Visual prototyping of audio applications

David Garcia and **Pau Arumi**
IUA, Universitat Pompeu Fabra
Psg. de Circumval.lacio, 8. 08003 Barcelona, Spain
{dgarcia,parumi}@iua.upf.edu

Xavier Amatrinain
CREATE, Univ. of California Santa Barbara
Santa Barbara, CA, USA
xavier@create.ucsb.edu

Abstract

This article describes an architecture that enables visual prototyping of real-time audio applications and plugins. Visual prototyping means that a developer can build a working application, including user interface and processing core, just by assembling elements together and changing their properties in a visual way. The article addresses the problem of having to bind interactive user interface to a real-time processing core, when both are defined dynamically with an extensible set of components, allowing bidirectional communication of arbitrary types of data and still fulfiling real-time requirements of audio applications. It also introduces some design patterns that have enabled its implementation.

Keywords

Audio applications, Visual prototyping, interfaces, GUI, Frameworks

1 Introduction

Having a proper development environment is something that may increase development productivity. Development frameworks offer system models that enables system development dealing with concepts of the target domain. Eventually, they provide visual building tools which also boost the development productivity. In the audio and music domain, the approach of modeling systems using visual data-flow tools has been widely and successfully used in system such as PD [1], Marsyas [2], Open Sound World [3] and CLAM [4]. But, such environments are used to build just processing algorithms, not full applications ready for the public. A full application would need further development work addressing the user interface and the application work-flow.

User interface design is supported by existing toolboxes and visual prototyping tools. They provide a similar flexibility than the one data-flow tools provide to build the processing core. Examples of such environments which are freely available are Qt Designer [5], Fltk Fluid [6] or Gtk's Glade [7]. But such tools just solve the composition of graphical components into a layout and limited reactivity. They still do not address a lot of low level programming that is needed to solve the typical problems that an audio application has. Those problems are mostly related to the communication between the processing core and the user interface.

This article describes an architecture that addresses this gap and enables fully visual building of real-time audio processing applications by combining visual data-flow tools and visual GUI design tools.

Section 2 describes the target applications to be built with the architecture. Section 3 describes the development work flow that the architecture offers and the main tools involved. Section 4 goes deeper on the key part of the architecture, the run-time engine, and describes how it solves different problems it faces. Section 5 describes some audio related design patters that enable the actual implementation of the architecture. Those patterns are part of a bigger pattern catalog which is briefly described on section 6. And, finally, section 7 explains the current status and achievements and the future lines both for the architecture implementation and the pattern catalog.

2 Target applications

The set of applications the architecture is able to visually build includes real-time audio processing applications, which have a relatively simple application logic. That is synthesizers, real-time music analyzers (figure 1) and audio effects and plugins (figure 2).

So application logic should support just starting and stopping the processing algorithm, configuring it, connecting it to the system streams (audio from devices, audio servers, plugin hosts, MIDI, files, OSC...), visualizing the inner processing data and controlling some algorithm parameters while running.

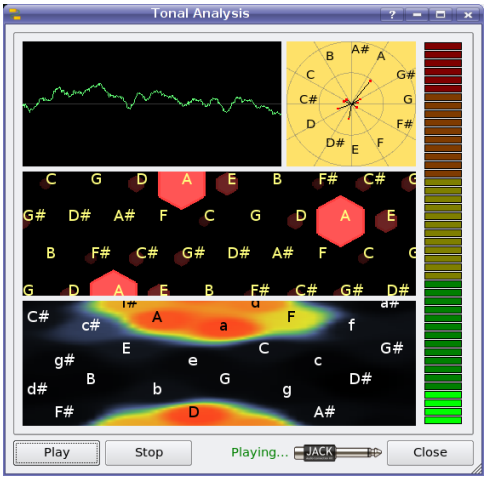

Figure 1: A sample of audio analysis application: Tonal analysis with chord extraction

