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: roland/pachet@laforia.ibp.fr

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.

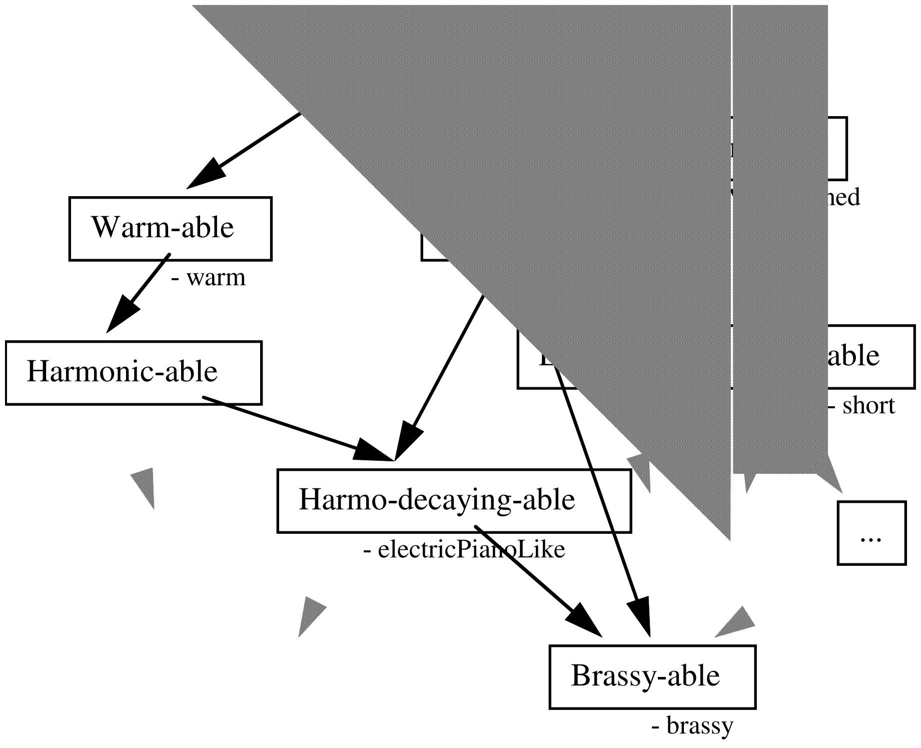

Figure 1. A part of the sound type hierarchy for the 05R/W synthesizer

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 {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").

