A framework to integrate environments for computer aided composition and sound synthesis

Peter Hanappe, Gérard Assayag

A framework to integrate environments for computer aided composition and sound synthesis

Peter Hanappe et Gérard Assayag

Journées d'informatique musicale, Juin 1997, Lyon (France)

Abstract:

In this paper we define data structures which describe general concepts of composition and sound synthesis, and establish a client-server architecture. The combination of these two elements should allow preexisting and new applications to work together and share information.

1 Introduction

The work presented in this paper is part of a larger project that focuses on the control of sound synthesis. Already in 1963 Max Mathews reported the first experiments with digital sound synthesis (using the MusicV environment) and computer aided composition [Mathew 1963]. In the same article he also stated the need of a better understanding of psychoacoustic phenomena and more precise tools for the control of sound synthesis. Since then many applications for synthesis and CAC have reached computer screens: CSound [Vercoe 1986], CMix [Garton 1995], Max [Puckette 1991], Kyma [Scaletti 1989], and Modalys [Morisson et Adrien 1993] are more sound oriented; Formes [Rodet et Cointe 1984], PatchWork [Assayag et Rueda 1993], and DMIX [Oppenheim 1996] are more composition oriented; and Common Lisp Music [Taube 1991] and Foo [Eckel et Gonzales-Arroyo 1994] handle both. In the field psychoacoustics progress has also been made [MacAdams 1994]. In particular the concept of timbre is now better understood [McAdams, Winsberg, Donnadieu, De Soete & Krimphoff 1995]. There have been projects that targeted the control of sound synthesis [Mathews, Moore, Risset 1974, Wessel 1979, Wessel 1992, Miranda 1993, Rolland et Pachet 1995], however, few tools have really survived to the world of computer music. Our goal is to develop new tools which help the composer to create the sound he desires. We have started with examining some of the available environments for sound synthesis and composition, and tried to establish an environment for our experimentation.

As mentioned above, there already exist a wide variety of environments which have proven their usefulness, and which have their particular characteristics and user groups. We want to continue using these environments. Considering the implementation of a new environment for CAC or synthesis is beyond the scope of this project.

Every environment has some good tool or some feature that is not found in any other. Yet, there is rarely a way to make the two environments work together and benefit from both. For example, one environment can be gifted with a well designed breakpoint function editor. But it might be impossible to use the editor when working in a other environment. Making these environments communicate will extend the possibilities found in either of them.

One might wonder how control of sound synthesis is related to composition. The answer is closely. The composer Marco Stroppa, in one of our conversations, said that the control of sound synthesis is an act of composition, because a sound has a meaning only if it is imagined within a composition. Since control of sound synthesis is closely related to the use of timbre as a musical element in the composition [Lerdahl 1987], so should tools for the control of sound synthesis be intimately realted with CAC environments; to

the extend that they cooperate closely, but not completely depend upon each other. PatchWork, for example, includes a library to prepare data for CSound. But the data structures used by PatchWork are conceived uniquely for CSound. Converting this library to use with an other synthesis kernel takes more then a hard day's work. It would require re-designing the library.

A last argument concerns all large, monolithic applications in general. Extending the application or replacing an existing functionality is, in most cases, impossible for the user. This means no replacement of the breakpoint function editor with a better one. No adding a new signal processing function to the synthesis kernel. The user is forced to work with the application as it was designed by the author, even if is desirable to add new features.

So what can we conclude from these observations? First, that we need to come up with a strategy allowing our tools to be used from within the available environments. We then take advantage of existing software and guarantee our tools a greater usability. Second, the architecture of our solution should be modular, and its interfaces public. Users/programmers should be able to add new tools or replace some of the existing modules of the environment. Last, we attempt to make the environment independent of the computer platform. This will urge us to design a clean architecture independent of platform specific features, therefore running less risk of our work becoming obsolete.

We have developed a framework, with the project name JavaMusic, that tries to fulfill these requirements. This framework can be viewed as a crossroads where different applications/modules meet and share data, and whose architecture allows the dynamic addition and removal of synthesis kernels, CAC environments and other tools.

We hope to achieve this goal with the introduction of several elements:

- data structures which describe general concepts of composition and synthesis,
- translators to convert data structures of a specific environment to the proposed data structures and vice versa,
- a client-server architecture to connect the different environments.

