CLAM, an Object Oriented Framework for Audio and Music

Pau Arumí and **Xavier Amatriain**
Music Technology Group, Universitat Pompeu Fabra
08003 Barcelona, Spain
{parumi,xamat}@iua.upf.es

Abstract

CLAM is a C++ framework that is being developed at the Music Technology Group of the Universitat Pompeu Fabra (Barcelona, Spain). The framework offers a complete development and research platform for the audio and music domain. Apart from offering an abstract model for audio systems, it also includes a repository of processing algorithms and data types as well as a number of tools such as audio or MIDI input/output. All these features can be exploited to build cross-platform applications or to build rapid prototypes to test signal processing algorithms.

Keywords

Development framework, DSP, audio, music, object-oriented

1 Introduction

CLAM stands for C++ Library for Audio and Music and is a full-fledged software framework for research and application development in the audio and music domain. It offers a conceptual model as well as tools for the analysis, synthesis and transformation of audio signals.

The initial objective of the CLAM project was to offer a complete, flexible and platform independent sound analysis/synthesis C++ platform to meet the needs of all the projects of the Music Technology Group (**?**) at the Universitat Pompeu Fabra in Barcelona. Those initials objectives have slightly changed since then, mainly because the library is no longer seen as an internal tool for the MTG but as a framework licensed under the GPL (**?**).

CLAM became public and Free in the course of the AGNULA IST European project (**?**). Some of the resulting applications as well as the framework itself were included in the Demudi distribution.

Although nowadays most the development is done under GNU/Linux, the framework is cross-platform. All the code is ANSI C++ and it is regularly compiled under GNU/Linux, Windows and Mac OSX using the GNU C++ compiler but also the Microsoft compiler.

CLAM is Free Software and all its code and documentation can be obtained though its web page (**?**).

2 What CLAM has to offer ?

Although other audio-related environments exist [1] —see (Amatriain, 2004) for an extensive study and comparison of most of them— there are some important features of our framework that make it somehow different:

- All the code is *object-oriented* and written in C++ for efficiency. Though the choice of a specific programming language is no guarantee of any style at all, we have tried to follow solid design principles like design patterns (**?**) and C++ idioms (**?**), good development practices like test-driven development (**?**) and refactoring (**?**), as well as constant peer reviewing.
- It is *efficient* because the design decisions concerning the generic infrastructure have been taken to favor efficiency (i.e. inline code compilation, no virtual methods calls in the core process tasks, avoidance of unnecessary copies of data objects, etc.)
- It is *comprehensive* since it not only includes classes for processing (i.e. analysis, synthesis, transformation) but also for audio and MIDI input/output, XML and SDIF serialization services, algorithms, data visualization and interaction, and multi-threading.
- CLAM deals with wide variety of *extensible data types* that range from low-level signals (such as audio or spectrum) to higher-level semantic-structures (a musical phrase or an audio segment)

[1] to cite only some of them: OpenSoundWorld, PD, Marsyas, Max, SndObj and SuperCollider

