DEVELOPING CROSS-PLATFORM AUDIO AND MUSIC APPLICATIONS WITH THE CLAM FRAMEWORK

Xavier Amatriain

CREATE, Dept. of Music
University of California
Santa Barbara CA 93106
USA
{xavier@create.ucsb.edu}

Pau Arumí

Music Technology Group
Universitat Pompeu Fabra
08003 Barcelona,
Spain
{parumi@iua.upf.es}

ABSTRACT

CLAM is a C++ framework that 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.

In this article we will review the main features included in the framework. We will also present some of the applications that have been developed and can be used by themselves. We will finally introduce the newest tools that have been added in order to use the framework as a rapid prototyping tool.

1. INTRODUCTION

CLAM stands for C++ Library for Audio and Music and it is a full-fledged software framework for research and application development in the audio and music domain. It offers a conceptual model; algorithms for analyzing, synthesizing and transforming audio signals; and tools for handling audio and music streams and creating cross-platform applications.

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 (www.iua.upf.es/mtg) at the Universitat Pompeu Fabra in Barcelona.

But the library is no longer seen as an internal tool for the MTG but rather as a publicly distributed framework licensed under the GPL [12]. CLAM became public and Free in the course of the AGNULA IST European project [8]. Some of the resulting applications as well as the framework itself were included in the Demudi distribution. CLAM is now Free Software and all its code and documentation can be obtained through its web page (www.iua.upf.es/mtg/clam).

The CLAM 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 offers a processing kernel that includes an *infrastructure* and processing and data *repositories*.

In that sense, CLAM is both a *black-box* and a *white-box* framework [22]. 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 to extend the framework components with new processes or data classes.

2. 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* [1]. 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*.

2.1. Processing infrastructure

Processing objects are connected through intermediate channels. These channels are the only mechanism for communicating between Processing objects and with the

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

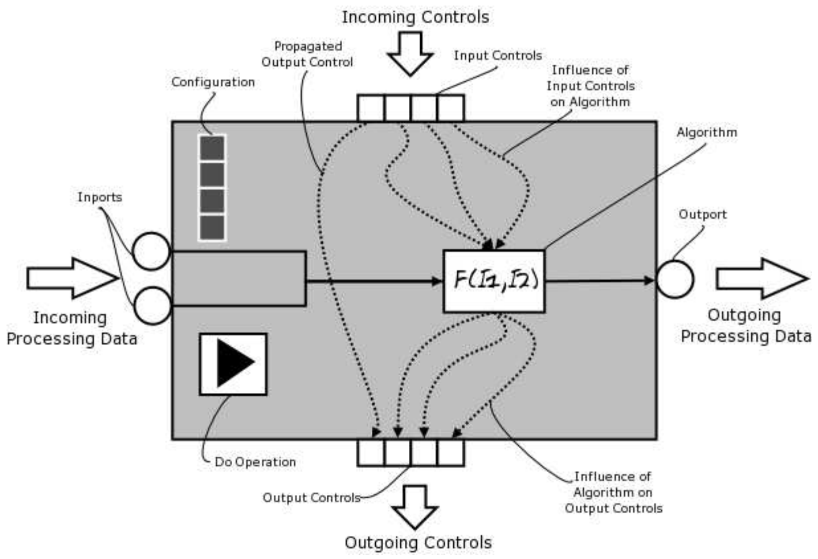


Figure 1. CLAM processing detailed representation

outside world. Messages are enqueued (produced) and dequeued (consumed) in these channels, which act as FIFO queues.

In CLAM we clearly differentiate two kinds of connection mechanisms: *ports* and *controls*. Ports transmit data and have a synchronous data flow nature while controls transmit events and 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 1 is a representation of a CLAM processing object. 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 will receive these control events through various (input) control channels: one for the gain amount, another for the frequency, etc.

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

Processing objects can consume and produce at different rates and consume an arbitrary number of tokens at each firing. Connecting these processing objects 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 ports or not enough space into its output ports.

2.1.1. *Configurations*

Apart from the input controls, a processing object receives another kind of parameter: the configurations.

Configuration parameters, unlike controls, produce expensive or structural changes in the processing. For instance, a configuration parameter may include the number of ports that a processing will have or the numbers of tokens that will be produced in each firing. Therefore, and as opposed to controls that can be received at any time, configurations can only be set into a processing when this is not in running state.

CLAM configurations make use of the framework's data infrastructure and offer services such as automatic built-in persistence or homogeneous interface.