For the development of JavaMusic we have chosen the programming language Java. Java has the advantages of being a high level, dynamic language which is freely available, and widely used. The main reason for choosing Java, however, is its portability. Indeed, Java is an interpreted language and its specifications are independent of the local CPU. Furthermore, Java provides the classes which abstract machine dependent system components such as the file system, graphics, and networking.

In the next section we discuss the data structures, then, in section 3, we will describe the client-server architecture.

2 Data structures

In this section we define the data structures that form the basis of our framework using the Java programming language. The data structures are grouped into three sections: Parameters, VirtualInstrument, and Composition.

2.1 Parameters

We will consider three types of Parameters: Numbers, Nuplets, and ControlSignals. We do not define the internal structure of the Parameters. Instead, we define the interface to the Parameters, i.e. how to access their data.

A Number represents a numerical value. Its interface defines two methods, one of which returns its value as an integer, the other as a floating point number.

```
 public interface Number

{  public int    intValue();

   public float  floatValue();

}
```

A Nuplet is an abstraction of an array of Numbers. Its interface is defined as follows:

```
 public interface Nuplet

{  public float  value(int i);

   public int    dimension();

}
```

The method `dimension` returns the length of the array. The method `value` returns the element with index `i` of the array. This structure can be used, for example, to stock the wave form of an oscillator.

A ControlSignal implements two methods:

```
 public interface ControlSignal

{  public Nuplet  value(float time);

   public int     dimension();

}
```

ControlSignals are data structures which have a time dimension. They will provide the necessary input values during the synthesis. The value of a ControlSignals, at any given time, is a Nuplet. Because the Nuplet has a fixed dimension larger or equal to zero, we can use ControlSignals as a multi-dimensional control structure; the Nuplet groups the value of every dimension into one object. Some examples of ControlSignals are the ConstantSignal and the BreakPointFunction.

2.2 VirtualInstrument

In this section we discuss a number of data structures which help us describe a synthesis technique. The definition of Module, Connection, and Patch will lead us to the discussion of VirtualInstrument.

2.2.1 Definitions

A Module is an object that has a type, a number of typed inputs, a number of typed outputs, and a value.

```
 public class Module

{  String mValue;
```

```
   byte   mModuleType;

   byte   mInputType[];

   byte   mOutputType[];

}
```

A Module abstracts a function which takes a number of inputs, performs some calculation and outputs a number of results.

Connections will be used to link Modules and are directed from an output to an input.

```
 public class Connection

{  Module mInModule;

   int    mInputNum;

   Module mOutModule;

   int    mOutputNum;

}
```

A Patch consists of a set of Modules and a set of Connections:

```
 public class Patch

{  Vector mModules;

   Vector mConnections;

}
```

2.2.2 Discussion

We want to present a formal description of a synthesis technique. We call such a description a VirtualInstrument [Battier1995]. We assume that synthesis techniques can be established using unit building blocks [Mathews 1963, Risset 1993]. This leads us to the definition of a VirtualInstrument as a Patch of Modules and Connections. Connections link Modules together to form a *networked instrument*.

A VirtualInstrument only *describes* a synthesis technique. The actual synthesis will be performed by a synthesis kernel. How the VirtualInstrument and its Modules will be converted to a set of signal processing functions within the kernel is briefly discussed in section 3.2.3.

We currently accept two basic types of Modules: param-Modules and mod-Modules. The mod-Module represents a primitive building block of the synthesis technique. When the VirtualInstrument is

implemented by the synthesis kernel, the mod-Module is mapped onto one of the signal processing functions of the kernel. The value of the mod-Module indicates the name of this function.

The param-Module is used to hand over data to the synthesis kernel, both for the initialization and for the control of the synthesis. Its value is an index in an array of Parameters (see section 2.3.2).

A couple of remarks. First, a VirtualInstrument must satisfy a number of constraints. For example, some synthesis environments only manage acyclic structures or tree structures. We will thus need additional functions to test these conditions.

Second, she current definition of VirtualInstrument is well suited to describe synthesis techniques that model the analoge studio (signal models) [Di Poli 1993]. It can also represent physical models that use a waveguide description. Physical models that use a modal description, however, are not well represented in this formalism. Indeed, in these models a two-way interaction between the Modules is necessary and the connections are expressed in terms of accesses at at certain location on the Module. The description for this interaction between Modules is not possible without complicating the current one and we will leave the issue as is for now.