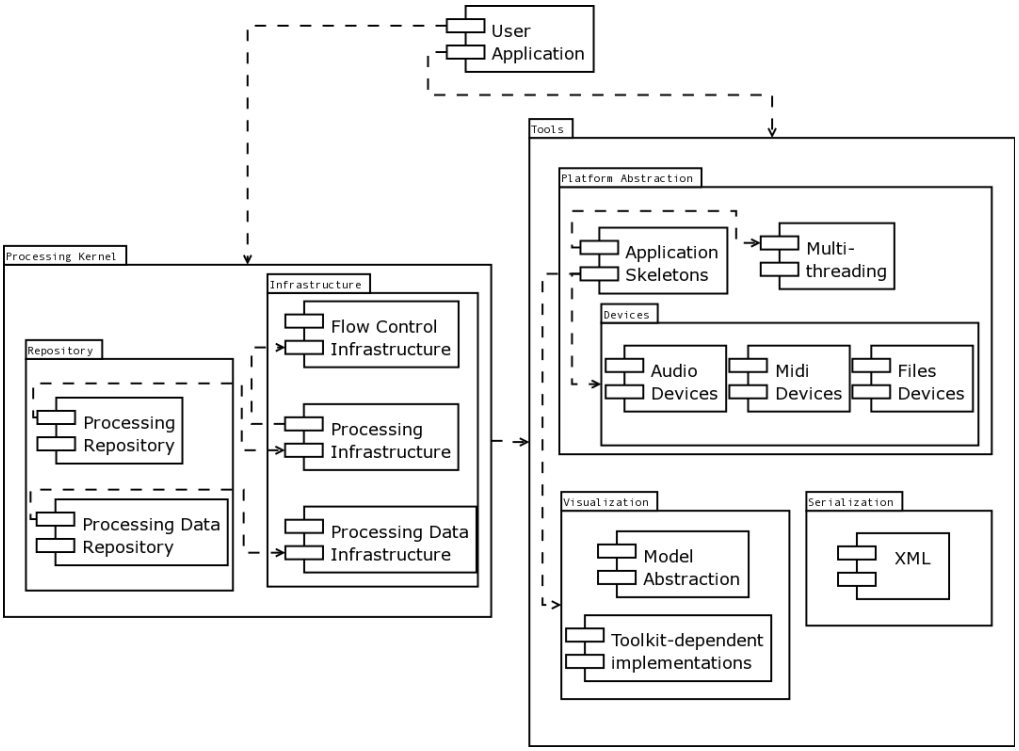

Figure 1: CLAM modules

- As stated before, it is *cross-platform*
- The project is licensed under the *GPL* terms and conditions.
- The framework can be used either as a regular C++ *library* or as a *prototyping tool*.

In order to organise all these features CLAM is divided into different architectonic modules. Figure 1 shows the modules and submodules that exist in CLAM. The most important ones are those related to the processing kernel, with its *repositories* and *infrastructure* modules. Furthermore, a number of auxiliary *tools* are also included.

In that sense, CLAM is both a *black-box* and a *white-box* framework (**?**). It is black-box because already built-in components included in the repositories can be connected with minimum programmer effort in order to build new applications. And it is *white-box* because the abstract classes that make up the infrastructure can be easily derived in order to extend the framework components with new processes or data classes.

2.1 The CLAM infrastructure

The CLAM infrastructure is defined as the set of abstract classes that are responsible for the white-box functionality of the framework and define a related *metamodel* [2]. This metamodel is very much related to the Object-Oriented paradigm and to Graphical Models of Computation as it defines the object-oriented encapsulation of a mathematical graph that can be effectively used for modeling signal processing systems in general and audio systems in particular.

The metamodel clearly distinguishes between two different kinds of objects: *Processing* objects and *Processing Data* objects. Out of the two, the first one is clearly more important as the managing of Processing Data constructs can be almost transparent for the user. Therefore, we can view a CLAM system as a set of Processing objects connected in a graph called *Network*.

Processing objects are connected through intermediate channels. These channels are the only mechanism for communicating between Processing objects and with the outside world. Messages are enqueued (produced) and dequeued (consumed) in these channels, which acts as FIFO queues.

In CLAM we clearly differentiate two kinds of communication channels: *ports* and *controls*.

[2] The word *metamodel* is here understood as a "model of a family of related models", see (Amatriain, 2004) for a thorough discussion on the use of metamodels and how *frameworks* generate them.

Ports have a synchronous data flow nature while controls have an asynchronous nature. By synchronous, we mean that messages get produced and consumed at a predictable —if not fixed— rate. And by asynchronous we mean that such a rate doesn't exist and the communication follows an event-driven schema.

Figure 2 is a representation of a CLAM processing. If we imagine, for example, a processing that performs a frequency-filter transformation on an audio stream, it will have an input and an out-port for the incoming audio stream and processed output stream. But apart from the incoming and outcoming data, some other entity —probably the user through a GUI slider— might want to change some parameters of the algorithm.

This control data (also called events) will arrive, unlike the audio stream, sparsely or in bursts. In this case the processing would want to receive these control events through various (input) control channels: one for the gain amount, another for the frequency, etc.

The streaming data flows through the ports when a processing is fired (by receiving a *Do()* message).

Different processings can consume and produce at different velocities or, likewise, a different number of tokens. Connecting these processings is not a problem as long as the ports are of the same data type. The connection is handled by a *FlowControl* entity that figures out how to schedule the firings in a way that avoids firing a processing with not enough data in its input-port or not enough space into its output-ports.