2.1.2. *Static vs. dynamic processing compositions*

When working with large systems we need to be able to group a number of independent processing objects into a larger functional unit that may be treated as a new processing object in itself.

This process, known as composition, can be done in two different ways: *statically* or at compile time, and *dynamically* or at run time (see [9]). Static compositions in CLAM are called Processing Composites while dynamic compositions are called Networks. In both cases inner ports and controls can be *published* to the parent processing.

Choosing between Processing Composites and Networks is a trade-off between boosting efficiency or understandability and flexibility.

But another important difference is that while in a Processing Composite the developer is in charge of handling most of the internal flow and data management, Networks, as we will see in the next paragraphs, can offer an advanced level of automation.

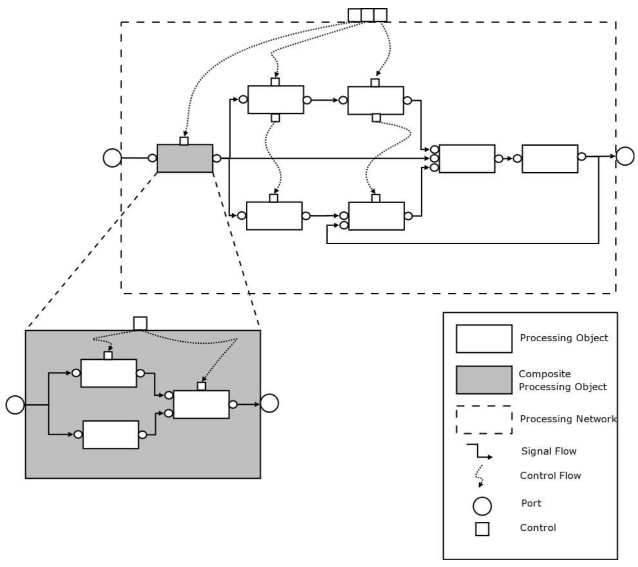


Figure 2. a CLAM processing network

In Processing Networks the instantiation of concrete processing objects is possible by simply passing string identifiers to a *factory*. Static factories are a well documented C++ idiom [1] that make the process of adding or removing processings to the repository as easy as issuing a single line of code in the processing class declaration.

Apart from helping in the instantiation process, the Network class offers interface for connecting the processing objects and, most important, it automatically controls their firing (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 [16, 19] for more details on scheduling dataflow process networks.

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.

3. 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 that act as data containers to be input or output to the processing 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 make use of the data infrastructure and are therefore able to offer services such as a homogeneous interface or built-in automatic XML persistence.

4. TOOLS

Apart from the infrastructure and the repositories, which together make up the CLAM *processing kernel* CLAM also includes a number of tools that can be necessary to build an audio application.

4.1. 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 [14].

4.2. GUI

Just as many frameworks, 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 to 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. The CLAM team, however, aimed at offering a toolkit-independent 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 [10] and the qt framework [24]. 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. . .

4.3. 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.

5. CLAM APPLICATIONS

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 [15], a sax synthesizer; Rappid [23] a real-time processor used in live performances.

5.1. SMS Analysis/Synthesis

At the time CLAM was started the MTG's flagship applications were SMSCommandLine and SMSTools. As a matter of fact one of the main goals when starting CLAM was to develop the substitute for those applications. The SMS Analysis/Synthesis example substitutes those applications and therefore illustrates the core of the research being carried out at the MTG.

The application has three different versions: SMSTools, which has a FLTK graphical user interface; SMSConsole, which is a command-line based version; and SMSBatch, which can be used for batch processing a whole directory. Out of these three it is clearly the graphical version that can find more usages, the other two are only used for very specific problems.

The main goal of the application is to analyze, transform and synthesize back a given sound. For doing so, it uses the Sinusoidal plus Residual model [4]. In order to do so the application has a number of possible inputs:

1. An XML **configuration** file that is used to configure both the analysis and synthesis processes.

2. An SDIF [27] or XML **analysis** file that is the result of a previously performed and stored analysis.

3. A **Transformation score** in XML format. This file includes a list of all transformations that will be applied to the result of the analysis and the configuration for each of the transformations.

Note that all of them can be selected and generated at run-time from the user interface in the SMSTools version.

From these inputs, the application is able to generate the following outputs:

1. An XML or SDIF **Analysis data** file.

2. An XML **Melody** file that is extracted from a monophonic melodic input.

3. **Output sound** separated into three different outputs: global sound, sinusoidal component, and residual component.