A third remark concerns the multidimensional ControlSignals. This concept is not new [Eckel et Gonzales-Arroyo 1994], but we would like to underline its usefulness again. Consider that we want to use a VirtualInstrument which uses additive synthesis. Using the unit generators currently found in most synthesis kernels, we can construct a VirtualInstrument that synthesizes 1 component using a sine wave oscillator which is controlled in frequency and in amplitude. If we now want to use additive synthesis using 2 components, we have to use two sine wave oscillators. With the initial description of the VirtualInstrument altered, we need a description for an additive synthesis regardless of the number of components. For this problem we propose the use of multidimensional ControlSignals. If the ControlSignals input to sine wave oscillator have a dimension larger than one we sum the resulting signals.

Lastly, the value of a mod-Module now depends on the synthesis kernel that will realize the VirtualInstrument. Most kernels, however, offer similar kinds of signal processing functions. For example, modules for adding two sound signals can be found in every kernel. If we can determine the functions common to most kernels, and associate one value to every group of similar functions we can construct VirtualInstruments independent of the underlying kernel. This project, which we have not started yet, will be of importance when we create our tools for the control of sound synthesis.

2.3 Composition

If Parameters and VirtualInstruments stand closer to the synthesis, we now arrive at the definition of the structures which stand closer to composition.

2.3.1 SoundObject

A SoundObject is composed of a start time, a duration, and a reference to a Process.

```
 public class SoundObject

{  float   mStart;

   float   mDuration;

   Object  mProcess;

}
```

A SoundObject is *one* element of the composition. It can represent a single note as well as a complex sound that evolves in time - in essence ``a single sound which might last several minutes with an inner life, and ... [has] the same function in the composition as an individual note once had." [Cott 1974]. SoundObjects can be seen as ``cells, with a birth, life and death" [Grisey 1987].

The Process is a structure that determines the content of the SoundObject. It is the life, evolution, or force of a SoundObject. The Process can be one of two different kinds: it can be a SoundProcess or a Texture.

2.3.2 SoundProcess

```
 public class SoundProcess
{  VirtualInstrument mVirtualInstrument;

   Parameter         mParameter[];

}
```

A SoundProcess is an object which represents a synthesis process, and which contains the recipe and the ingredients for this synthesis. A SoundProcess is the combination of a VirtualInstrument and an array of Parameters. The VirtualInstrument describes the synthesis algorithm. The Parameters serve as control structures for the synthesis, or as initialization values during the creation of the VirtualInstrument.

2.3.3 Texture

```
 public class Texture

{  SoundObject mSoundObject[];

}
```

A Texture is a *composed* Process and contains a number of SoundObjects. The definitions of Textures and SoundObjects refer to each other: a SoundObject can refer to a Texture that itself can refer to a number of SoundObjects. However, we do not allow cyclic paths: a Texture cannot contain a SoundObject referring to the initial Texture. The composition can thus be organized in a tree structure [Rodet et Cointe 1984].

3 Client-server architecture

In this section we come to description of our client-server architecture. What we are looking for is not a new application but rather a strategy that allows the different kernels to communicate and work together. To accomplish this we have based the architecture of JavaMusic on the work by Anselm Baird-Smith [Baird-Smith 1995].

3.1 Distributed personalization of applications

The first part of Baird-Smith's work focuses on the dynamic and distributed personalization of applications. We will give a simple example as an introduction. Consider an editor which connects itself from a remote computer to an application whose graphical interface we want to personalize. The focused application incorporates a communication kernel through which the editor can connect itself. Once the

communication is established the editor can change the appearance of the application (e.g. window color) and its behavior (e.g. mouse click). This concept of distributed personalization is worked out first by defining the concept of Resource and second by the definition of a communication protocol between the application and the editor.

In the following sections, we will call the editor the client and the application the server.

3.1.1 Resource

The Resources of an application are defined as the objects which the application developer has made accessible and editable such that the end user can personalize the application.

A Resource has a name, a value, and an identification number. This identification number is unique such that at any given time there exists a bijection between the set of identification numbers and the set of Resources.

The Resources are structured hierarchically into a tree structure: a Resource can have any number of children, and one parent.