Configurations: why not just controls? Apart from the input-controls, a processing receives another kind of parameter: the configurations.

Configurations parameters, unlike controls parameters, are dedicated to expensive or structural changes in the processing. For instance, a configuration parameter can decide the number of ports that a processing will have. Therefore, a main difference with controls is that they can only be set into a processing when they are not in running state.

Composites: static vs dynamic It is very convenient to encapsulate a group of processings that works together into a new composite processing. Thus, enhancing the abstraction of processes.

CLAM have two kinds of composites: *static* or hardcoded and *dynamic* or nested-networks. In both cases inner ports and controls can *published* to the parent processing.

Choosing between the static vs dynamic composites is a trade-off between boosting efficiency or understandability. See in-band pattern in (**?**).

2.2 The CLAM repositories

The *Processing Repository* contains a large set of ready-to-use processing algorithms, and the *Processing Data Repository* contains all the classes corresponding to the objects being processed by these algorithms.

The Processing Repository includes around 150 different Processing classes, classified in the following categories: Analysis, ArithmeticOperators, AudioFileIO, AudioIO, Controls, Generators, MIDIIO, Plugins, SDIFIO, Synthesis, and Transformations.

Although the repository has a strong bias toward spectral-domain processing because of our group's background and interests, there are enough encapsulated algorithms and tools so as to cover a broad range of possible applications.

On the other hand, in the Processing Data Repository we offer the encapsulated versions of the most commonly used data types such as Audio, Spectrum, SpectralPeaks, Envelope or Segment. It is interesting to note that all of these classes have interesting features such as a homogeneous interface or built-in automatic XML persistence.

2.3 Tools

XML Any CLAM *Component* can be stored to XML as long as `StoreOn` and `LoadFrom` methods are provided for that particular type (**?**). Furthermore, Processing Data and Processing Configurations –which are in fact Components– make use of a macro-derived mechanism that provides automatic XML support without having to add a single line of code (**?**).

GUI Just as almost any other framework in any domain, CLAM had to think about ways of integrating the core of the framework tools with a graphical user interface that may be used as a front-end of the framework functionalities.

The usual way to work around this issue is to decide on a graphical toolkit or framework and add support to it, offering ways of connecting the framework under development to the widgets and other graphical tools included in the graphical framework. The CLAM team, however, aimed at offering a toolkit-independent

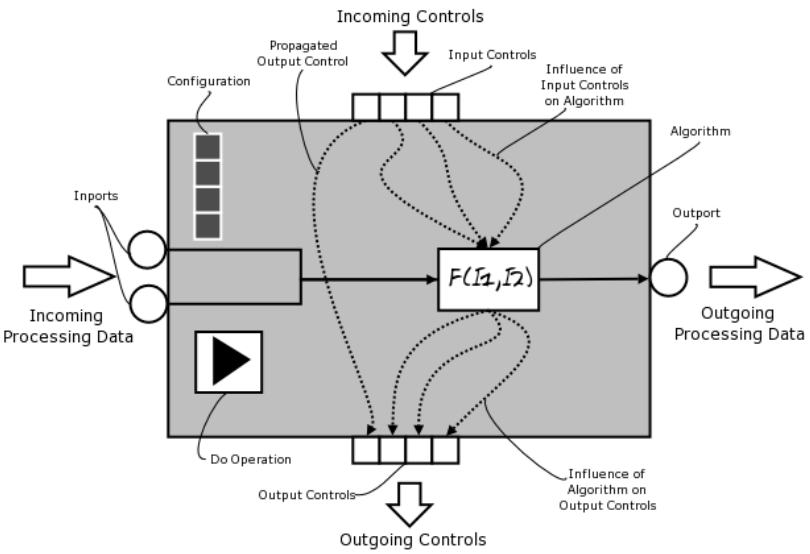

Figure 2: CLAM processing detailed representation

support. This is accomplished through the CLAM Visualization Module.

This general Visualization infrastructure is completed by some already implemented presentations and widgets. These are offered both for the FLTK toolkit (**?**) and the QT framework (**?**; **?**). An example of such utilities are convenient debugging tools called Plots. Plots offer ready-to-use independent widgets that include the presentation of the main Processing Data in the CLAM framework such as audio, spectrum, spectral peaks...

