as a Yamaha SY99, or Korg 05R/W).

We are interested in representing the knowledge manipulated by the first category of experts, and claim that it is possible to capture substantial chunks of this knowledge, in order to produce better man-machine patch programming interface.

**The CAS paradigm**

This observation is the basis of our approach in the domain of computer-aided synthesizer programming (CASP). From our point of view, the two main goals of CAS are 1) to help musicians create sounds, and 2) to help musicians understand a synthesis technique.
In order to achieve these goals, CAS systems should have the following characteristics. First, it should propose a consensual, common vocabulary, shared between the machine and the user. In particular, the machine and the user must reach an agreement on ways to describe certain perceived sound characteristics (amplitude, pitch, timbre, etc.), for instance under the form of adjectives (decaying, high-pitch, warm etc.). Moreover, a CAS system should represent knowledge about the synthesis technique itself. For example, a CAS system might encapsulate knowledge corresponding to the following questions : what is the synthesis model (e.g. : FM (Chowning 1973), additive synthesis (Risset 1969; Serra 1989)) ? Is there a physical interpretation of that model ? What synthesis parameters are available (e.g. carrier frequencies, number of partials etc.) ? What are the constraints between the various parameter values, particularly in relation to the physical sound-making phenomenon modeled ? Do particular parameters have direct influences on perceived sound characteristics ?

**Background in CAS systems**

The CHANT system (Rodet et al. 1984) is based on the *synthesis by rule* concept. Implementing the Formant-Wave-Functions (FOF) model, CHANT addresses the synthesis of the singing voice, among others. Despite the large number of synthesis parameters and the complexity of parameter-sound relations, the user can have the system produce realistic singing sounds by just specifying "surface aspects" such as: which vowel is to be obtained, at what pitch etc. ISEE, an Intuitive Sound Editing Environment (Vertegaal and Bonis 1994) allows direct manipulation of timbre in a four-dimension "timbre space" whose coordinates correspond to timbre parameters, such as "brightness" which controls the spectral energy distribution, or "overtones". This approach, which builds upon the concept of 'encapsulation of synthesis expertise in the synthesis model' (Vertegaal and Bonis 1994), captures expertise limited to the control of a fixed set of four timbre parameters that have known sound signal characteristics counterparts, e.g. spectrum properties. CHANT and ISEE are powerful CAS environments that can deal with different synthesis models, but do not directly

address the issue of encapsulating practical expertise pertaining to one given synthesizer.

The prototype synthesizer implemented in ARTIST (Miranda 1994) produces human voice sounds using subtractive synthesis. Sounds are described at different abstraction levels using slot values and user-definable attribute values. Designed to be an adaptive assistant, ARTIST uses the Artificial Intelligence technique of Inductive Learning (Michalski 1983) to increase its knowledge about synthesis through user interaction. The idea of transforming sounds has been introduced in ARTIST through the Sound Tools module. However the transformations proposed correspond to elementary operators which, for instance, change the sound's fundamental frequency.

The Kyma/Platypus platform (Scaletti 1989) is a framework for defining and manipulating sound objects. Each sound represents a stream of samples, and can be defined as a composition of other sound objects or a transformation operating on sound objects. Kyma is intended to give the composer or musician direct access and control on the sound structure, and provides tools to organize complex sounds. The Kyma platform has successfully been applied in a variety of applications like the Javelina system (Hebel 1989). However, the system as such does not aim at capturing consensual knowledge about sound synthesis, but rather provides a framework in which composers may define and manipulate their own organizations of sounds. In a similar spirit, the DMIX system (Openheim 1989) is an environment for performance and composition. It proposes sophisticated tools and editors to be used by a composer or a performer. Kyma and Dmix are both exemplary in their use of object-oriented programming, but they do not make use of particular knowledge representation techniques.

The study of existing CAS stems the following two observations. First, little research has been concerned with commercial synthesizers. Most of the systems address sophisticated synthesis techniques, which are not commonplace in the community of real musicians. Second, most systems propose environments in which musicians can define, control and manipulate their own sound structures. No system proposes, yet, to represent the common sense knowledge experts have on patch programming.

This is what motivates our attempt to study *CASP* (Computer-Aided Synthesizer Programming, as opposed to Computer-Aided Synthesis per se.

## Assumptions

Our approach to CASP is based on the following two assumptions:

**A1. Commercial synthesizer programmers use superficial knowledge**

Our hypothesis is that commercial synthesizer programmers use lots of know-how and very little deep knowledge. The case of FM synthesis is particularly interesting in this respect. The success story of the DX series showed that lots of famous patches were designed by people who understood only a limited fraction of the underlying - and complex - FM theory. Instead, they would use some kind of superficial knowledge about the complex interactions between the DX parameters, gained from experience or by studying other patches. Attempts to widespread FM theory such as (Chowning and Bristow 1986) showed that understanding FM theory played only a minor role in programming effectively DX synthesizers. Although the book does contain some theoretical material, emphasis is clearly put on giving practical hints to musicians. Following is a typical "know how" rule (R) for FM synthesis; programmers can use it to build sounds without being aware of its theoretical meaning :
(R) You can get a "tremolo" sound (modulated amplitude) by setting the carrier frequency to a fixed, low value and the modulator to a variable value, i.e. which depends on the note played.

**A2. Most of the expertise lies in the transformation knowledge**

It is a well known fact that synthesizer experts have trouble teaching how to make a sound from scratch. However, they are much more at ease in explaining how to transform one sound into another.
Based on this observation, we think that most of the expertise lies in the transformation knowledge. This leads us to represent the knowledge associated to transformations, rather than to sound structures themselves.
Following are two typical examples of transformation rules that we want our system to take into account. (R1) is a typical transformation rule that may apply to virtually any kind of commercial synthesizer providing parameters for a filter section. The second one, (R2), also applies to most commercial synthesizers, but is valid only for "harmonic" sounds, i.e. sounds that have a perceived pitch, like piano tones, as opposed to cymbal tones:
(R1) You can make a sound *brighter* by increasing the low-pass filter's cutoff frequency.
(R2) You can make a sound *warmer* by duplicating the sound and applying a slight detune - typically 1/10 tone - to the duplicate.

## Transformation rules and sound classification

### Organization of knowledge

Transformation rules such as the ones above are easy to represent in a knowledge-based system, e.g. by means of production rules. Of course, as the examples show, not all transformations are applicable to all kind

of sounds. More precisely, transformations apply to particular *origin sound types* to produce *target sound types*. Target sound types are characterized here by simple adjectives, such as "warm", "bright", "squeaky", etc. In order to organize those sound types, we need some kind of classification scheme. Classifying target sound, based for instance on results of work in the area of timbre perception (Wessel 1979; Grey 1975), is irrelevant in our context: we are interested in classifying sounds according to what future transformations they can afford, and not according to the transformations that produced them. For example, we are not interested in classifying "brassy" sounds within a given typology of sounds. Rather, we are interested in identifying "sounds-that-may-become-brassy", i.e. sounds for which we know of a particular transformation that can make them "brassy". In this scheme being an instance of a sound type means nothing more than being able to undergo the various transformations associated to that particular sound type, thus sound type names are not significant. However, since we need names (e.g. for browsing purposes), we will, by convention, name sound types by suffixing them with "able", e.g. "brassy-able" or "warm-able". When a sound type affords several transformations, we give it a compound name, such as "brassy-and-bright-able". Consequently, each transformation is associated to a particular sound type in this classification. For instance, transformation (R2) is associated to sound type "warm-able", (R1) is associated to "brassy-bright-able", and so forth.

The hierarchical relation linking a sound type to one or more parent sound types indicates a specialization relation. This relation will be interpreted by a classification mechanism to classify new sounds. As a consequence, transformations associated to a sound type are inherited: they are also applicable to its children.

A part of a sound hierarchy for the Korg 05R/W synthesizer is shown on Figure 1. Sound types are represented as boxes, while names under boxes indicate applicable transformations. Arrows represent type/sub type relationships.

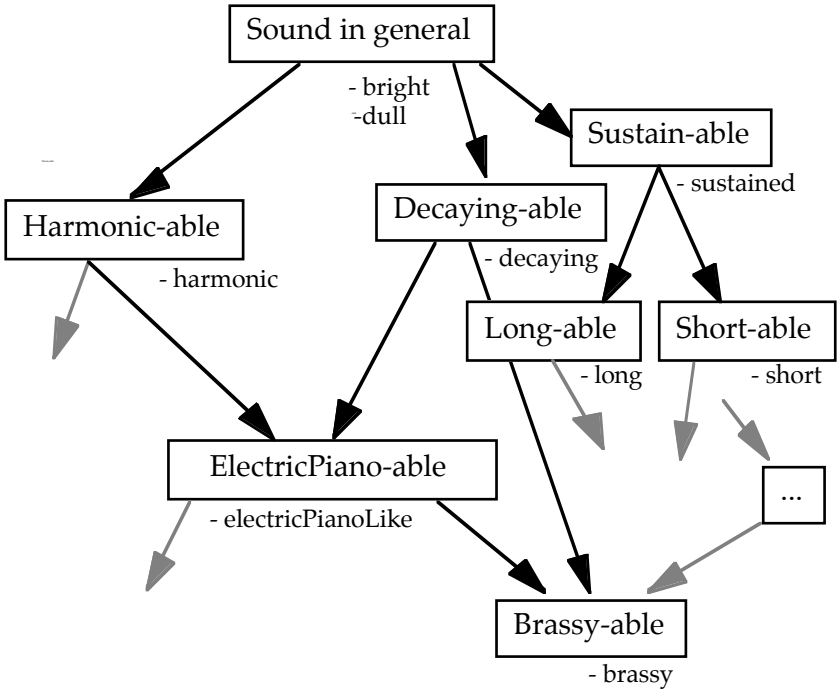


*Figure 1. A part of the sound type hierarchy for the 05R/W synthesizer*

The main task of the system is therefore to classify a sound according to a pre-defined classification. We will now describe our representation framework, with which we represent sound types.