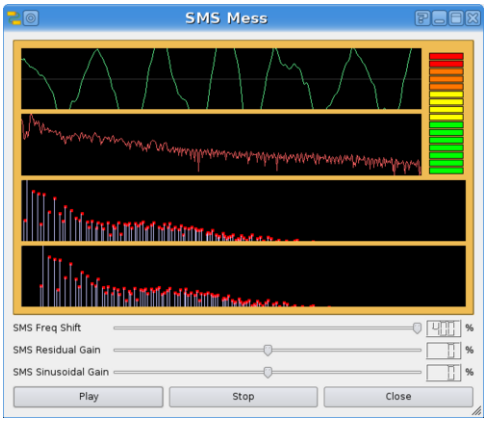

Figure 2: A sample of audio effect application: JACK enabled SMS transposition

Given those limitations, the defined architecture does not claim to visually build every kind of audio application. For example, audio authoring tools, which have a more complex application logic, would be out of the scope, although the architecture would help to build important parts of such applications.

The architecture will provide the following features:

- Communication of any kind of data and control objects between GUI and processing core (not just audio buffers)
- The prototype could be embedded in a wider application with a minimal effort
- Plugin extensibility for processing units, for graphical elements which provide data visualization and control sending, and for system connectivity backends (JACK, ALSA, PORTAUDIO, LADSPA, VST, AudioUnit...)

3 The big picture

The proposed architecture (figure 3) has three main components: A visual tool to define the audio processing core, a visual tool to define the user interface and a third element, the run-time engine, that dynamically builds definitions coming from both tools, relates them and manages the application logic. We implemented this architecture using some existing tools. We are using CLAM NetworkEditor as the audio processing visual builder, and Trolltech's Qt[1] Designer as the user interface definition tool. Both Qt Designer and CLAM NetworkEditor provide similar capabilities in each domain, user interface and audio processing, which can be exploited by the run-time engine.

Qt Designer can be used to define user interfaces by combining several widgets. The set of widget is not limited; developers may define new ones that can be added to the visual tool as plugins. Figure 4 shows a Qt Designer session designing the interface for an audio application, which uses some audio related widgets provided by CLAM as a widgets plugin. Note that other audio related widgets are available on the left panel list.

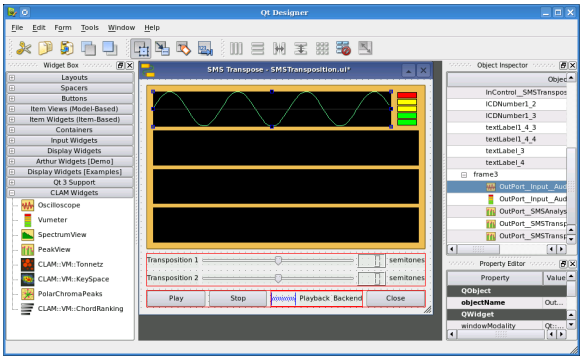

Figure 4: Qt Designer tool editing the interface of an audio application.

Interface definitions are stored as XML files with the ".ui" extension. Ui files can be rendered as source code or directly loaded by the application at run-time. Applications may, also, discover the structure of a run-time instatiated user interface by using introspection capabilities.