Platform Abstraction Under this category we include all those CLAM tools that encapsulate system-level functionalities and allow a CLAM user to access them transparently from the operating system or platform.

Using these tools a number of services –such as Audio input/output, MIDI input/output or SDIF file support– can be added to an application and then used on different operating systems without changing a single line of code.

3 Levels of automation

The CLAM framework offers three different levels of automation to a user who wants to use its repositories, which can also be seen as different levels of use of the generic infrastructure:

Library functions The user has explicit objects with processings and processing data and calls processings *Do* methods with data as its parameters. Similarly as any function library.

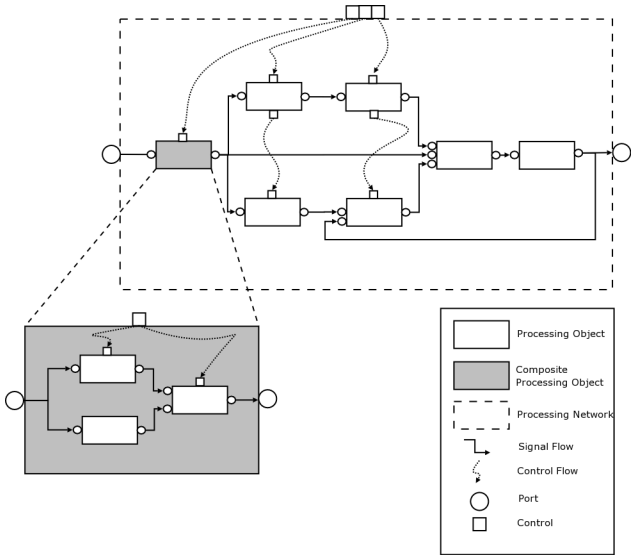

Figure 3: a CLAM processing network

Processing Networks The user has explicit processing objects but streaming data is made implicit, through the use of ports. Nevertheless, the user is in charge of *firing*, or calling a *Do()* method without parameters.

Automatic Processing Networks It offers a higher level interface: processing objects are hidden behind a layer called Network, see Figure 3 Thus, instantiation of processing objects are made through passing strings identifiers to a factory. Static factories are a well documented C++ idiom (**?**) that allow us to decouple the factory class with its registered classes in a very convenient way. They makes the process of adding or removing processings to the reposi-

tory as easy as issuing a single line of code in the processing class declaration.

Apart from *instantiation*, the Network class offers interface for connecting the components processings and, most important, it automatically controls the firing of processings (calling its *Do* method).

Actually, the firing scheduling can follow different strategies, for example a *push strategy* starting firing the up-source processings, or a *pull strategy* where we start querying for data to the most down-stream processings, as well as being dynamic or static (fixed list of firings). See (**?**; **?**) for more details on scheduling dataflow systems.

To accommodate all this variability CLAM offers different FlowControl sub-classes which are in charge of the firing strategy, and are pluggable to the Network processing container.

4 Integration with GNU/Linux Audio infrastructure

CLAM input/output processings can deal with a different kinds of device abstraction architectures. In the GNU/Linux platform, CLAM can use audio and midi devices through the ALSA layer (**?**), and also through the portaudio and portmidi (**?**; **?**) layers.

ALSA: ALSA low-latency drivers are very important to obtain real-time input/output processing. CLAM programs using a good soundcard in conjunction with ALSA drivers and a well tuned GNU/Linux system —with the real-time patch— obtains back-to-back latencies around 10ms.

Audio file libraries: Adding audio file writing and reading capability to CLAM has been a very straight-forward task since we could delegate the task on other good GNU/Linux libraries: *libsndfile* for uncompressed audio formats, *libvorbis* for ogg-vorbis format and finally *libmad* and *libid3* for the mp3 format.

Jack: Jack support is one of the big to-dos in CLAM. It's planned for the 1.0 release or before —so in a matter of months. The main problem now is that Jack is callback based while current CLAM I/O is blocking based. So we should build an abstraction that would hide this peculiarity and would show those sources and sinks as regular ones.

LADSPA plugins: LADSPA architecture is fully supported by CLAM. On one hand, CLAM can host LADSPA plugins. On the other hand, processing objects can be compiled as LADSPA plugins.