**The representation framework**

The characteristics of the knowledge we want to represent about sound transformations led us to look for a representation framework integrating both 1) classification facilities to manipulate sound hierarchies and 2) a procedural language to express transformations of the current sound. Several attempts have been made to integrate term classification facilities within object-oriented programming languages (Cf. e.g. (Yelland 1992)) but integrated system are not commercially available yet. Moreover, there is a tradeoff to choose in classification systems, between expressiveness, efficiency and completeness. Among the various available classification systems (Heinson et al. 1994), we were particularly interested in the compromise proposed by the BACK system (Hoppe et al. 1993), an implementation of the Description Logics formalism (see section 4).

*Description logics to classify sounds*

An important characteristics of the sounds we want to model is that — at least on a superficial level — they may be described in a symbolic fashion, and not simply as an array of parameter values. For instance, a sound in the Korg 05R/W decomposes into one to sixteen *voices*. Each

voice in turn decomposes into one or two *tones*, each characterized by a *filter section* , an *amplification section*, and so forth. These abstract structures are eventually described in terms of actual parameters having terminal values, such as attack rate, attack level, sample name, effect number, etc.
This strong structuring of sounds is well captured by term classification techniques in which structural relations are stated in a declarative manner by the user and the system handles all inferences, i.e. classification and subsumption.

*OOP to represent transformations*

On the other hand, we need to represent transformations effectively and not only at an abstract level. As we will see, the formalism of Description logics does not allow to represent and organize transformation easily. Moreover, MIDI communication as well as user interface are typically a lot easier to program using object-oriented techniques. Sounds may be represented as instances of sound classes, and transformations as methods for these classes. We chose Smalltalk-80 for its acknowledged ease of use, and for its widespread use in the computer music research community (see CMJ, 13 (2), 1989 for example).

*A scheme to link both representations*

Having two different knowledge representation paradigms coexist is not straightforward, mainly because 1) information is redundant. A sound for instance will be represented both as a concept instance in BACK and as a class instance in Smalltalk and 2) the representations are not compatible as there is no clear relation between a concept and a class. We propose a scheme for coupling the two representations (section 6).

## The classification-oriented representation of sounds

### Description logics and BACK

For about fifteen years a number of research efforts in knowledge representation have focused on the development of representation languages with well-founded semantics, called *description logics* (DLs, previously called *terminological logics*). DLs propose a hierarchical knowledge organization in which every *concept* inherits information from more general ones. Also, the basis for an structuring of concepts is provided under the form of inter-concept links or *roles*.
More than a dozen DL implementations, referred to as Terminological Representation Systems, have been proposed to date (Heinsohn et al. 1994), derived from the early system KL-ONE (Brachman 1985). In such systems, information retrieval can take diverse forms including

*subsumption tests* and *classification requests*, as we will be explain now.

*A brief overview of BACK V5*

Our Terminological Representation System is the Berlin Advanced Computational Knowledge representation system (BACK). In our project we have used the Prolog-based, public domain BACK Version 5 system. In this sub-section BACK V5's main features are described. More details can be found in (Hoppe et al. 1993).

Concepts are sets of objects specified either intensionaly or extensionaly, and objects are instances of one or more concepts. In this work we have been using *primitive* and *defined* concepts. Conditions specified in the introduction of a concept are *necessary* conditions for the classification of its instances when the concept is primitive, *necessary and sufficient* conditions when the concept is defined. Roles are binary relations between objects. For any given object o, the objects $o_1$, ..., $o_n$ that are linked to o through the binary relation R corresponding to role r, (i.e. $\forall$ i, i $\in$ {1,...n}, R(o, $o_i$)), are called *role fillers* for role r and object o. Numerous examples will be provided in the next sections to illustrate all of these rather abstract notions.

Providing information to BACK V5 can be done through term introductions, object creations - called *assertions* - or through the use of non-definitional rules (RNDs) which impose particular logical relations between given concepts and roles. Besides, various kinds of information can be retrieved. Retrieving the result of the classification for an object o yields the list of concepts o instantiates. Querying the system for concept subsumption, on the other hand, produces a Boolean answer as to whether a concept c1 is more specific than a concept c2, i.e. whether any instance of c1 is also an instance of c2.

*Description of sounds as seen by the synthesizer manufacturer*

In the Korg 05R/W, a sound is internally represented as the list of synthesis parameter values, which are numbers or character strings. The "technical description" of sounds corresponds to this low-level representation. Figure 2 shows the synthesis architecture as presented by the manufacturer. The signal available at the synthesizer's outputs is obtained by adding the signals produced by up to 16 independent structures and passing the resulting signal through an effect processor. Each structure includes one or two interdependent sub-structures. Using technical terms, a sound is produced by mixing 1 to 16 independent *voices*—each made up of one or two *tones*—eventually processed by some effect processors.

In Figure 2, square boxes represent audio signal generation or processing units (e.g., Oscillator, Variable Digital Filter (VDF), Effect Processor, Panoramics unit) while rounded cornered boxes represent envelope or modulation generating units (e.g. VDF Envelope Generator).

Envelope generators produce angled line type functions which are used to control signal processing unit parameters, as explained below. Similarly, modulation generators produce random or periodic control functions whose frequency, if any, has an order of magnitude of 10 Hz. A tone structure is composed of several such units. An Oscillator unit generates a waveform from a library of samples stored in the Read Only Memory. The waveform's "pitch" is controlled by the Pitch Modulation Generator and the Pitch Envelope Generator. The resulting signal passes through a Variable Digital Filter, a high-pass filter whose cutoff frequency's variations are controlled, like for pitch, by an Envelope Generator (EG) and a Modulation Generator (MG), etc. Other units in a tone structure include a Panoramics unit which manages the stereo routing of the signal (left to right). As already mentioned, a voice can decompose into one or two tones. A single-tone voice is roughly identical to a tone, while a double-tone voice corresponds to the grouping of two tones which share the same Pitch EG and VDF MG. The Effect Processor allows to use simultaneously two effect types picked up from a list of 47 possible types (delays, choruses, etc.). Examples of synthesis parameters stored in the 05R/W's memory are 'Tone 2 of Voice 7 VDF1 EG's Attack Level' (a number between 0 and 99) or 'Effect Processor's Second Effect's name' (a character string such as "Hall-reverb").

**Representation of the sounds in BACK**

We built up a representation of sounds using the formalism of Description Logics, by introducing two kinds of concepts: *fundamental* and *abstract* concepts. fundamental concepts are used to represent the various entities shown on Figure 2; abstract concepts represent the structural part of the programmer's expert knowledge. We will now examine these two concept categories in more detail.

*Fundamental concepts*

We represent the technical description of sounds (as previously introduced) by a set of Back terms. We call these terms "fundamental" because they are a quasi direct transcription of the synthesis model architecture of the synthesizer.

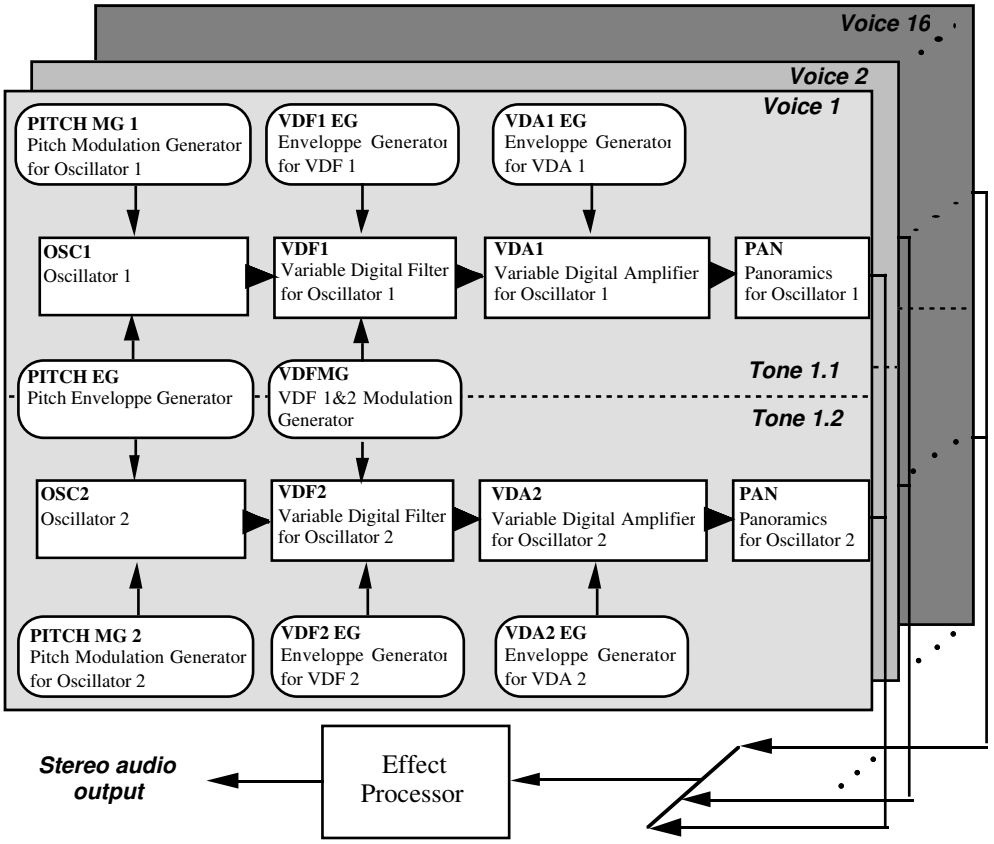


*Figure 2. Korg's sound representation: diagram of the 05R/W synthesis architecture.*