[1] We are using Qt version 4.2

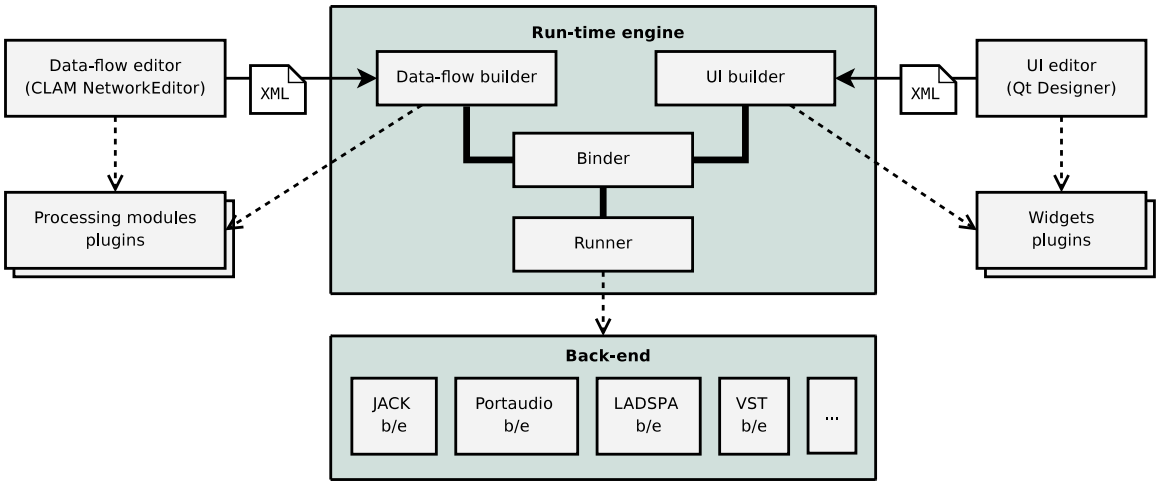

Figure 3: Visual prototyping architecture

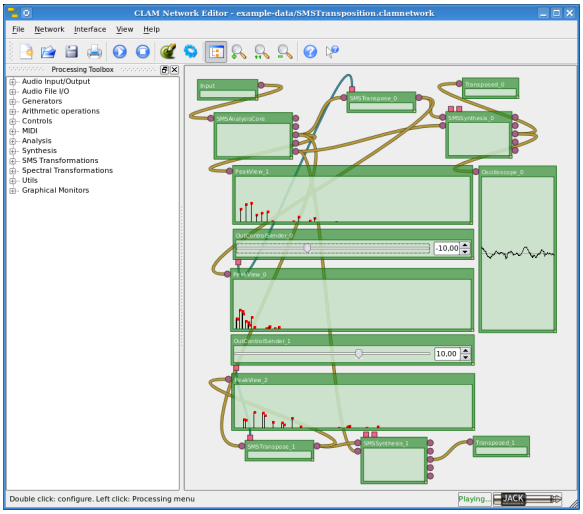

Figure 5: The processing core of an application built with the CLAM Network Editor

Analogously, CLAM Network Editor allows to visually combine several processing modules into a processing network definition. The set of processing modules in the CLAM framework is also extensible with plugin libraries. Processing network definitions can be stored as XML files that can be loaded later by applications in run-time. And, finally the CLAM framework also provides introspection so a loader application may discover the structure of a run-time loaded network.

4 Run-time engine

Just by having a data flow visual tool and a visual interface designer, we should still do some programming to glue it all and launch the application. The purpose of the run-time engine, which is called Prototyper in our implementation, is to figure out which should be that code and supply it. Next, we enumerate the problems that the run-time engine faces and how it solves them.

4.1 Dynamic building

Both component structures, the audio processing network and the user interface, have to be built up dynamically in run-time from an XML definition. CLAM and Qt frameworks supports dynamic building mostly because they provide object factories. Object factories provide objects given a type identifier. Object factories are a very well known general design pattern and it is covered better by E. Gamma et al. [8] and implementation details are further covered by Alexandrescu [9].

Because we want interface and processing components to be expandable, the factories should be able to incorporate new objects defined by plugin libraries. To enable the creation of a certain type of object, the class provider must register it on the factory at plugin initialization.

4.2 Relating objects

The run-time engine must relate components of both structures. For example, the spectrum view on the SMS Transposition application (second panel on figure 2) needs to periodically access spectrum data flowing by a given port of the processing network. The run-time engine firstly has to identify which components, according to the developer's intent. Then guess

whether the connection is able to be done: spectrum data can not be viewed by an spectral peaks view. And then, perform the connection, all that without the run-time engine knowing nothing about spectra and spectral peaks.

The proposed architecture uses properties such the component name to relate components on each side. Then components are located by using introspection capabilities on each side framework.

Once located, the run-time engine must assure that the components are compatible and connect them. The run-time engine is not aware of the types of data that connected objects will handle, we deal that by applying the Typed Connections design pattern explained in section 5.1. In short, this design pattern allows to establish a type dependant connection construct between two components without the connector maker knowing the types and still be type safe.

4.3 Thread safe communication in real-time

One of the main issues that typically need extra effort while programming is multi-threading. In real-time audio applications based on a data-flow graph, the processing core is executed in a high priority thread while the rest of the application is executed in a normal priority one following the Out-of-band and In-band partition pattern [10]. Being in different threads, safe communication is needed, but traditional mechanisms for concurrent access are blocking and the processing thread can not be blocked. Thus, new solutions, as the one proposed by the Port Monitor pattern in section 5.2, are needed.