LADSPA plugins transform buffers of audio while can receive control events. Therefore these plugins map very well with CLAM processings that have exclusively audio ports (and not other data types ports) and controls.

CLAM takes advantage of this fact on two ways: The *LADSPA-loader* gets a *.so* library file and a plugin name and it automatically instantiate a processing with the correspondent audio ports and controls. On the other hand, we can create new LADSPA plugins by just compiling a C++ template class called LadspaProcessingWrapper, where the template argument is the wrapped processing class.

DSSI plugins: Although CLAM still does not have support for DSSI plugins, the recent development of this architecture allowing graphical user interface and audio-instruments results very appealing for CLAM. Thus additions in this direction are very likely. Since CLAM provides visual tools for rapid prototyping applications with graphical interface, these applications are very suitable to be DSSI plugins.

4.1 What CLAM can be used for ?

The framework has been tested on —but also has been driven by— a number of applications, for instance: SMSTools, a SMS Analysis/Synthesis (**?**) graphical tool; Salto (**?**), a sax synthesizer; Rappid (**?**) a real-time processor used in live performances.

Other applications using CLAM developed at the research group includes: audio features extraction tools, time-stretching plugins, voice effect processors, etc.

Apart from being a programmers framework to develop applications, the latest developments in CLAM have brought important features that fall into the *black-box* and *visual builder* categories.

That lets a user concentrate on the research of algorithms forgetting about application development. And, apart from research, it is also valuable for rapid application prototyping of applications and audio-plugins.

5 Rappid Prototyping in CLAM

5.1 Visual Builder

Another important pattern that CLAM uses is the *visual builder* which arises from the observation that in a *black-box* framework, when

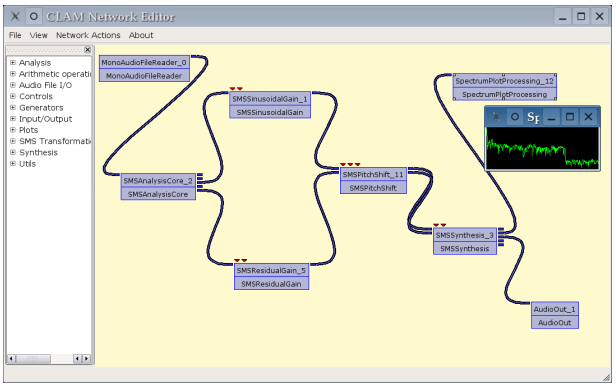

Figure 4: NetworkEditor, the CLAM visual builder

connecting objects the connection script is very similar from one application to another.

Acting as the *visual builder*, CLAM have a graphical program called NetworkEditor that allows to generate an application –or at least its processing engine– by graphically connecting objects. And another application called Prototyper, that acts as the glue between an application GUI designed with a graphical tool and the processing engine defined with the NetworkEditor.

5.2 An example

Here we will show how we can set up a graphical stand-alone program in just few minutes. The purpose of this program is to make some spectral transformations in real-time with the audio taken from the audio-card, apply the transformations and send the result back to the audio-card. The graphical interface will consist in a simple pane with different animated representations of the result of the spectral analysis, and three sliders to change transformation parameters.

First step: building the processing network (Figure 4) Patching with NetworkEditor is a very intuitive task to do. See Figure 4. We can load the desired processings by dragging them from the left panel of the window. Once in the patching panel, processings are viewed as little boxes with attached inlets and outlets representing its ports and control. The application allows all the typical mouse operations like select, move, delete and finally, connect ports and controls.

Since CLAM ports are typed, not all out-ports are compatible with all in-ports. For example in the Figure 4, the second processing in the chain is called SMSAnalysis and receives audio samples and produces: sinusoidal peaks, fundamental, several spectrums (one corresponding to the original audio and another corresponding to the residual resulting of subtracting the sinusoidal component).