For example, the `tone` concept cannot be defined using less specific concepts other than a special preset concept called `anything`. Therefore, the `tone` concept is introduced using necessary classification conditions with respect to the classification of its instances, which is materialized by a primitive concept introduction, denoted by the symbol :< as follows:

```
tone :< anything
```

The `waveform` concept is *defined* (symbol :=) as its extensive introduction yields necessary and sufficient instance classification conditions. For obvious reasons, instead of listing all possible waveforms (over 300), we provide here an excerpt with just a few examples.

```
waveform := attribute_domain ([sine, square, saw, organ,
                fluteLoop, whiteNoise]).
```

The primitive role `hasWaveform` can then be introduced to represent which waveform a tone is based on. We call such a role *terminal* as its

fillers represent actual synthesis parameters as opposed to abstract structures like voices or envelopes.

```
hasWaveform :< domain(tone) and range(waveform).
```

Another defined concept is `doubleVoice`, which is introduced based on the primitive concept `voice`, with *number restrictions* applying to the role `hasTone`:

```
doubleVoice := voice and exactly(2, hasTone).
```

*Abstract concepts*

On top of this first layer of representation, we build up a hierarchy of concepts that represent the structural part of the programmer's expertise. As will be explained later, the concepts introduced here are used to define transformations. We divide these concepts into three categories:

**Abstract concepts built from fundamental concepts.** These include partial descriptions of sounds, such as `heldTimeFunction`, defined as a `TimeFunction` whose 'sustainLevel' value is greater or equal to 1. In synthesis terms, this allows to describe, for instance, sounds whose loudness eventually stabilizes to a non-zero value. On the 05R/W, this is obtained by setting the Time Variant Amplifier Envelope Generator's Sustain Level to a strictly positive value.

```
timeFunction    and the (sustainLevel, ge(1))
                => heldTimeFunction.
```

Similarly, we introduce abstract concepts that, as will be seen below, play a part in building up the brassy-able sound type's representation. Figure 3 shows the manner in which these interdependent, abstract concepts are introduced.

```
FilterEnveloppe   and heldTimeFunction
                  and  the(attackTime,ge(17)  and  le(25))
                  and the(attackLevel, ge(85))
                  and the(decayTime, ge(60) and le(75))
                  and the(intermediateLevel,ge(25) and le(35))
                  and  the(heldLevel,  ge(25)  and  le(35))
                  => brassyAbleFilterEnv.

filter            and the(hasEnveloppe,  brassyAbleFilterEnv)
                  => brassyAbleFilter.

brightTone        and  the(hasFilter,  brassyAbleFilter)
                  and the(hasAmp, brassyAbleAmp)
                  => brassyAbleTone.

voice             and atleast (1, hasTone,
                  brassyAbleTone)
```

```
                    => brassyAbleVoice.
```

*Figure 3. Example of code used for introducing abstract concepts*

Other examples of abstract concepts which reflect structural expert knowledge are those describing 'non transformable' sounds. Contrary to the above abstract concepts, these concepts provide *complete* descriptions of sounds which do not afford any particular transformation but which are used for describing transformable sounds:

```
sound               and no(hasVoice, voiceWithInharmonicTone)
                    => harmonicSound
```

Note that since any sound type is subsumed by the transformable sound type `soundInGeneral`, even instances of non transformable sounds types can undergo some general transformations, such as 'make bright' or 'make dull'**.**

**Concepts describing transformable sounds.** Figure 4 shows a few examples of such concepts.

```
harmonicSound       and some (hasVoice, brassyAbleVoice)
                    => brassyAbleSound

sound               and all(hasVoice, sustainAbleVoice)
                    => sustainAbleVoice.

sound               => soundInGeneral.
```

*Figure 4. Examples of transformable sounds.*

**Concepts representing transformations themselves.** To each sound type we associate a list of transformations, represented as mere character strings, as illustrated on figure 5. This list is materialized by a sub-concept of `possibleTransformations`, an extensional concept which lists all possible transformations for all existing sound types.

```
possibleTransformations:= attribute_domain ([warm,
brassy, dull, decaying (...)]).

affordsTransformation :< domain(sound) and range
(possibleTransformations).

brassyAbleSound :< affordsTransformation : brassy.

WarmAndSharpAbleSound :< affordsTransformation :
                  warm and sharp.
```

*Figure 5. The BACK representation of transformations.*

## The Object-Oriented representation of sounds

Representing sounds as objects, in the sense of object-oriented programming, is particularly natural in our context, where emphasis is put on transformations. Each transformation is represented by a Smalltalk method defined in class `CurrentSound`. This method modifies the values of "terminal parameters", using a pre-defined set of modifiers.
For instance, Figure 6 shows the Smalltalk method that makes a sound "sustained":

```
beSustained
     tones do: [:t | t be: #held].
```

Where be: is defined as follows:

```
be:  aSymbol
     aSymbol = #held ifTrue:

          [filterEnvelope decayTime: 99.

          ampEnvelope decayTime: 99].
     aSymbol = #...  ifTrue: [...].
```

*Figure 6. A Smalltalk method that makes a sound "sustained".*

## Integration

In order to link these two representations, we need some kind of integration scheme. This scheme is based on two principles, related respectively to concepts and transformations:

**Concepts**

Each fundamental concept is represented by a Smalltalk class. Each role of the concept is represented by an instance variable of the class. Of course, the Smalltalk representation is more rudimentary than the BACK one: cardinality and types are not represented. An actual sound is represented by an instance of class `CurrentSound`, and its parameters by instances of the corresponding classes.

**Transformations**

Transformations are represented by methods associated to class `CurrentSound`. Each method modifies the current sound by changing some values of its parameters. Therefore, the semantics of the BACK symbol representing transformations is given by the corresponding Smalltalk method.

This yields a two-level representation framework with two links, as shown in Figure 7.

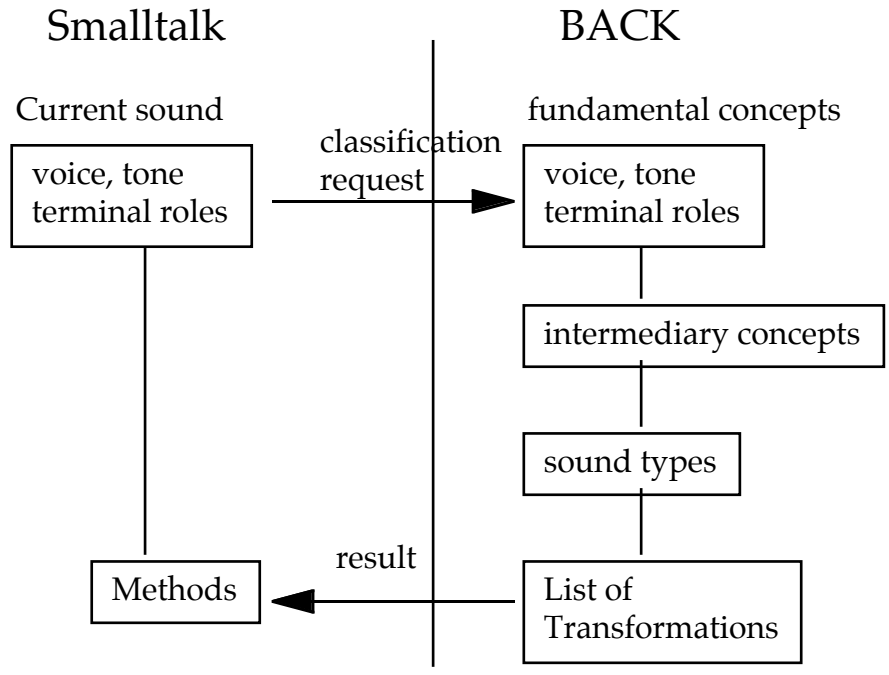


*Figure 7. The two representations of sounds and their connection.*

## Execution

A session with our system is described by the following iterative cycle of operations :

Step 1. The user chooses an initial patch. This initial patch may be either one of the user's own patches or one selected from an external library, including patches created during past sessions.
Step 2 The patch is transmitted to BACK to be classified. As a result, BACK generates the list of sound types instantiated, together with the associated transformations. This data is then transmitted to Smalltalk.
Step 3. The user selects one of the proposed transformations. Parameters in the `currentSound` Smalltalk object are modified accordingly, and actual synthesizer parameters are changed so the user can play and listen to the new sound.
Step 4. Back to step 2, and loop until the user is satisfied with the current sound.
Figure 8 shows an example of a typical user session. In order to provide some flexibility, our system offers an alternative to step 3 in which the user directly changes individual synthesizer parameters. This still allows for the resulting sound to get classified and thus undergo further transformation cycles.