Therefore, apart from simply storing the result of the analysis, the input sound can be synthesized back, separating each component: residual, sinusoidal, and the sum of both.

To transform your sound an XML transformation score must be loaded or created using the graphical transformation editor available in SMSTools. Although a repository of SMSTransformations – that includes some such as pitch shifting, time-stretching or morph – new transformations can be implemented and added to the CLAM repository very easily.

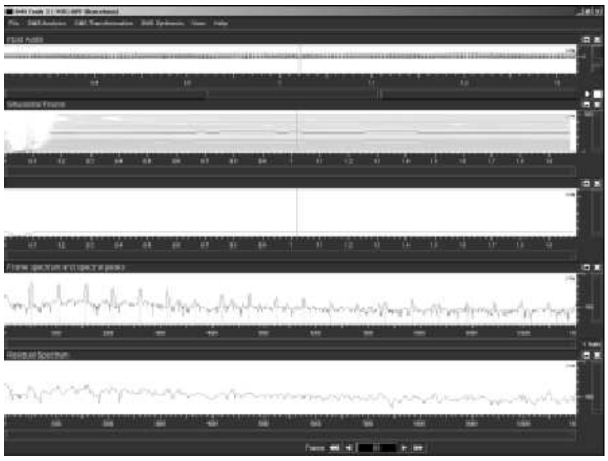


Figure 3. The SMSTools graphical user interface

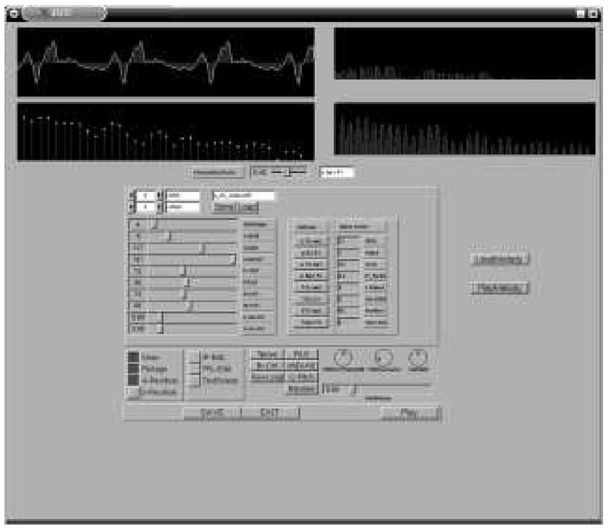


Figure 4. The Salto GUI

5.2. SALTO

SALTO is a software based synthesizer that is also based on the SMS technique. It implements a general architecture for these synthesizers but it is currently only prepared to produce high quality sax and trumpet synthesis. Pre-analyzed data are loaded upon initialization. The synthesizer responds to incoming MIDI data or to musical data stored in an XML file. Output sound can be either stored to disk or streamed to the sound card on real-time. Its GUI allows to modify synthesis parameters on real-time.

The synthesizer uses a database of SDIF files that contain the result of previous SMS analysis. These SDIF files contain spectral analysis samples for the steady part of some notes, the residual and the attack part of the notes. These SDIF files can be viewed, transformed and synthesized with the previously explained SMSTools.

Apart from this SDIF input, SALTO has three other possible inputs: MIDI, an XML Melody, and the GUI. Using MIDI as an input SALTO can be used as a regu-

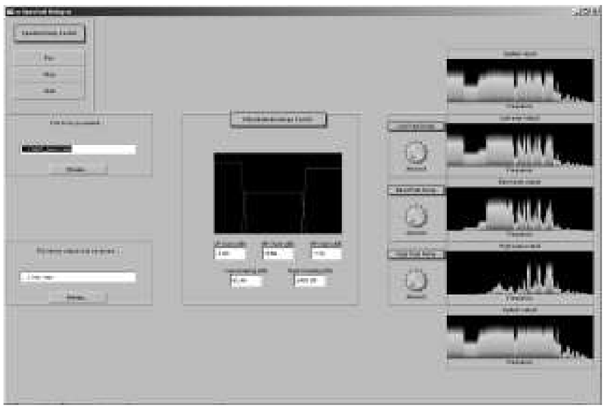


Figure 5. The Spectral Delay

lar MIDI synthesizer on real-time. SALTO is prepared to accept MIDI messages coming from a regular MIDI keyboard or a MIDI breath controller. On the other hand if an XML melody is used as an input this melody is synthesized back. Finally the GUI can be basically used to control the way the synthesis is going to work and to test configurations by generating single notes.