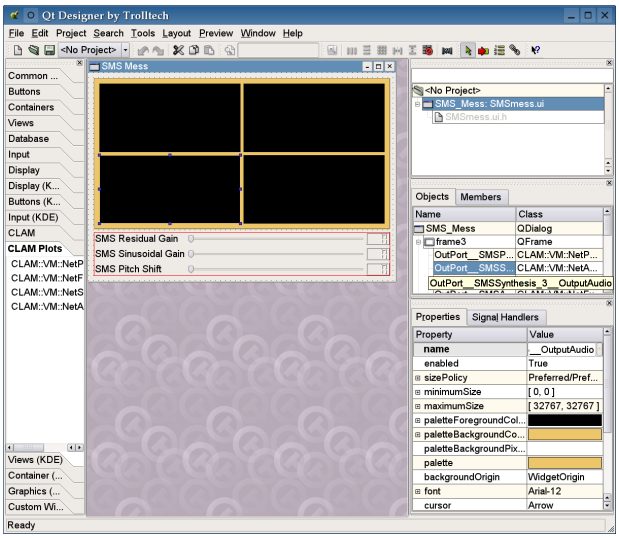

Figure 5: the QT GUI designer tool

Connected to SMSAnalysis out-ports we have placed three processings to perform transformations: one for controlling the gain of the sinusoidal component, another to control the gain of the residual component and the last one for shifting the pitch. The latest modifies both sinusoidal and residual components.

Then the signal chain gets into the SMSSynthesis which output the resynthesizes audio ready to feed the AudioOut (which makes the audio-card to sound)

Before starting the execution of the network, we can right click upon any processing view to open a dialog with its configuration. For instance, the SMSAnalysis configuration includes the window type and window size parameters among many others.

Another interesting feature of the NetworkEditor is that it allows loading visual plots widgets for examining the flowing data in any out-port. And also, slider widgets to connect to the in-control inlets.

Once the patch is finished we are ready to move on directly to the graphical user interface.

Second step: designing the program GUI (Figure 5) The screen-shot in Figure 5 is taken while creating a front end for our processing network. The designer is a tool for creating graphical user interfaces that comes with the QT toolkit (**?**; **?**).

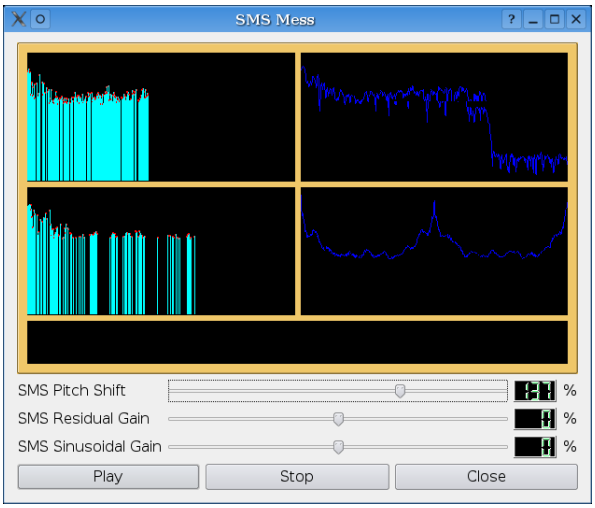

Figure 6: the running prototype

Normal sliders can be connected to processing in-ports by just setting a suited name in the properties box of the widget. Basically this name specify three things in a row: that we want to connect to an in-control, the name that the processing object have in the network and the name of the specific in-control.

On the other hand we provide the designer with a *CLAM Plots* plugin that offers a set of plotting widgets that can be connected to out-ports.

In the example in Figure 5 the black boxes corresponds to different plots for spectrum, audio and sinusoidal peaks data.

Now we just have to connect the plots widgets by specifying —like we did for the sliders— the out-ports we want to inspect.

We save the designer *.ui* file and we are ready to run the application.

Third step: running the prototype (Figure 6) Finally we run the prototyper program. Figure 6. It takes two arguments, in one hand, the xml file with the network specification and in the other hand, the designer ui file.

This program is in charge to load the network from its xml file —which contains also each processing configuration parameters— and create objects in charge of converting QT signals and slots with CLAM ports and controls.

And done! we have created, in a matter of minutes, a prototype that runs fast C++ compiled code without compiling a single line.

6 Conclusions

CLAM has already been presented in other conferences like the OOPSLA'02 (**?**; **?**) but since then, a lot of progress have been taken in different directions, and specially in making the framework more *black-box* with *visual builder* tools.

CLAM has proven being useful in many applications and is becoming more and more easy to use, and so, we expect new projects to begin using the framework. Anyway it has still not reached a the stable 1.0 release, and some improvements needs to be done.