4.4 System back-end

Target applications, being real-time processing, have smaller application logic than others but it still has application logic to define. Most of the application logic is coupled to the sink and sources for audio data and control events. Audio sources and sinks depend on the context of the application: JACK, ALSA, ASIO, DirectSound, LADSPA... So the way of dealing with threading, callbacks, and assigning input and outputs is different in each case. The architectural solution for that has been providing backend plugins to deal with this issues.

This also transcends to the user interface as sometimes the application may let the user to choose the concrete audio sink or source, and even choose the audio backend.

5 Enabling design patterns

Probably the two most complex technical problems involved in implementing this visual prototyping architecture are the following: Managing connections when port types are not limited but extensible; and monitoring data being processed in the high-priority thread from a UI thread. Interestingly, similar problems to these are often found in other contexts. However, their solutions are very similar, so they can be generalized and formalized as a design pattern.

A software pattern is a proved solution to a recurring problem. It pays special attention to the context in which is applicable, to the competing "forces" it needs to balance, and the teaching component on the implications of its application. Patterns provide a convenient way to formalize and reuse design experience. However, neither data-flow systems nor other audio audio-related areas have yet received many attention on domain specific patterns.

The next sections describes Port Monitor and Typed Connections two patterns for those mentioned problems.

5.1 PATTERN: Typed Connections

Context

Most simple audio applications have a single type of token: the sample or the sample buffer. But more elaborated processing applications must manage some other kinds of tokens such as spectra, spectral peaks, MFCC's, MIDI... You may not even want to limit the supported types. The same applies to events channels, we could limit them to floating point types but we may use structured events controls like the ones OSC [11] allows.

Heterogeneous data could be handled in a generic way (common abstract class, void pointers...) but this adds a dynamic type handling overhead to modules. Module programmers should have to deal with this complexity and this is not desirable. It is better to directly provide them the proper token type. Besides that, coupling the communication channel between modules with the actual token type is good because this eases the channel internal buffers management.

But using typed connections may imply that the entity that handles the connections should deal with all the possible types. This could imply, at least, that the connection entity would have a maintainability problem. And it could even be unfeasible to manage when the set of

those token types is not known at compilation time, but at run-time, for example, when we use plugins.

Problem

Connectable entities communicate typed tokens but token types are not limited. Thus, how can a connection maker do typed connections without knowing the types?

Forces

- Process needs to be very efficient and avoid dynamic type checking and handling.
- Connections are done in run-time by the user, so they can mismatch the token type.
- Dynamic type handling is a complex and error prone programming task, thus, placing it on the connection infrastructure is preferable than placing it on concrete modules implementation.
- Token buffering among modules can be implemented in a wiser way by knowing the concrete token type rather than just knowing an abstract base class.
- The set of token types evolves and grows.
- A connection maker coupled to the evolving set of types is a maintenance workhorse.
- A type could be added in run time.

Solution

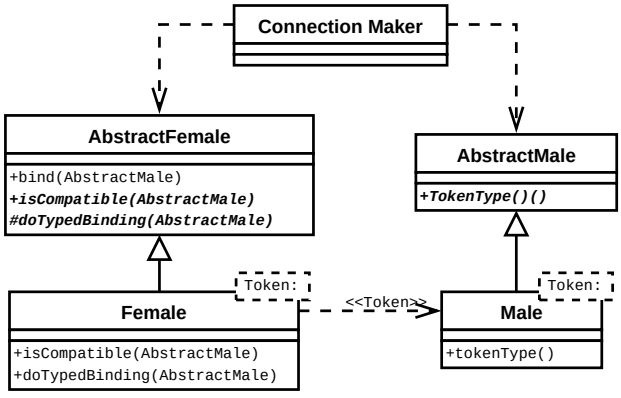

Figure 6: Class diagram of a cannonical solution of Typed Connections

Split complementary ports interfaces into an abstract level, which is independent of the token-type, and a derived level that is coupled to the token-type. The class diagram of this solution is shown in figure 6.