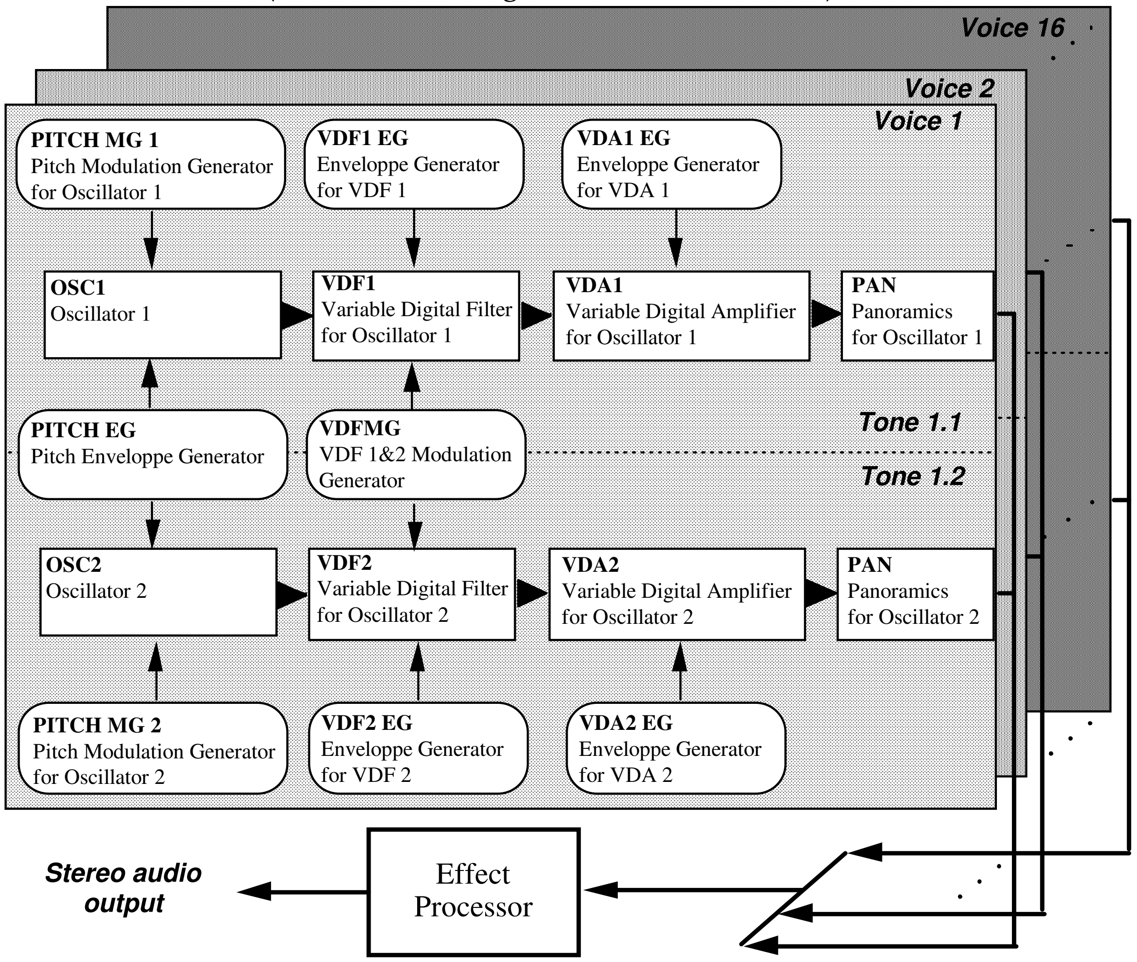
Voice 16
Voice 2
Voice 1
PITCH MG 1 Pitch Modulation Generator for Oscillator 1
VDF1 EG Enveloppe Generator for VDF 1
VDA1 EG Enveloppe Generator for VDA 1
OSC1 Oscillator 1
VDF1 Variable Digital Filter for Oscillator 1
VDA1 Variable Digital Amplifier for Oscillator 1
PAN Panoramics for Oscillator 1
PITCH EG Pitch Enveloppe Generator
VDFMG VDF 1&2 Modulation Generator
Tone 1.1
Tone 1.2
OSC2 Oscillator 2
VDF2 Variable Digital Filter for Oscillator 2
VDA2 Variable Digital Amplifier for Oscillator 2
PAN Panoramics for Oscillator 2
PITCH MG 2 Pitch Modulation Generator for Oscillator 2
VDF2 EG Enveloppe Generator for VDF 2
VDA2 EG Enveloppe Generator for VDA 2
Stereo audio output
Effect Processor

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

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.

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.

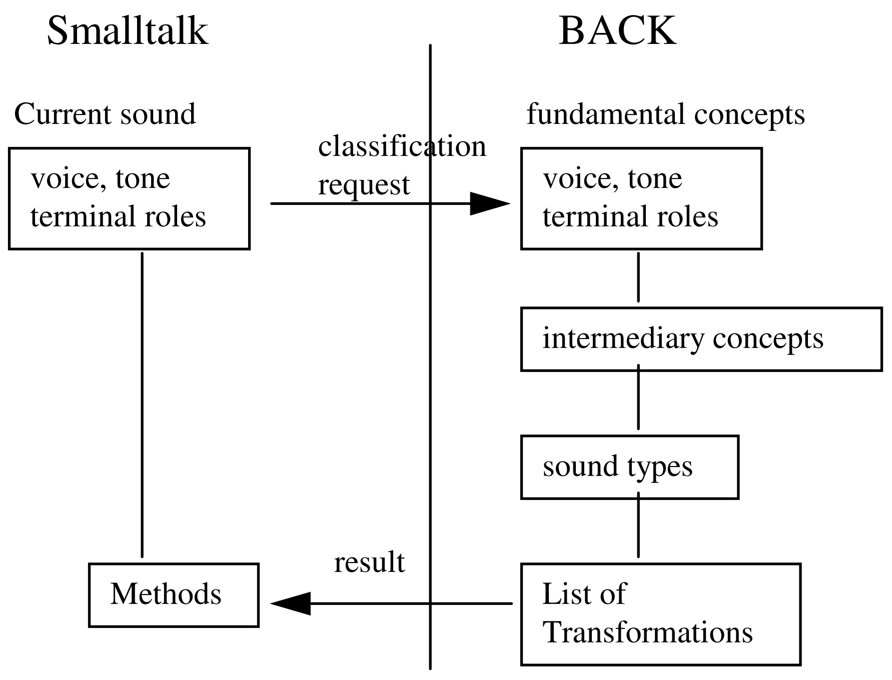

Figure 7. The two representations of sounds and their connection.