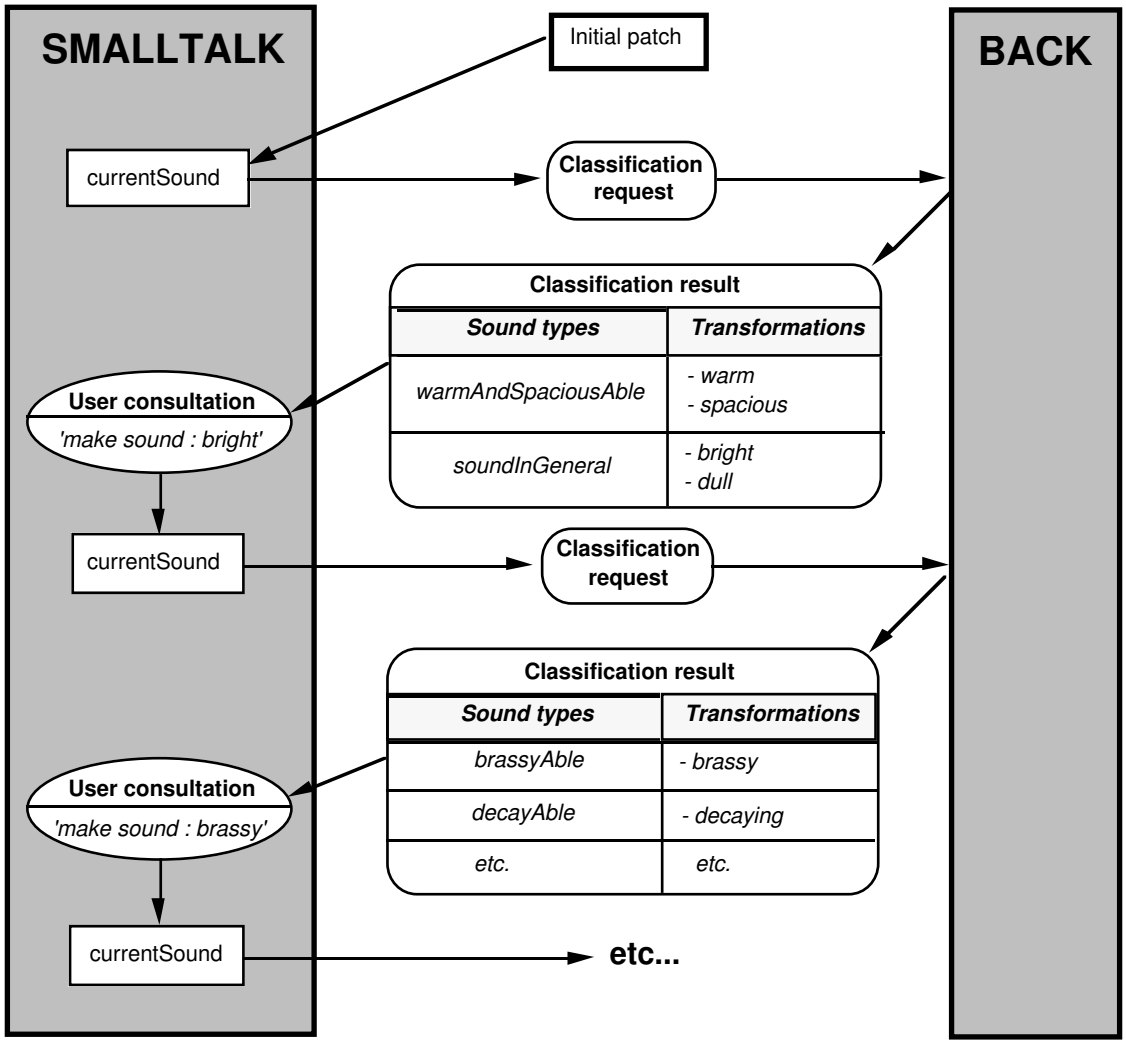

*Figure 8 An example session.*

## Defining a new sound type

In the current state of our prototype, introducing a new sound type in the classification is not an easy task. It implies three modifications to the system:
First, adding a set of terms that describe the new sound type, as a function of existing sound types and fundamental concepts.
For instance, adding sound type `choralAndBreathyAble` could lead to the following introductions (see Figure 9):

```
padWave    := aset([organ, fluteLoop, sine,
           square], waveForm).

tone       and sustainedTone
           and the(hasWaveform, padWave)
           => padTone.

(...)

softSound  and padSound
           and atLeast(1, hasVoice, doubleVoice)
           => choralAndBreathyAbleSound
```

*Figure 9. Instructions for adding sound type 'choralAndBreathyAble'*

Second, specifying which transformations the new sound type affords, at the BACK level:

```
choralAndBreathyAbleSound :< affordsTransformation :
                        choral and breathy.
```

Third, specifying, at the Smalltalk level, the transformations themselves, by defining instance methods for class `CurrentSound` (see example on Figure 10).

```
beChoral
     self voices detuneOfPercentHalfTone: 5; beSymetric.
     self setVibratosOfKind: #Choral.
```

where `beSymetric, detuneOfPercentHalfTone:` and `setVibratosOfKind:` are Smalltalk methods such as the one below:

```
detuneOfPercentHalfTone: aPercentage
     self firstTone pitchFine: (0.5 * aPercentage) negated.
     self secondTone pitchFine: (0.5 * aPercentage).
```

*Figure 10. A Smalltalk method specifying a transformation*

## Conclusion

The main contribution of this work concerns the use of a sophisticated AI representation formalism, Description Logics, to capture superficial knowledge about synthesizer patch programming. This knowledge can then be exploited to help musicians browse through the timbre space of a commercial synthesizer in an intuitive way, thereby reducing the complexity of the sound making process.
A prototype was built and tested with the Korg 05R/W synthesizer. This work is still in progress, and our efforts concentrate on: 1) refining the user interface, 2) improving the communication between BACK and Smalltalk (using DLL), and 3) experiment our system with novice users. We also plan to provide the system with a learning capability (e.g. based on inductive learning from examples), which will allow the definition of new sound types by presenting sets of example patches to the system. Further, transformations will be definable from sets of patch couples, each containing an example of origin sound and its associated transformed sound.

## References


Brachman, R.J., and J.G. Schmolze, 1985. "An overview of the KL-ONE knowledge representation system." *Cognitive Science* 9(2): 171-216.

Chowning, J. and D. Bristow. 1986. *FM Theory & applications by musicians for musicians*. Yamaha Music Foundation Corp.

Goldberg, A., and D. Robson. 1983. "Smalltalk-80 : The language and its implementation." Reading, MA: Addison-Wesley.

Grey, J. 1975. *An Exploration of Musical Timbre*. Ph.D. dissertation, Stanford University Psychology Dept. CCRMA Report STAN-M-2.

Hebel, K, 1989. "Javelina: An Environment for Digital Signal Processing Software Development." *Computer Music Journal* 13(2) Summer 1989.

Reprinted in S. Pope (Ed.) *The Well-Tempered Object*, Cambridge, MA: MIT Press, 1991.

Heinsohn, J. Kudenko, D. Nebel, B. Profitlich, H.-J. 1994. "An empirical analysis of terminological representation systems." *Artificial Intelligence* 2: 367-397. Germany: Elsevier.

Hoppe, T. C, Kindermann, J. Quantz, A. Schmiedel, M. Fischer. 1993. *Back V5 Tutorial & Manual*, Institut fûr Software und theoretische Informatik, W-1000 Berlin 10, Germany, march 1993.

KORG. Undated. Korg 05R/W *Manuel d'utilisation, AI2 Synthesis Module*. Korg Inc.

Michalski, R. S. 1983. "A theory and methodology of inductive learning." in Michalski, R.S., J.G. Carbonell and T.M. Mitchell, (eds.), *Machine Learning: an Artificial Intelligence approach*. Palo Alto, California: TIOGA Publ. Co. pp. 83-134.

Miranda, E. 1992. *From symbols to sounds: an AI-based investigation of Sound Synthesis (Ph.D. Thesis Proposal).* DAI Discussion Paper, No. 117, Dept. of Artificial Intelligence, University of Edinburgh.

Openheim, D.V. 1989. "DMIX: An Environment for Composition." 1989 *International Computer Music Conference*. San Francisco: Computer Music Association.

Openheim, D. V. 1991. "Shadow: An Object-Oriented Performance System for the DMIX Environment." Proceedings of the 1991 *International Computer Music Conference*, Montréal. San Francisco: Computer Music Association, pp. 281-284.

Risset, J.C. 1969. "An introductory catalogue of computer-synthesized sounds". Bell Telephone Labs.

Rodet, X., Potard, Y., Barrière, J.B. 1984. "The CHANT project : from the synthesis of the singing voice to synthesis in general." In C. Roads (ed.), *The Music Machine*, Cambridge, MA: MIT Press.