Let the connection maker set the connections throught the generic interface, while the connected entities use the token-type coupled interface to communicate each other. Access to typed tokens from the concrete module implementations using the typed interface.

Use run-time type checks when modules get connected (*binding time*) to get sure that connected ports types are compatible, and, once they are correctly connected (*processing time*), rely just on compile-time type checks.

To do that, the generic connection method on the abstract interface (`bind`) should delegate the dynamic type checking to abstract methods (`isCompatible`, `typeId`) implemented on token-type coupled classes.

Consequences

By applying the solution, the connection maker is not coupled to token types. Just concrete modules are coupled to the token types they use.

Type safety is assured by checking the dynamic type on binding time and relying on compile time type checks during processing time. So this is both efficient and safe.

Because both sides on the connection know the token type, buffering structures can deal with tokens in a wiser way when doing allocations, initializations, copies, etc.

Concrete modules just access to the static typed tokens. So, no dynamic type handling is needed.

Besides the static type, connection checking gives the ability to do extra checks on the connecting entities such as semantic type information. For example, implementations of the bind method could check that the size and scale of audio spectra match.

Related Patterns

This pattern enriches Multi-rate Stream Ports and Event Ports, and can be also useful for the binding of the visualization and the Port Monitor.

The proposed implementation of Typed Connections uses the Template Method [8] to call the concrete binding method from the generic interface.

Examples

OSW [3] uses Typed Connections to allow incorporating custom data types.

The CLAM framework uses this pattern notably on several pluggable pairs such as in and out ports and in and out controls, which are, in addition, examples of the Multi-rate Stream Ports and Event Ports patterns.

But the Typed connection pattern in CLAM is not limited to port like pairs. For example,

CLAM implements sound descriptors extractor modules which have ports directly connected to a descriptor container which stores them. The extractor and the container are type coupled but the connections are done as described in a configuration file, so handling generic typed connections is needed.

The Music Annotator [12] is a recent application which provides another example of non-port-like use of Typed Connections. Most of its views are type coupled and they are mostly plugins. Data to be visualized is read from an storage like the one before. A design based on the Typed Connection pattern is used in order to know which data on the schema is available to be viewed with each vista so that users can attach any view to any type compatible attribute on the storage.

5.2 PATTERN: Port Monitors

Context

Some audio applications need to show a graphical representation of tokens that are being produced by some module out-port. While the visualization needs just to be fluid, the processing has real-time requirements. This normally requires splitting visualization and processing into different threads, where the processing thread has real-time requirements and is a high priority scheduled thread. But because the non real-time monitoring should access to the processing thread tokens some concurrency handling is needed and this often implies locking.

Problem

We need to graphically monitor tokens being processed. How to do it without locking the real-time processing while keeping the visualization fluid?

Forces

- The processing has real-time requirements (ie. audio)
- Visualizations must be fluid; that means that it should visualize on time and often but it may skip tokens
- Just the processing is not filling all the computation time

Solution

The solution is to encapsulate concurrency in a special kind of process module, the *Port monitor*, that is connected to the monitored out-port. *Port monitors* offers the visualization thread an special interface to access tokens in a thread safe way.

In order to manage the concurrency avoiding the processing to stall, the *Port monitor* uses two alternated buffers to copy tokens. In a given time, one of them is the writing one and the other is the reading one. The *Port monitor* state includes a flag that indicates which buffer is the writing one. The *Port monitor* execution starts by switching the writing buffer and copying the current token there. Any access from the visualization thread locks the buffer switching flag. Port execution uses a *try lock* to switch the buffer, so, the process thread is not being blocked, it is just writing on the same buffer while the visualization holds the lock.