5.3. Spectral Delay

SpectralDelay is also known as CLAM's Dummy Test. In this application it was not important to actually implement an impressive application but rather to show what can be accomplished using the CLAM framework. Special care has been taken on the way things are done and why they are done.

The SpectralDelay implements a delay in the spectral domain, what basically means that the input audio signal can be divided into three bands and each of these bands can be delayed separately, obtaining interesting effects.

The core of the process is an STFT that performs the analysis of the input signal and converts it to the spectral domain. The signal is synthesized using a `SpectralSynthesis` Processing that implements the inverse process. It is transformed – i.e. filtered and delayed – , in between these two steps, in the spectral domain.

The graphical interface depicted in Figure 5 controls the frequency cut-offs and gains of the filters and the delay times of the delays.

5.4. Others

Apart from the main sample applications CLAM has been used in many different projects that are not included in the public version either because the projects themselves have not reached an stable stage or because their results are protected by non-disclosure agreements with third parties. In this section we will outline these other users of CLAM.

Rappid [23] is a testing workbench for the CLAM framework in high demanding situations. The first version

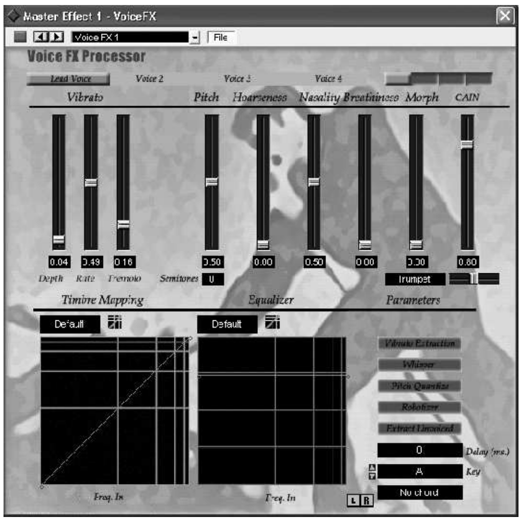


Figure 6. The Vocal Processor

of Rappid implements a quite simple time-domain amplitude modulation algorithm. Any other CLAM based algorithm, though, can be used in its place. The most interesting thing about Rappid is the way that multithreading issues are handled, using a watchdog mechanism.Rappid has been tested in a live-concert situation. Gabriel Brnic used Rappid as a essential part of his composition for harp, viola and tape, presented at the Multiphonies 2002 cycle of concerts in Paris.

The Time Machine project implemented a high quality time stretching algorithm that was later integrated and included in a commercial product. The algorithm uses multi-band processing and works in real-time. It is a clear example of how the core of CLAM processing can be used in isolation as it lacks of any GUI or audio input/output infrastructure.

The Vocal Processor (see figure 6) is a prototype also developed for a third party. It is a VST plug-in for singing voice transformations. It includes transformations such as smart harmonization, hoarseness, whispering or formant change. This prototype was a chance to test CLAM integration into VST API and also to check the efficiency of the framework in highly demanding situations. Most transformations are implemented in the frequency domain and the plug-in must work in real-time, consuming as few resources as possible.

The CUIDADO IST European project [26] was completely developed with CLAM. The focus of the project was on automatic analysis of audio files. In particular rhythmic and melodic descriptions were implemented. The CLAM code was integrated as a binary dll into a commercial product named the Sound Palette. The algorithms and research applications are currently being integrated into the CLAM project and incorporated into standalone sample applications such as the Swinger, an application that computes rhythmic descriptors from a sound file and applying a time-stretching algorithm is able to change the *swing* of the piece.

The Open Drama project was another IST European project that used CLAM extensively. The project focus was on finding new interactive ways to present opera. In particular, a prototype application called MDTools was built to create an MPEG-7 compliant description of a complete opera play.

The AudioClass project aims at building automatic tools for managing large collections of audio effects. Analysis algorithms implemented in CLAM have been integrated and are called from a web application. The results are then added to a large metadata database.

Also CLAM is being used for educational purposes in different ways. On one hand, it is the base for a course on Music and Audio Programming. On the other hand it is the base of many Master Thesis. In this context, it has been used for applications such as Voice-to-MIDI conversion, Timbre Space based synthesis and morph, or song identification. All these results are by definition public and will be integrated into the public repository.

6. CLAM AS A RAPID PROTOTYPING ENVIRONMENT

The latest developments in CLAM have brought *visual building* capabilities into the framework. These allow the user to concentrate on the research of algorithms forgetting about application development. Visual patching is also valuable for rapid application prototyping of applications and audio-plugins.