Scaletti, C. 1989. "The Kyma/Platypus Computer Music Workstation." *Computer Music Journal* 13(2): 23-38.

Serra, X. 1989. *A system for sound analysis/transformation/synthesis based on a deterministic plus stochastic decomposition.* Ph.D. Thesis, Stanford University.

Vertegaal, R., and E. Bonis. 1994. "ISEE : an intuitive sound editing environment." *Computer Music Journal* 18(2) 21-29.

Wessel, D. 1979. "Timbre Space as a Musical Control Structure." *Computer Music Journal* 3(2): 45-52.

Yelland, P. 1992. "Experimental Classification Facilities for Smalltalk." in *Proceedings of OOPSLA' 92*, Vancouver, Canada, pp. 235-246.

# REPRÉSENTATION DE CONNAISSANCES SUR LA PROGRAMMATION DE SYNTHÉTISEURS

P.Y. Rolland, F. Pachet, LAFORIA-IBP, Université Paris 6

E-mail: rolland/pachet@laforia.ibp.fr

## Résumé

Le problème de l'interface homme-machine est aujourd'hui critique dans le domaine des synthétiseurs du commerce. Nous nous attachons à capturer la couche superficielle de connaissances—le savoir-faire des "experts en programmation de sons (ou patches)". Nous proposons un cadre conceptuel et fonctionnel pour représenter ce savoir superficiel dans le but de fournir une aide aux musiciens non-experts : leur permettre de créer des patches de synthétiseur du commerce de manière plus intuitive qu'avec les interfaces actuelles. L'idée centrale de notre architecture est celle d'une classification des sons en fonctions des transformations que l'expert sait leur appliquer. Notre cadre conceptuel est fondé sur une représentation des sons à deux niveaux, combinant programmation par objets (Smalltalk) et mécanismes de classification (BACK). Nous proposons un schéma d'intégration des deux paradigmes, et illustrons notre approche par un système prototype d'aide à la programmation pour synthétiseurs de la famille Korg 0xx/W.

## 1. De la synthèse assistée à la programmation assistée

### Métaphore de l'unité de soins intensifs

La remarque suivante est à l'origine de notre approche. Dans une unité de soins intensifs, une infirmière sait en général parfaitement se servir d'un respirateur artificiel. Néanmoins celle-ci peut être "experte" en matière de manipulation de ces appareils sans pour autant posséder une connaissance théorique de leur mode d'action, ou des aspects médicaux liés au diagnostic, au traitement etc. Il se peut qu'elle soit plus experte que le médecin; mais elle possède uniquement le savoir dont elle a besoin, un savoir superficiel lui indiquant comment utiliser l'appareil avec efficacité et sécurité. Par analogie, nous comparons d'un côté les experts en *programmation de patches* qui savent programmer efficacement un synthétiseur, de l'autre les experts en *synthèse sonore* pour qui les transformées de Fourier et le traitement de signal n'ont aucun secret mais qui ne savent pas comment, par exemple, rendre un son "plus chaud" sur un synthétiseur "hardware" du commerce tel que Yamaha W-7, E-Mu Proteus ou Korg 05R/W.

Nous visons à représenter les connaissances manipulées par la première catégorie d'experts, en affirmant qu'une telle représentation est possible et fournit le moyen d'enrichir les interfaces actuelles de programmation de synthétiseurs.

### Paradigme de la Synthèse Assistée par Ordinateur (S.A.O.)

L'observation qui précède est à la base de notre approche en matière de programmation de synthétiseurs assistée par ordinateur (P.S.A.O.). De notre point de vue, les objectifs principaux d'un système de S.A.O. sont 1) d'aider le musicien à créer des sons, et 2) de l'aider à comprendre une technique de synthèse. Un certain nombre de travaux ont été consacrés à la conception de tels systèmes. Ceux ci se fondent sur des paradigmes variés, allant de la *synthèse par règles* implémentée par le système CHANT (Rodet et al. 1984) à *l'assistance adaptative à la synthèse* (ARTIST, de Miranda (1992)). Parmi les systèmes existants on peut également citer L'Intuitive Sound Editing Environment de Vertegaal & Bonis (1994), Kyma/Platypus (Scaletti 89), Javelina (Hebel 89), DMIX (Openheim 89) etc.

### Conséquences

L'étude des système de SAO existants montre que :

1) la majorité des travaux ont concerné, non pas des synthétiseurs du commerce, mais des synthétiseurs informatiques sophistiqués qu'utilisent très peu de musiciens.

2) La plupart des systèmes proposent des environnements dans lesquels le musicien définit et manipule des objets et structures sonores. A notre connaissance, aucun de ces systèmes ne se propose de représenter directement les connaissances pratiques des programmeurs de patches experts.

C'est sur ces constatations que s'appuie notre choix d'aborder la *PSAO*, plutôt que la PAO proprement dite. Notre approche se fonde sur deux hypothèses:.

## 2. Hypothèses

**H1. Les programmeurs de patches experts font appel à des connaissances superficielles**

Nous faisons l'hypothèse que la programmation de synthétiseurs du commerce nécessite beaucoup de savoir faire et peu de connaissances "fondamentales". Le cas de la synthèse FM est éloquent à cet égard. L'histoire à succès de la famille de synthétiseur DX montre que bien des patches "fameux" ont été créés par des programmeurs possédant un savoir "superficiel" acquis grâce à l'expérience et à l'observation de patches programmés par leurs pairs. Pour se convaincre du rôle limité joué par la compréhension des aspects théoriques complexes, on se référera aussi aux tentatives faites pour répandre la théorie FM (Chowning et Bristow 1986) où l'accent est clairement mis sur les aspects pratiques. La "règle de savoir-faire" (R) ci-dessous est typique de la programmation FM, et peut être utilisée par un programmeur sans que celui-ci en comprenne le fondement théorique.

(R) On peut obtenir un son "tremolo" (modulation d'amplitude) en fixant la fréquence porteuse à une valeur basse et en réglant la modulante à une valeur variable en fonction de la note jouée.

**H2. Le "savoir expert" concerne surtout les transformations sur les sons**

Il est bien connu que les programmeurs de patches experts préfèrent, pour obtenir un son donné, partir d'un patch proche de l'objectif et le *transformer* plutôt que de "partir de zéro". Ainsi, considérant que la majorité de l'expertise concerne les transformations, nous nous attachons à représenter les connaissances liées plutôt aux transformations qu'aux sons eux-mêmes. Nous présentons ici deux exemples typiques des *règles transformationnelles* que nous souhaitons prendre en compte dans notre système. La première est applicable à tout synthétiseur possédant un filtre passe-bas paramétrique. La seconde règle est elle aussi applicable à une majorité de synthétiseurs du commerce, mais n'est valide que pour des sons pseudo-harmoniques, c'est à dire dont on perçoit la hauteur (exemple: son de piano, par opposition à un son de cymbale) :

(R1) On sait rendre un son *plus brillant* en augmentant la fréquence de coupure du filtre.

(R2) On sait rendre un son *plus chaud* en dupliquant le son et en imposant un *léger désaccord* entre le son initial et sa copie.

## 3. Règles transformationnelles et classification des sons

**Organisation des connaissances**

Le genre de règles transformationnelles décrites ci-dessus se représentent aisément par l'intermédiaire d'un système à base de connaissances, c'est à dire par des règles de production. Bien sûr comme nous l'avons vu toutes les transformations ne s'appliquent pas à tous les types de sons. Plus précisément les transformations s'appliquent à des types de sons "origines" et donnent lieu à des types de sons "cibles". Ces derniers sont caractérisés par de simples adjectifs ("brillant, "chaud", etc.). Afin d'organiser ces types de sons, il faut faire appel à un mécanisme de classification . Dégager une classification concernant les types de sons "cibles", basée par exemple sur les résultats de la psychoacoustique (Wessel 1979, Grey 1975) serait inadéquat. En effet un son donné doit être classifié en fonction des *transformations qu'il peut subir* et non de celles qui ont permis de l'obtenir: par exemple, nous ne cherchons pas à ranger les sons "cuivrés" au sein d'une quelconque typologie; au contraire, nous visons à identifier les "sons que l'on sait rendre cuivrés" grâce à une transformation connue.

Dans cette conception, être instance d'un type de sons signifie simplement être capable de subir les différentes transformations associées à ce type de sons, ainsi les noms des types de sons n'ont pas de sens direct. Pour des raisons pratiques, nous donnons aux types de sons un nom arbitraire construit sur le préfixe 'pré', par exemple préCuivré ou préSourd. Dans le cas de types de sons pouvant subir plusieurs transformations spécifiques, nous utilisons un nom composé du type préBrillant-Cuivré. Chaque transformation est associée à un type de sons particulier : par exemple, la transformation (R2) est associée au type de sons préChaud, la transformation (R1) au type préBrillant-Cuivré.

En résumé, la tâche principale du système est de classifier les sons selon une classification pré-définie. La figure 1 présente une partie de la hiérarchie de types de sons pour le Korg 05R/W. Les types de sons sont matérialisés par des rectangles en dessous desquels sont indiqués les transformations associées. Les flèches traduisent des liens "type/sous-type".