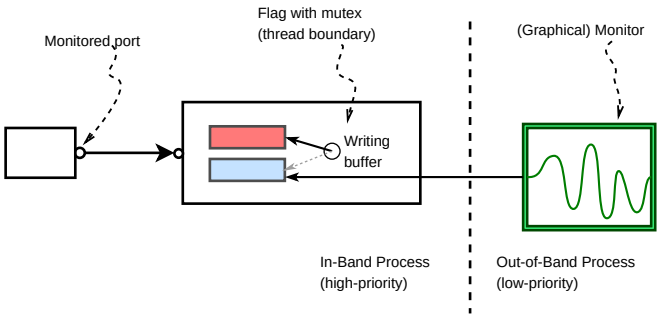

Figure 7: A port monitor with its switching two buffers

Consequences

Applying this pattern we minimize the blocking effect of concurrent access on two fronts. On one side, the processing thread never blocks. On the other, the blocking time of the visualization thread is very reduced, due that it only lasts a single flag switching.

Any way, the visualization thread may suffer starvation risk. Not because the visualization thread will be blocked but because it may be reading always the same buffer. That may happen when every time the processing thread tries to switch the buffers, the visualization is blocking. This effect is not that critical and can be avoided by minimizing the time the visualization thread is accessing tokens, for example, by copying them and release.

Another issue with this pattern is how to monitor not a single token but a window of tokens. For example, if we want to visualize a sonogram (a color map representing spectra along the time) where each token is a single spectrum. The simplest solution, without any modification on the previous monitor is to do the buffering on the visualizer and pick samples

at monitoring time. This implies that some tokens will be skipped on the visualization, but, for some uses, this is a valid solution.

Related Patterns

Port Monitor is a refinement of Out-of-band and In-band Partition pattern [10]. Data flowing out of a port belongs to the In-band partition, while the monitoring entity (for example a graphical widget) is located in the out-of-band partition.

It is very similar to the Ordered Locking real-time pattern [13]. Ordered Locking ensures that deadlock can not occur, preventing circular waiting. The main difference is in their purpose: *Port Monitor* allows communicate two band partitions with different requirements.

Examples

The CLAM Network Editor [14] is a visual builder for CLAM that uses Port Monitor to visualize stream data in patch boxes. The same approach is used for the companion utility, the Prototyper, which dynamically binds defined networks with a QT designer interface.

The Music Annotator also uses the concurrency handling aspect of Port Monitor although it is not based on modules and ports but in sliding window storage.

6 Growing a data-flow pattern language for audio

Typed Connections and Port Monitor patterns are part of a broader catalog [15], with 10 patterns that address recurrent problems in data-flow audio systems and which is expected to grow with new patterns.

Some patterns of this catalog are very high-level, like Semantic Ports and Driver Ports, while other are much focused on implementation issues, like Phantom Buffer). Although the catalog is not domain complete, it could be considered a *pattern language* because each pattern references higher-level patterns describing the context in which it can be applied, and lower-level patterns that could be used after the current one, to further refine the solution. These relations form a hierarchical structure drawn in figure 8. The arcs between patterns mean "enables" relations: introducing a pattern in the system enables other patterns to be used.

This pattern catalog shows how to approach the development of a complete data-flow system in an evolutionary fashion without the need to do *big up-front design*. The patterns at the top of the hierarchy suggest that you start with high

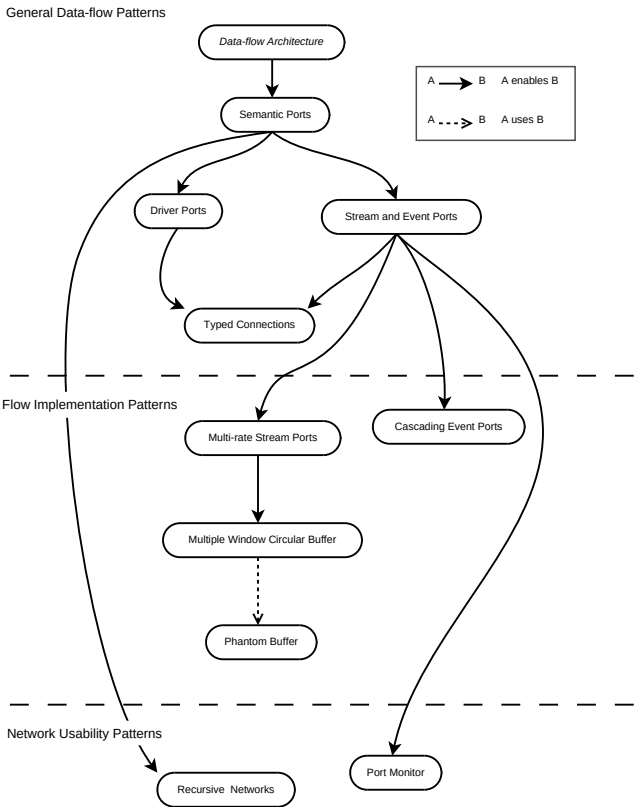

Figure 8: The audio data-flow pattern language. High-level patterns are on the top.