3.1.2 Communication protocol

If we want to establish a communication between the client and the server a communication protocol must be defined. The primitives currently found in the protocol include setting and retrieving the value, requesting the statistics, and obtaining the list of children of a resource. The protocol also includes a notification mechanism which informs the clients of the operations performed on a Resource. To the operations defined by Baird-Smith, we have added the creation and deletion of Resources.

3.2 Services, Providers and Requests

In JavaMusic every instance of SoundProcess, Texture, VirtualInstrument, or Parameter is stored as a Resource and can be created and manipulated by clients that connect themselves to JavaMusic. The functionality offered by JavaMusic has become minimal. The application reduces to a server, concerned only with the management of the Resources and notifying the clients of any change. Examples of clients that will connect to the server are CAC environments and synthesis kernels. The question we will concern ourself with next is how a CAC environment can call a synthesis kernel and ask to synthesize a SoundObject. How does a CAC environment even know what clients are connected, and what services they provide?

We have adopted the following solution. Clients can create a Resource which includes a reference to themselves. This Resource can have children describing the services the client assumes. We have defined a class Service for this description. This class holds the name of the service as well as the type of arguments needed and the type of result returned by the service.

We will call a client that settles itself as a Resource and publishes a number of Services a Provider. A client can now inspect the available Resources and search for the appropriate Provider and Service. To make use of a service, the client sends the Provider a Request. A Request is an object containing the name and arguments for the service. On completion the Provider returns the Request, including the result of the service, to the client.

This modular architecture provides the means of communication between different parts of an application without the need of knowing each other's interface before hand. New Providers and new Services can be added easily using the dynamic instantiating Java offers. What Providers are inserted into the environment, and what Services they implement is open to the user. The environment becomes a collection of specialized Providers, each dealing with one specific aspect of composition [Oppenheim

1996].

Providers which we need in particular are editors for Textures, SoundProcesses, and VirtualInstruments, CAC environments, and synthesis kernels. Some of these, such as the editor for Textures, we will need to create ourself. Others, such as the synthesis kernels, can be based on preexisting software. In the next paragraphs we comment on some of the currently existing Providers.

3.2.1 CommonLispKernel

We have made a wrapper class that abstract the Common Lisp environment. This environment attains its importance because of the CAC environments such as PatchWork, OpenMusic [Assayag 1996] and Common Music, which are written in the Common Lisp language.

Communication between JavaMusic and Common Lisp is established over a TCP/IP connection, and enabled in both directions. Services in the JavaMusic environment can be called from within the Common Lisp environment, and clients, in their turn, can request the evaluation of a lisp expression.

3.2.2 Synthesis kernels

Most synthesis kernel were developed in the ANSI C language on Unix machines and have a command line interface. We compile these libraries as shared libraries. Java has the facility to declare methods as native. This specifier determines that the method is written in C and is implemented in a shared library. Using this facility we create a wrapper class which calls the main function of the synthesis kernel. We are currently using CSound [Vercoe 1986].

3.2.3 Player

The Player is a Provider which offers to synthesize a SoundObject. It does this with the help of the available synthesis kernels and a Scheduler for every kernel. The Scheduler converts the VirtualInstrument and the Parameters to a description understood by the kernel before asking the kernel to synthesize the SoundObject. If a user want to provide a new synthesis kernel, he also has to provide the associated Scheduler.

4 Conclusion

We have defined a set of classes which reflect general concepts of composition and sound synthesis. Together with the client-server architecture they form a framework within which a complete environment for composition and sound synthesis can be built. This framework allows preexisting environments to cooperate, and provides a common ground for new tools of composition and synthesis. Its modular design allows the addition or replacement of software tools with a minimum of effort. The architecture has the additional advantage that it can be distributed over a network, and that, for example, the synthesis kernel can be run on one computer and shared between several users.

References

[Assayag et Rueda 1993] G. Assayag and C. Rueda. The music representation project at Ircam. In Proceedings of the Int. Computer Music Conference, Tokyo, Japan, 1993. Int. Computer Music Association.

[Assayag 96] G. Assayag. OpenMusic. In Proceedings of the Int. Computer Music Conference, Hong Kong, 1996. Int. Computer Music Association.

[Battier95] M. Battier. In Les Cahiers de l'Ircam: Instruments, Paris, France, 1995. Editions Ircam - Centre Georges-Pompidou.