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.

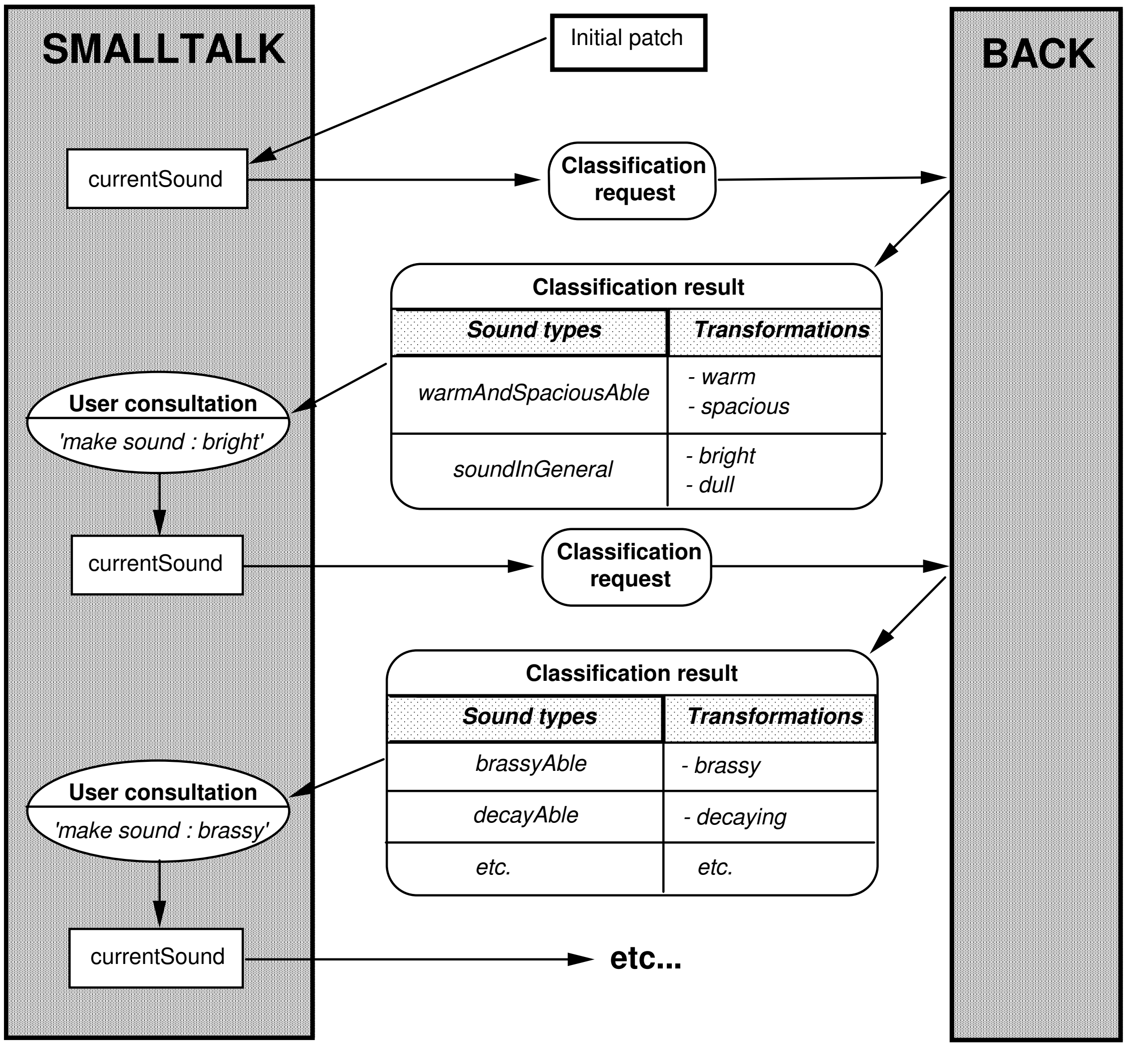

Figure 8 An example session.

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. "*A*n 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*, *AI^2 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." in *Proceedings of the 1989 International Computer Music Conference*, Columbus, Ohio. 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, Canada. 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. Reprinted in S. Pope (Ed.), *the Well-Tempered Object*, Cambridge, MA: MIT Press, 1991.

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.