**Paradigme de représentation**

Compte tenu de la nature des connaissances à représenter concernant la classification nous avons intégré, au sein de notre cadre de représentation, à la fois 1) des mécanismes de classification afin de manipuler les hiérarchies de sons, et 2) un langage procédural permettant d'exprimer les transformations appliquées au son courant. Des tentatives ont été faites (Yelland 1992) pour intégrer la classification automatique dans les langage orientés objets (LOO) mais aucun système intégré n'est directement disponible. Compte tenu du compromis expressivité / efficacité / complétude auquel sont soumis les systèmes de classification existants (Heinson et al. 94), nous avons opté pour le système BACK (Hoppe et al. 1993), une implémentation du formalisme des logiques descriptives — cf. section 4.

***Une logique descriptive pour classifier les sons***

Les sons que nous modélisons ont cette particularité qu'ils peuvent se décrire par des symboles au lieu de simples tableaux de valeurs de paramètres. Par exemple un son du Korg 05R/W se compose d'une à seize parties ("parts"), chacune composée d'une à deux voix ("voices"), etc. En bout de chaîne, ces structures abstraites sont décrites en termes de valeurs *terminales* de paramètres ('temps d'attaque', 'nom de la forme d'onde' etc.). Cette forte structuration est bien adaptée aux techniques de *classification de termes* dans lesquelles l'utilisateur établit "déclarativement" les liens structuraux et le système se charge des tâches d'inférence (classification et subsomption).

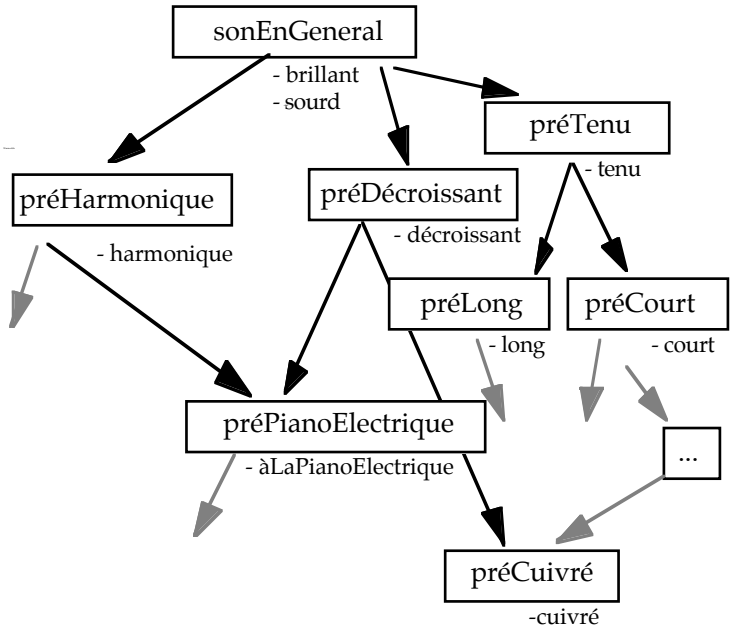


*Figure 1. Hiérarchie de types de sons partielle pour le synthétiseur Korg 05R/W*

***Un LOO pour représenter les transformations***

Les transformations, pour leur part, doivent faire l'objet d'une représentation effective et pas uniquement abstraite, ce que BACK ne permet pas directement. De plus, la programmation de l'interface utilisateur et de la communication MIDI est bien plus aisée avec un LOO (en l'occurrence Smalltalk-80). Les sons sont représentés comme *instances* de classes de sons, les transformations comme *méthodes* pour ces classes.

3) Un mécanisme de couplage entre les deux représentations

Faire coexister deux paradigmes de représentation n'est pas trivial, en particulier en raison de problèmes de cohérence : les instances d'un même type de sons sont de nature complètement différente suivant que celui-ci est représenté comme une classe (LOO) ou un concept (LD). La section 6 décrit précisément le mécanisme de couplage que nous proposons.

## 4. La classification automatique des sons

### Les logiques de description (LD) et BACK

Issues de travaux de recherche initiés vers 1980, les LD sont des formalismes de représentation proposant une organisation hiérarchisée des connaissances où tout *concept* hérite les propriétés des concepts *plus généraux* que lui et entretient avec les autres concepts des liens binaires appelés *rôles*. Les nombreuses implémentations existantes (Heinsohn et al. 1994), dérivées du précurseur KL-ONE (Brachman 1985), offrent différents "services" d'inférence comme les tests de subsomption et les requêtes de classification.

Nous avons utilisé l'implémentation BACK (Berlin Advanced Computational Knowledge, version 5 basée sur Prolog) que nous décrivons ici dans ces grandes lignes. Les concepts sont des ensembles d'*objets* introduits intensionnellement ou extensionnellement, et les objets sont instances d'un ou plusieurs concepts. Les concepts peuvent être primitifs ou définis, selon qu'ils donnent lieu à des conditions nécessaires (resp. nécessaires et suffisantes) de classification pour leurs instances. Le *contenu d'un rôle* (*role-filler*) correspond au second membre de la relation binaire correspondant à ce rôle. Toutes ces notions quelque peu abstraites seront illustrées par divers exemples ci-après.

L'information est introduite dans BACK à travers la création de *termes* (concepts ou rôles) et d'objets, ou à travers l'usage de *règles non définitionnelles* imposant des relations logiques particulières entre les concepts. Les services d'inférences comprennent la *classification* qui fournit la liste des concepts instanciés par un quelconque objet, et le *test de subsomption* qui indique si un concept c1 est plus spécifique que c2, c'est à dire si toute instance de c1 est instance de c2.

### Description "technique" des sons

Dans la mémoire du Korg 05R/W, un son est représenté par une liste de valeurs de paramètres de synthèse. La *description technique* des sons correspond à cette représentation de bas niveau. La figure 2 montre l'architecture de la synthèse telle que la présente Korg. On retrouve les *parties* et les *voix* introduites plus haut. Sur le schéma de la figure 2, les cases rectangulaires symbolisent un premier type d'unités chargées de générer ou de traiter le signal audionumérique (Oscillator, Variable Digital Filter ou VDF, processeur d'effets, etc.). Les cases arrondies symbolisent, elles, un second type d'unités : des générateurs d'enveloppe ou de modulation contrôlant l'évolution temporelle des paramètres des unités de type 1 (exemple: pitch enveloppe generator, VDF modulation generator). Plusieurs unités sont associées au sein d'une structure de type 'voix'. En ce qui concerne les parties, nous avons dit que celles-ci regroupent chacune une ou deux voix. Les deux voix d'une partie à deux voix ont certaines de leurs unités en commun (pitch enveloppe generator, VDF modulation generator). En revanche, une partie ne comportant qu'une voix se résume en gros à cette voix.

### Représentation des sons dans BACK

La représentation BACK traduisant le diagramme de la figure 2 se fait par l'introduction de deux types de concepts: concepts fondamentaux et concepts abstraits.

#### *Concepts fondamentaux*

Les *termes* constituant la représentation 'technique' des sons seront appelés "fondamentaux" car ils sont une transcription presque directe de l'architecture du modèle de synthèse. Par exemple, le concept **voix** ne peut être introduit à partir de concepts plus généraux que lui, autres que le concept prédéfini **anything.**Les conditions de classification d'objets associées à **voix** étant seulement nécessaires, nous l'introduisons comme un concept *primitif* (d'où le symbole BACK :<).

```
voix :< anything.
```

Le concept forme_dOnde est *défini* (symbole :=), car son introduction en extension se traduit par des conditions *nécessaires et suffisantes* de classification d'objets. Pour des raisons évidentes, nous avons considérablement écourté ici la liste des 300 formes d'ondes stockées dans la mémoire du 05R/W.

```
forme_dOnde := attribute_domain([sinus, carré, triangle,
bruitBlanc]).
```

Le rôle primitif **aForme_dOnde** peut alors être introduit pour spécifier la forme d'onde sur laquelle est basée une voix. Nous qualifions un tel rôle de *terminal*, signifiant que son contenu est directement lié à des valeurs de paramètres (ici par exemple: 'sinus', 'carré' etc.) et non à des structures abstraites, comme c'est le cas pour le rôle **aVoix** utilisé plus bas.

```
aForme_dOnde :< domain(voix) and range(forme_dOnde).
```