See the CLAM roadmap in the web (**?**) for details on things to be done. The most prominent are: *Library-binaries* and separate sub-modules, since at this moment modularity is mostly conceptual and at the source code organization level. Finish the audio *feature-extraction framework* which is work-in-progress. *Simplify parts of the code*, specially the parts related with processing data and configurations classes. Have working *nested networks*

7 Acknowledgements

The authors wish to recognize all the people who have contributed to the development of the CLAM framework. A non-exhaustive list should at least include Maarten de Boer, David Garcia, Miguel Ramírez, Xavi Rubio and Enrique Robledo.

Some of the the work explained in this paper has been partially funded by the Agnula Europan Project num.IST-2001-34879.

References

A. Alexandrescu. 2001. *Modern C++ Design*. Addison-Wesley, Pearson Education.

X. Amatriain, P. Arumí, and M. Ramírez. 2002a. CLAM, Yet Another Library for Audio and Music Processing? In *Proceedings of the 2002 Conference on Object Oriented Programming, Systems and Application (OOPSLA 2002)(Companion Material)*, Seattle, USA. ACM.

X. Amatriain, M. de Boer, E. Robledo, and D. Garcia. 2002b. CLAM: An OO Framework for Developing Audio and Music Applications. In *Proceedings of the 2002 Conference on Object Oriented Programming, Systems and Application (OOPSLA 2002)(Companion Material)*, Seattle, USA. ACM.

X. Amatriain. 2004. *An Object-Oriented Meta-*

model for Digital Signal Processing. Universitat Pompeu Fabra.

K Beck. 2000. *Test Driven Development by Example*. Addison-Wesley.

J. Blanchette and M. Summerfield. 2004. *C++ GUI Programming with QT 3*. Pearson Education.

AGNULA Consortium. 2004. AGNULA (A GNU Linux Audio Distribution) homepage, http://www.agnula.org.

FLTK. 2004. The fast light toolkit (fltk) homepage: http://www.fltk.org.

M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. 1999. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley.

Free Software Foundation. 2004. Gnu general public license (gpl) terms and conditions. http://www.gnu.org/copyleft/gpl.html.

Johnson R. Gamma E., Helm R. and Vlissides J. 1996. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley.

D. Garcia and X. Amatrian. 2001. XML as a means of control for audio processing, synthesis and analysis. In *Proceedings of the MOSART Workshop on Current Research Directions in Computer Music*, Barcelona, Spain.

J. Haas. 2001. SALTO - A Spectral Domain Saxophone Synthesizer. In *Proceedings of MOSART Workshop on Current Research Directions in Computer Music*, Barcelona, Spain.

C. Hylands et al. 2003. Overview of the Ptolemy Project. Technical report, Department of Electrical Engineering and Computer Science, University of California, Berklee, California.

D. A. Manolescu. 1997. A Dataflow Pattern Language. In *Proceedings of the 4th Pattern Languages of Programming Conference*.

MTG. 2004. Homepage of the Music Technology Group (MTG) from the Universitat Pompeu Fabra. http://www.iua.upf.es/mtg.

D. Roberts and R. Johnson. 1996. Evolve Frameworks into Domain-Specific Languages. In *Procedings of the 3rd International Conference on Pattern Languages for Programming*, Monticelli, IL, USA, September.

E. Robledo. 2002. RAPPID: Robust Real Time Audio Processing with CLAM. In *Proceedings of 5th International Conference on Digital Audio Effects*, Hamburg, Germany.

X. Serra, 1996. *Musical Signal Processing*, chapter Musical Sound Modeling with Sinusoids plus Noise. Swets Zeitlinger Publishers.

Trolltech. 2004. Qt homepage by trolltech. http://www.trolltech.com.

www ALSA. 2004. Alsa project home page. http://www.alsa-project.org.

www CLAM. 2004. CLAM website: http://www.iua.upf.es/mtg/clam.

www PortAudio. 2004. PortAudio homepage: www.portaudio.com.

www PortMidi. 2004. Port Music homepage: http://www-2.cs.cmu.edu/ music/portmusic/.

www Ptolemy. 2004. Ptolemy project home page. http://ptolemy.eecs.berkely.edu.