level decisions, driven by questions like: "do all ports drive the module execution or not?" and "do you have to deal only with stream flow or also with event flow?" It might also happen that at some point you will need different token types. Then you'll have to decide "does ports need to be strongly typed while connectable by the user?", or "does the stream ports needs to consume and produce different block sizes?", and so on. On each decision that introduces more features and complexity you face a recurrent problem that is addressed by one pattern in the language.

The main limitation of this pattern language is that do not cover all the typical problems in its domain. Some features of existing data-flow systems proposes recurring problems that should be incorporate into the language. Those include: communication of modules running in different processes, module firing scheduling, integration of heterogeneous (push based vs pull based) systems, ports hand-shacking for meta-data propagation, etc.

The architecture for visual prototyping pre-

sented in this article also contains many recurring problems that could be generalized as patterns. However, our opinion is that more application examples are needed for those patterns to mature.

7 Conclusions

The presented architecture has been already implemented as free software within the CLAM framework and it is available for download as source code or as binary for several platforms. Several already built applications are provided such the ones shown on the screen captures in this article.

Using this architecture, one can build audio applications in few minutes, and developers may concentrate on the development of novel components. Still some work is needed on the implementation of connection classes and back-ends so that they could be provided also as plug-ins.

This article presents some ideas that we hope could help on improving the audio software development by offering an architecture that enables visual development of full applications, a ready to use implementation of such architecture within the CLAM framework, and two related design patterns for a further reuse of design experience.

8 Acknowledgements

Authors of this paper would like to thank Dietmar Schuetz for being our mentor in pattern writing, encouraging us to improve the patterns again and again. We are very grateful to Ralph Johnson, a member of the Gang of Four, who provided insightful feedback and courage to keep our effort on data-flow patterns, during the PLoP workshops. We also thank contributions from past developers and signal-processing experts at the MTG lab. Josep Blat, from the UPF, have provided great support for the CLAM project. This work has been funded by UPF scholarships and by a grant from the STSI division of the Catalan Government. We are also indebted with the Trolltech crew and the Linux Audio Community for giving us the chance to build upon their work.

References

[1] M. Puckette, "Pure Data: Another Integrated Computer Music Environment," in *Proceedings of the Second Intercollege Computer Music Concerts*, Tachikawa, 1996, pp. 37–41.

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

[3] 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.

[4] Clam website. [Online]. Available: http://www.iua.upf.es/mtg/clam

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

[6] (2006, Dec.) The fast light toolkit (fltk) homepage. [Online]. Available: http://www.fltk.org

[7] Glade home page. [Online]. Available: http://glade.gnome.org

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

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

[10] D. A. Manolescu, "A Dataflow Pattern Language," in *Proceedings of the 4th Pattern Languages of Programming Conference*, 1997.

[11] M. Wright, "Implementation and Performance Issues with Open Sound Control," in *Proceedings of the 1998 International Computer Music Conference (ICMC '98)*. Computer Music Association, 1998.

[12] X. Amatriain, J. Massaguer, D. Garcia, and I. Mosquera, "The clam annotator: A cross-platform audio descriptors editing tool," in *Proceedings of 6th International Conference on Music Information Retrieval*, London, UK, 2005.

[13] B. P. Doublass, *Real-Time Design Patterns*. Addison-Wesley, 2003.

[14] X. Amatriain and A. P., "Developing cross-platform audio and music applications with the clam framework," in *Proceedings of the 2005 International Computer Music Conferenc (ICMC'05)*, 2005, in press.

[15] P. Arumí, D. Garcia, and X. Amatrian, "A data-flow pattern catalog for sound and music computing," in *Pattern Language of Programming PLoP 2006*, Oct. 2006.