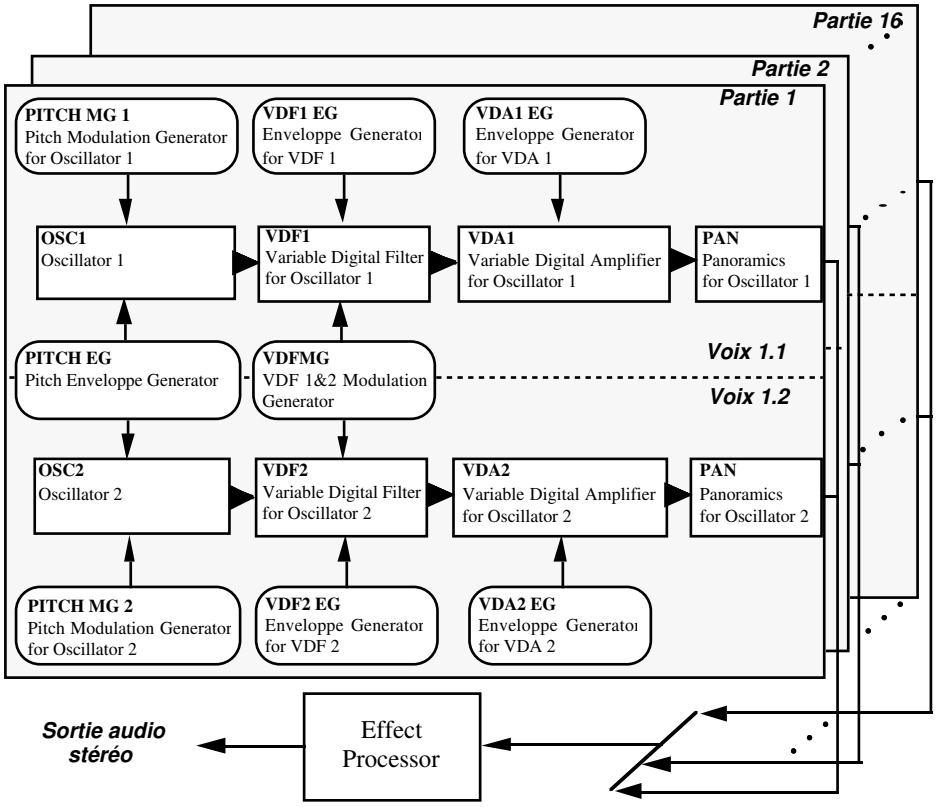


*Figure 2. Représentation "technique" des sons : schéma de synthèse du 05R/W*

Un autre concept défini est **partieBiVoix**, dérivé du concept primitif **partie** par une restriction sur la cardinalité du rôle **aVoix** :

```
partieBiVoix := partie and exactly(2, aVoix).
```

***Concepts abstraits***

Au dessus de cette première couche de représentation, nous construisons une hiérarchie de concepts représentant la partie "structurale" de l'expertise. Comme on le verra plus loin, la définition des transformations fait intervenir ces concepts . Nous répartissons ces derniers en trois catégories :

i) des concepts abstraits construits à partir de concepts fondamentaux

Parmi ces concepts, des descriptions partielles de sons comme enveloppeTenue, définie comme une enveloppe dont la valeur du paramètre 'niveauDeSustain' est non nulle. En termes de synthèse, cela permet de décrire des sons dont, par exemple l'amplitude, se stabilise à une valeur non nulle (oscillation entretenue).

```
enveloppe                    and the(niveauDeSustain, ge(1))
                             => enveloppeTenue.
```

De façon similaire sont introduits les concepts abstraits jouant un rôle dans la tâche de représentation du type de sons préCuivré (voir plus bas). Par souci de clarté, dans les exemples qui suivent nous utilisons des valeurs numériques explicites (par exemple dans 'the(tempsAttaque, ge(17) and le(25))') tandis que dans notre implémentation nous avons recours à des constantes nommées comme 'tempsAttaqueMinPourEnveloppeFiltreCuivrée"

```
enveloppeFiltre                 and the(tempsAttaque, ge(17)
                                and le(25))
                                and the(niveauAttaque, ge(85))
                                and the(tempsDecroi, ge(60) and
le(75))
                                and the(niveauInterm, ge(25)
                                and le(35))
                                and the(niveauTenu, ge(25)
                                and le(35))
                               => envFiltrePréCuivrée.

filtreVar                      and the(aEnveloppe,
                               envFiltrePréCuivree)
                               => filtreVarPréCuivré.

voixBrillante                  and the(aFiltre, filtreVarPréCuivré)
                               and the(aAmpli, ampliPréCuivré)
                               => voixPréCuivrée.

partie                         and atleast (1, aVoix, voixPréCuivrée)
                               =>partiePréCuivrée.
```

D'autres exemples de concepts abstraits reflétant les connaissances expertes structurelles sont ceux décrivant des sons "non transformables". Contrairement aux précédents, ces concepts abstraits fournissent des descriptions complètes de sons pour lesquels on ne connaît pas de transformation particulière mais qui servent à décrire des sons transformables :

```
son                            and no(aPartie,
partieAvecVoixInharmonique)
                               => SonHarmonique
```

Il est à noter que dans la mesure où tout son est subsumé par le type de son transformable 'sonEnGénéral', même les instances de sons non transformables sont passibles de transformations comme 'rendre brillant' ou 'rendre sourd'.

ii) des concepts décrivant des sons transformables

Voici quelques exemples de tels concepts :

```
SonHarmonique                  and some (aPartie, partiePréCuivrée)
                               => sonPréCuivré

son                            and all(aPartie, partiePréTenue)
                               => sonPréTenu.

son                            => sonEnGénéral.
```

iii) des concepts représentant les transformations elles-mêmes :

A chaque type de son nous associons une liste de transformations représentées comme des simples chaînes de caractères. Cette liste est matérialisée par un sous-concept de **transformationsPossibles**, lequel liste l'ensemble des transformations pour l'ensemble des types de sons

```
transformationsPossibles:= attribute_domain ([chaud, cuivré, sourd,
décroissant (...)]).

supporteTransformation :< domain(son) and range
(transformationsPossibles).

sonPréCuivré :< supporteTransformation : cuivré.

sonPréBrillant-Cuivré :< supporteTransformation : brillant and
cuivré.
```

## 5. Représentation des sons par objets

Une telle représentation des sons est tout à fait naturelle dans notre contexte où l'accent est mis sur les transformations. Chaque transformation est représentée par une méthode Smalltalk définie dans la classe **sonCourant**. Cette méthode modifie la valeur des paramètres 'terminaux' par le biais d'un ensemble pré-défini de "modifieurs". Voici par exemple la méthode rendant un son "tenu".

```
soisTenu
     voix do: [:v | v sois: #tenue].
     ...
```

où **sois:** est définie comme suit

```
sois: unSymbole
     unSymbole = #tenue ifTrue:
     [enveloppeFiltre tempsDécroissance: 99.
     enveloppeAmpli tempsDécroissance: 99].
     unSymbole = #...  ifTrue: [...]
```

## 6. Intégration

Le schéma d'intégration est fondé sur deux principes :

### Concepts

Chaque concept fondamental est représenté par une classe Smalltalk. Chacun des rôles du concept est représenté par une variable d'instance de la classe correspondante. Bien entendu, la représentation en Smalltalk est rudimentaire comparée à son analogue dans BACK, puisque les types et les cardinalités ne sont pas pris en charge. Les sons eux-mêmes sont représentés par des instances de la classe **sonCourant**, leurs paramètres par des instances des classes correspondantes.

### Transformations

Celles-ci sont représentées par des méthodes associées à la classe **sonCourant.** Chaque méthode modifie le son courant en changeant certaines de ses valeurs de paramètres. Ainsi, la sémantique du symbole BACK représentant les transformations est fournie par la méthode Smalltalk correspondante. La figure 3 illustre le cadre de représentation à deux niveaux issu de ce schéma d'intégration.

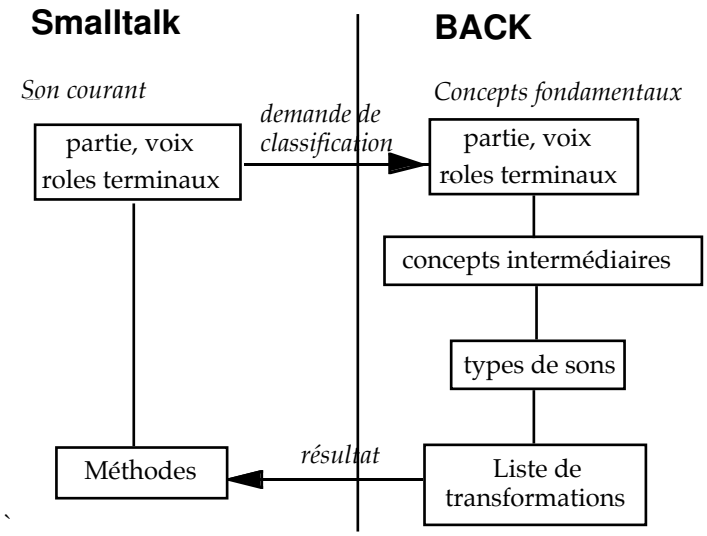


*Figure 3. Les deux représentations des sons et leur connexion*

## 7. Exécution

Une séance de travail avec notre système peut être vu comme un cycle itératif d'opérations :

1. Tout d'abord, l'utilisateur choisit un patch initial.
2. Le patch est transmis à BACK en vue d'être classifié. BACK rend la liste des types de sons instanciés, avec la liste des transformations associées. Ces données sont transmises à Smalltalk.
3. L'utilisateur choisit une des transformations proposées. Les paramètres de l'objet Smalltalk sonCourant, ainsi que ceux du synthétiseur réel sont modifiés en conséquence, et l'utilisateur a la possibilité d'écouter et de jouer de ce nouveau son.
4. Retour à l'étape 2.

Le figure 4 schématise une séance utilisateur typique.