[Baird-Smith 1995] A. Baird-Smith. Distribution et Interpretation dans les Interfaces Homme-Machine. PhD thesis, Universite Paris VI, Paris, France, 1995.

[Cott 74] J. Cott. Stockhausen: Conversations with the Composer. Picador (Pan Books Ltd.), London, 1974. ISBN 0-33024165-6.

[Di Poli 1993] G. De Poli. Audio signal processing by computer. In G. Haus, editor, Music Processing. Oxford University Press, 1993. BN 0-19-816372-X.

[Eckel, Gonzales-Arroyo 1994] G. Eckel and R. Gonzalez-Arroyo. Musically salient control abstractions for sound synthesis. In Proceedings of the Int. Computer Music Conference, Aarhus, Danmark, 1994. Int. Computer Music Association.

[Garton 1995] B. Garton. The CMIX Home Page, 1995.
CMIX

[Grisey 1987] G. Grisey. Tempus ex machina: A composer's reflection on musical time. Contemporary Music Review, 2:239-275, 1987. Harwood Academic Publishers.

[Lerdhal 1987] F. Lerdahl. Timbral hierarchies. Contemporary Music Review, 2:135-160, 1987. Harwood Academic Publishers.

[Morrison Adrien 1993] J.D. Morrison and J.-M. Adrien. Mosaic: A framework for modal synthesis. Computer Music Journal, 17(1), Spring 1993. MIT Press.

[Mathews 1963] M.V. Mathew. The digital computer as a musical instrument. Science, 142:553-557, November 1963. Am. Ass. for the Advancement of Science.

[McAdams 1994] S McAdams. Audition: physiologie, perception et cognition. In Traite de psychologie experimentale 1. ed. Richelle, Requin and Robert, Presses Universitaire de France, 1994.

[Miranda 1993] E.R. Miranda. From symbols to sound: Artificial intelligence investigation of sound synthesis. DAI Research Paper 640, Dept. of AI, University of Edinburgh, UK, 1993.

[Mathews, Moore et Risset 1974] M.V. Mathew, F.R. Moore, and J.C Risset. Computers and future music. Science, 183:263-268, January 1974. Am. Ass. for the Advancement of Science.

[McAdams, Winsberg, Donnadieu et De Soete 1995] S. McAdams, S. Winsberg, S. Donnadieu, G. De Soete, and J. Krimphoff. Perceptual scaling of synthesized musical timbres: Common dimensions, specificities, and latent subject classes. Psychol. Res., 58:117-192, 1995. Springer-Verlag.

[Oppenheim 1996] D. Oppenheim. DMIX: A multi faceted environment for composing and performing computer music. Mathematics and Computers, 1996.

[Puc91] M. Puckette. Combining events and signal processing in the max graphical programming environment. Computer Music Journal, 15(3), Fall 1991. MIT Press.

[Rodet Cointe 1984] X. Rodet and P. Cointe. FORMES: Composition and scheduling of processes. In C. Roads, editor, The Music Machine, Cambridge Massachusetts, 1984. MIT Press. Version française

[Risset 1993] J.-C. Risset. Synthèse et matériau sonore. In Les Cahiers de l'Ircam: La Synthèse Sonore, Paris, France, 1993. Editions Ircam - Centre Georges-Pompidou.

[Rolland Pachet 1995] P.Y. Rolland and F. Pachet. A framework for representing knowledge about synthesizer programming. Publication LAFORIA, LAFORIA-IBP, Univ. Paris 6, France, 1995.

[Scaletti 1989] C. Scaletti. The Kyma/Platypus computer music workstation. Computer Music Journal, 13(2):23-38, Summer 1989. MIT Press.

[Taube 1991] H. Taube. Common music: A music composition language in Common Lisp and CLOS. Computer Music Journal, 15(2), Summer 1991. MIT Press.

[Vercoe 1986] B. Vercoe. Csound: A Manual for the Audio Processing System and Supporting Programs with Tutorials. Media Lab, MIT, 1986.

[Wessel 1979] D. Wessel. Timbre space as a musical control structure. Computer Music Journal, 3(2):45-52, 1979. MIT Press.

[Wessel 1992] D. Wessel. Connectionist models for real-time control of synthesis and compositional algorithms. In Proceedings of the Int. Computer Music Conference, San Jose, California, 1992. Int. Computer Music Association.