Acting as the *visual builder*, CLAM has a graphical program called NetworkEditor that allows to generate an application –or at least its processing engine– by graphically connecting objects. Another application called Prototyper acts as the glue between a graphical GUI designing tool (such as qt Designer) and the processing engine defined with the NetworkEditor.

6.1. An example

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

6.1.1. First step: building the processing network

Patching with NetworkEditor is a very intuitive task to do. See Figure 7. We can load the desired processings by dragging them from the left panel of the window. Once in the patching panel, processing objects 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.

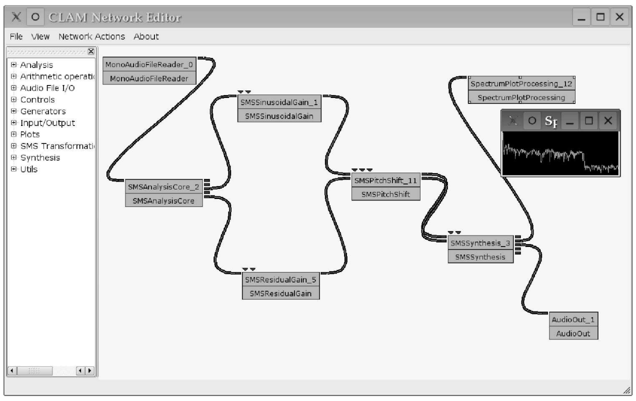
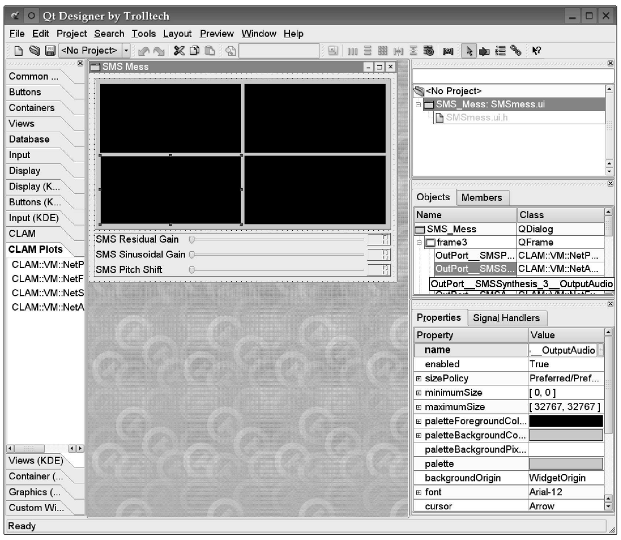


Figure 7. NetworkEditor, the CLAM visual builder



Name	Class
SMS_Mess	QDialog
frame3	QFrame
OutPort__SMSP...	CLAM::VM::NetP...
OutPort__SMSS...	CLAM::VM::NetA...

OutPort__SMSSynthesis_3__OutputAudio

Property	Value
name	__OutputAudio
enabled	True
sizePolicy	Preferred/Pref...
minimumSize	[0, 0]
maximumSize	[32767, 32767]
paletteForegroundCol...	■
paletteBackgroundCo...	▨
paletteBackgroundPix...	
palette	▨
backgroundOrigin	WidgetOrigin
font	Arial-12
cursor	Arrow

Figure 8. The qt Designer tool

Since CLAM ports are typed, not all out-ports are compatible with all in-ports. For example in the Figure 7, the second processing in the chain is called SMSAnalysisCore 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).

Connected to SMSAnalysisCore out-ports we have placed two processings to perform transformations: one for controlling the gain of the sinusoidal component, and another to control the gain of the residual component. The resulting stream of sinusoidal and residual components feeds another processing, SMSPitchShift, which modifies both components performing a pitch shift.

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 SMSAnalysisCore configuration includes the window type and window size parameters among many others.

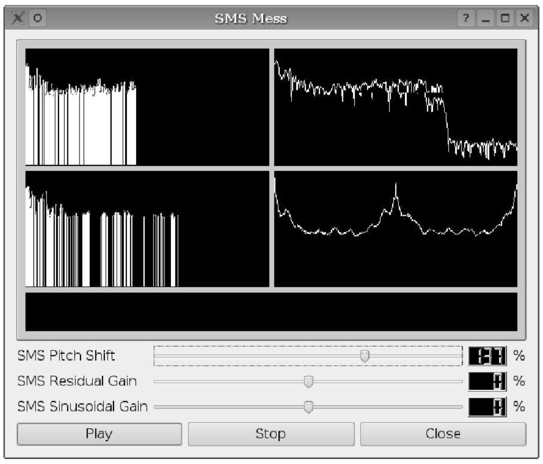


Figure 9. The final running prototype

Another interesting feature of the NetworkEditor is that it allows loading visual plots widgets for examining the data flowing through any out-port. Slider widgets can also be connected to the in-control inlets.