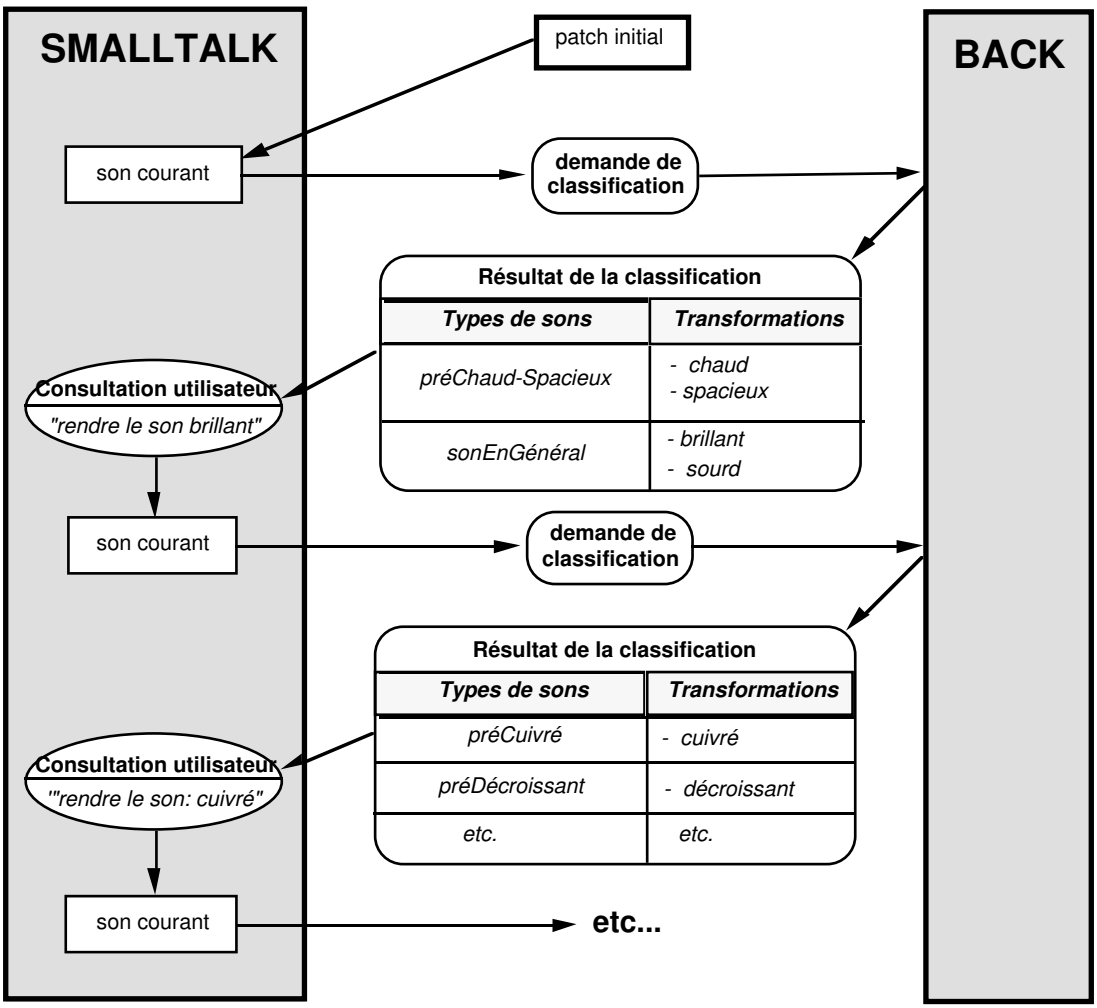


*Figure 4. Exemple de séance utilisateur.*

## 8. Ajout d'un nouveau type de sons

Au stade actuel de notre prototype, l'ajout d'un nouveau type de sons n'est pas une tâche aisée. Cela exige d'apporter trois modifications au système :

1. Ajouter un ensemble de termes décrivant le nouveau type de sons en fonction des types existants et des concepts fondamentaux. Pour illustrer notre propos, supposons que l'on souhaite ajouter le type de sons préChoral-Bruité. On serait amené à introduire les termes et règles suivants :

```
ondePourNappe                 := aset([orgue, fluteBoucle, sinus,
square],                      forme_dOnde).

voix                          and voixTenue
                              and the(aForme_dOnde, ondePourNappe)
                              => voixPourNappe.

(...)

sonDoux                       and sonNappé
                              and atLeast(1, aPartie, partieBiVoix)
                              => sonPréChoral-Bruité
```

2. Spécifier, dans BACK, quelles transformations spécifiques sont applicables au nouveau type de sons.

```
sonPréChoral-Bruité            => supporteTransformation:
                               choral and bruité.
```

3. Spécifier les transformations elles-mêmes côté Smalltalk. Par exemple :

```
soisChoral
     self parties désaccordEnDemiTons: 0.05; soisSymétrique
     self adopteVibratoDeType: #Choral.
```

où **désaccordEnDemiTons**:, **soisSymétrique** et **adopteVibratoDeType**: sont des méthodes soit définies précédemment soit rajoutées pour l'occasion, par exemple :

```
désaccordEnDemiTons: unNombre
     self voix1 accordFin: (0.5 * unNombre) negated.
     self voix2 accordFin: (0.5 * unNombre).
```

Dans une version future il est prévu de doter le système d'une capacité d'apprentissage (basée probablement sur l'apprentissage inductif à partir d'exemples), ce qui autorisera la définition de nouveaux types de sons par l'intermédiaire de patches représentatifs simplement présentés au système. De plus chaque transformation pourra être définie à partir d'un ensemble de paires de patches, chacune regroupant un exemple de son d'origine et ce même son après transformation.

## 9. Conclusion

La contribution principale de notre travail de recherche concerne l'implémentation d'un formalisme de représentation sophistiqué, celui des logiques descriptives, en vue de capturer les connaissances superficielles se rapportant à la programmation de synthétiseurs. L'exploitation de ces connaissances permet de proposer au musicien de naviguer dans l'espace des timbres d'un synthétiseur du commerce de façon intuitive, réduisant ainsi la complexité de l'acte de création de sons.

Nous avons conçu et testé sur le synthétiseur Korg 05R/W un prototype. Ce travail est en cours, et nos efforts se concentrent actuellement sur : 1) le raffinement de l'interface utilisateur, 2) l'amélioration de la communication entre BACK et Smalltalk, et 3) l'expérimentation avec des utilisateurs novices.

## Références


Brachman, R.J., and Schmolze, J.G., 1985. An overview of the KL-ONE knowledge representation system. Cognitive Science, 9 (2). p. 171-216.

Chowning, J. Bristow, D. 1986. FM Theory & applications by musicians for musicians. Yamaha Music foundation, cop. 1986.

Goldberg, A., Robson, D. 1983. Smalltalk-80 : The langage and its implementation. Addison Wesley.

Grey, J. 1975. An Exploration of Musical Timbre. PhD dissertation, Stanford University Psychology Dept. CCRMA Report STAN-M-2.

Hebel, K, 1989. Javelina: An Environment for Digital Signal Processing Software Development. Computer Music Journal, Vol. 13, N. 2, Summer 1989. Réimprimé dans the Well-Tempered Object, S. Pope Ed. MIT Press, 1991.

Heinsohn, J. Kudenko, D. Nebel, B. Profitlich, H-J. 1994. *"A*n empirical analysis of terminological representation systems" , *Artificial Intelligence 68 (August 1994),* Elsevier, Allemagne, vol.2, p. 367-397.

Hoppe, T. Kindermann, C. Quantz, J. Schmiedel, A. Fischer, M. 1993. Back V5 *Tutorial&Manual*, Institut fûr Software und theoretische Informatik, W-1000 Berlin 10, Germany, march 1993.

KORG. Non daté. Korg 05R/W *Manuel d'utilisation*, AI$^2$ SYNTHESIS MODULE. Korg Inc.

Michalski, R. S. 1983. A theory and methodology of inductive learning. in Machine Learning: an Artificial Intelligence approach. Michalski, R.S., Carbonell, J.G. and Mitchell, T.M., eds. Palo Alto: TIOGA Publ. Co. California. pp. 83-134.

Miranda, E. 1992. From symbols to sounds : An AI-based investigation of Sound Synthesis (PhD Thesis Proposal). DAI Discussion Paper No. 117, Dept. of AI, University of Edinburgh.

Openheim, D. V. 1989. DMIX: An Environment for Composition. Actes de ICMC'89, Columbus, Ohio. International Computer Music Association.

Openheim, D. V. 1991. Shadow: An Object-Oriented Performance System for the DMIX Environment. Actes de ICMC'91, Montréal, Canada, pp. 281-284.

Rodet, X., Potard, Y., Barrière, J.B. 1984. The CHANT project : from the synthesis of the singing voice to synthesis in general. In The Music Machine, C. Roads (ed.), MIT Press.

Royer, V. Volle, P. 1994. "Introduction aux Logiques de Description", *Tutoriel RFIA'94*, 11 Janvier 1994.

Scaletti, C. 1989. The Kyma/Platypus Computer Music Workstation, CMJ, 13:2, Summer 1989. Réimprimé dans the Well-Tempered Object, S. Pope Ed. MIT Press, 1991.

Smith J.O. 1992. Physical Modeling Using Digital Waveguides. Computer Music Journal. 16:4, pp. 74-87. MIT Press.

Vertegaal, R., Bonis, E. 1994. ISEE : an intuitive sound editing environment. Computer Music Journal 18:2. MIT Press

Wessel, D. 1979. Timbre Space as a Musical Control Structure. Computer Music Journal. 3:2. MIT Press

Yelland, 1992. Experimental Classification Facilities for Smalltalk. Actes de OOPSLA' 92, Vancouver. pp. 235- 246.