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

6.1.2. Second step: designing the program GUI (Figure 8)

The screen-shot in Figure 8 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 [24].

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 has 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 8 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.

6.1.3. Third step: running the prototype (Figure 9)

Finally we run the prototyper program (see Figure 9). 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 now have a prototype that runs fast C++ compiled code without compiling a single line.

7. IS CLAM DIFFERENT?

Although other audio-related environments exist (see for instance Max/Pd [21], CSL [20], OpenSoundWorld [7], Marsyas [25], SndObj [17] or SuperCollider [18]) there are some important features of our framework that make it somehow different (see [2] for an extensive study and comparison of most of them) :

(1) 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 [13] and C++ idioms [1], good development practices like test-driven development [6] and refactoring [11], as well as constant peer reviewing.

(2) 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.)

(3) 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.

(4) 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)

(5) As stated before, it is *cross-platform*

(6) The project is licensed under the *GPL* terms and conditions.

(7) The framework can be used either as a regular C++ *library* or as a *prototyping tool*.

8. CONCLUSIONS

CLAM has already been presented in other conferences like the OOPSLA'02 [5, 3] but since then, a lot of progress have been made in different directions, and specially in making the framework more *black-box* and adding *visual builder* tools.

CLAM has proven useful in many applications and is becoming more and more easy to use, and so, we expect new projects to begin using the framework even before it has reached its first stable 1.0 release.

9. 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.

10. REFERENCES

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

[2] X. Amatriain. *An Object-Oriented Metamodel for Digital Signal Processing*. Universitat Pompeu Fabra, 2004.

[3] X. Amatriain, P. Arumí, and M. Ramírez. 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, 2002. ACM.

[4] X. Amatriain, J. Bonada, A. Loscos, and X. Serra. *DAFX: Digital Audio Effects (Udo Zölzer ed.)*, chapter Spectral Processing, pages 373–438. John Wiley and Sons, Ltd., 2002.

[5] X. Amatriain, M. de Boer, E. Robledo, and D. Garcia. 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, 2002. ACM.

[6] K Beck. *Test Driven Development by Example*. Addison-Wesley, 2000.

[7] A. Chaudhary, A. Freed, and M. Wright. An Open Architecture for Real-Time Audio Processing Software. In *Proceedings of the Audio Engineering Society 107th Convention*, 1999.

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

[9] R.B. Dannenberg. Combining visual and textual representations for flexible interactive audio signal processing. In *Proceedings of the 2004 International Computer Music Conferenc (ICMC'04)*, 2004. in press.

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

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

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

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

[14] D. Garcia and X. Amatrian. 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, 2001.

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

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

[17] V. Lazzarini. Sound Processing with the SndObj Library: An Overview. In *Proceedings of the 4th International Conference on Digital Audio Effects (DAFX '01)*, 2001.

[18] J. McCartney. Rethinking the Computer Music Language: SuperCollider. *Computer Music Journal*, 26(4):61–68, 2002.

[19] T. M. Parks. *Bounded Schedule of Process Networks*. PhD thesis, University of California at Berkeley, 1995.

[20] S. T. Pope and C. Ramakrishnan. The Create Signal Library (”Sizzle”): Design, Issues and Applications. In *Proceedings of the 2003 International Computer Music Conference (ICMC '03)*, 2003.

[21] M. Puckette. Max at Seventeen. *Computer Music Journal*, 26(4):31–43, 2002.

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

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

[24] Trolltech. Qt homepage by trolltech, 2004. http://www.trolltech.com.

[25] G. Tzanetakis and P. Cook. *Audio Information Retrieval using Marsyas*. Kluewe Academic Publisher, 2002.

[26] H. Vinet, P. Herrera, and F. Pachet. The cuidado project. In *Proceedings of the 3rd International Symposium on Music Information Retrieval (ISMIR 2002)*, 2002.

[27] M. Wright. Audio applications of the sound description interchange format. In *Proceedings of the 107th AES Convention*, 1999.