Multimedia Systems (2008) 14:15–32
DOI 10.1007/s00530-007-0109-6

REGULAR PAPER

A framework for efficient and rapid development of cross-platform audio applications

Xavier Amatriain · Pau Arumi · David Garcia

Published online: 6 November 2007

Abstract In this article, we present CLAM, a C++ software framework, that offers a complete development and research platform for the audio and music domain. It offers an abstract model for audio systems and includes a repository of processing algorithms and data types as well as all the necessary tools for audio and control input/output. The framework offers tools that enable the exploitation of all these features to easily build cross-platform applications or rapid prototypes for media processing algorithms and systems. Furthermore, included ready-to-use applications can be used for tasks such as audio analysis/synthesis, plug-in development, feature extraction or metadata annotation. CLAM represents a step forward over other similar existing environments in the multimedia domain. Nevertheless, it also shares models and constructs with many of those. These commonalities are expressed in the form of a *metamodel* for multimedia processing systems and a design pattern language.

Keywords Software frameworks · Multimedia · Metamodels · Design patterns · Audio processing · Rapid prototyping

X. Amatriain (✉)
University of California Santa Barbara, Santa Barbara, CA, USA
e-mail: xavier@create.ucsb.edu

Present Address:
X. Amatriain
Telefonica R&D, Via Augusta 177, 08021 Barcelona, Spain
e-mail: xar@tid.es

P. Arumi · D. Garcia
Universitat Pompeu Fabra, Barcelona, Spain
e-mail: parumi@iua.upf.edu

D. Garcia
e-mail: dgarcia@iua.upf.edu

1 Introduction

The history of software frameworks is very much related to the evolution of the multimedia field itself. Many of the most successful and well-known examples of software frameworks deal with graphics, image or multimedia.[1] Although probably less known, the audio and music fields also have a long tradition of similar development tools. And it is in this context where we find CLAM, a framework that recently received the 2006 ACM Best Open Source Multimedia Software award.

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 with applicability also to the broader multimedia 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 CLAM framework is cross platform. All the code is ANSI C++ and it is regularly compiled under GNU/Linux, Windows and Mac OSX using FLOSS (Free Libre Open Source Software) tools, such as automatic integrated building/testing/versioning systems, and agile practices such Test-Driven Development.

Although parts of the framework have already been presented in previous articles (see [4] and [8], for instance) in this article we will give a thorough overview of the framework when it has now reached its first stable regime with the publication of the 1.0 release.

[1] The first object-oriented frameworks to be considered as such are the MVC (model view controller) for Smalltalk [13] and the MacApp for Apple applications [52]. Other important frameworks from this initial phase were ET++ [50] and interviews. Most of these seminal frameworks were related to graphics or user interfaces.

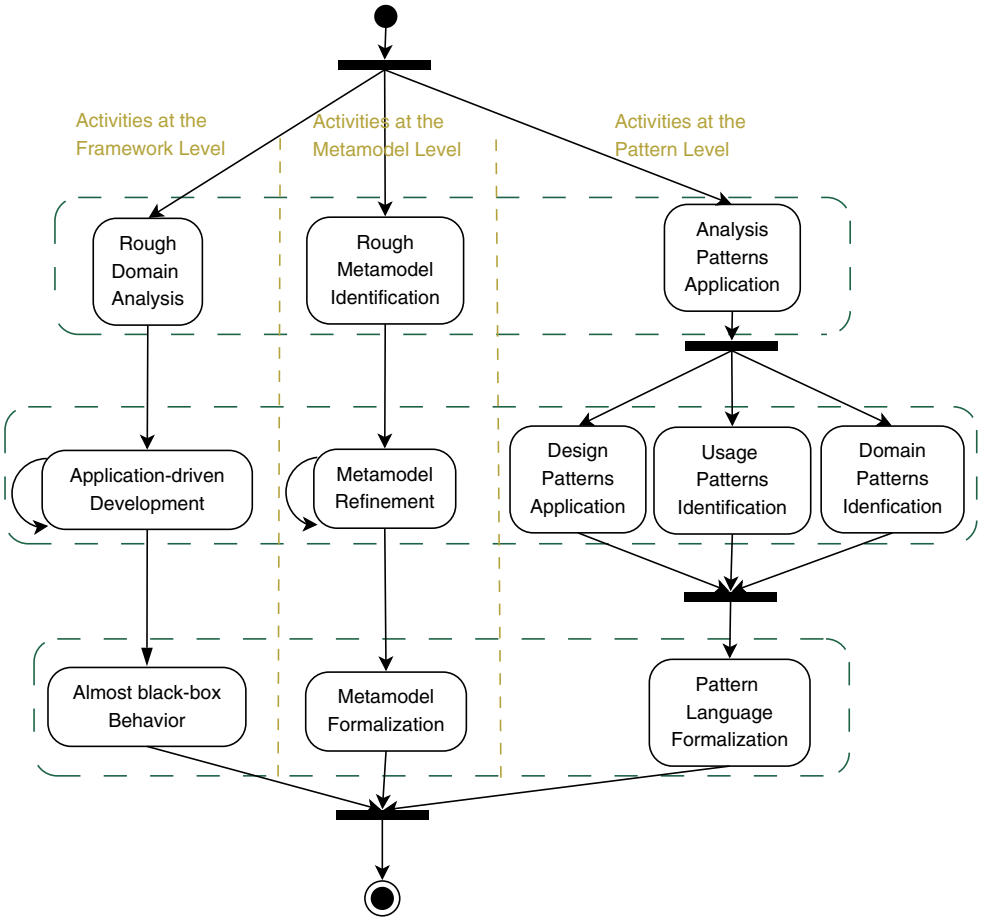

Fig. 1 CLAM development process and related activities

We will explain the different CLAM components and the applications included in the framework in Sect. 3. In Sect. 4, we will also explain the rapid prototyping features that have been added recently and already constitute one of its major assets.

Because CLAM cannot be considered in isolation in Sect. 5, we present a brief summary of a more extensive review of related environments that has taken place during the design process.

In this sense, the framework is not only valuable for its features but also for other outputs of the process that can be considered as reusable components and approaches for the multimedia field. The process of designing CLAM generated reusable concepts and ideas that are formalized in the form of a general purpose domain-specific metamodel and a pattern language both of which are outlined in the next section.

2 Metamodels and patterns

During the CLAM development process several parallel activities have taken place (see Fig. 1). While some seeked the objective of having a more usable framework, others dealt with also coming up with the appropriate abstractions and reusable constructs. In this section, we will focus on the metamodel that has been abstracted and the recurring design patterns that have been identified. Most of these ideas, although a result of the CLAM process itself, are validated by their presence in many other multimedia frameworks and environments.

2.1 4 MPS

The object-oriented metamodel[2] for multimedia processing systems, 4 MPS for short, provides the conceptual framework (metamodel) for a hierarchy of models of media processing systems in an effective and general way. The metamodel is not only an abstraction of many ideas found in the CLAM framework but also the result of an extensive review of similar frameworks (see Sect. 5) and collaborations with their authors. Therefore, the metamodel reflects ideas and concepts that are not only present in CLAM but in many similar environments. Although initially derived for the audio and music domains, it presents a comprehensive conceptual framework for media signal processing applications. In this

[2] The word *metamodel* is here understood as a "model of a family of related models", see [3] for a thorough discussion on the use of metamodels and how *frameworks* generate them.

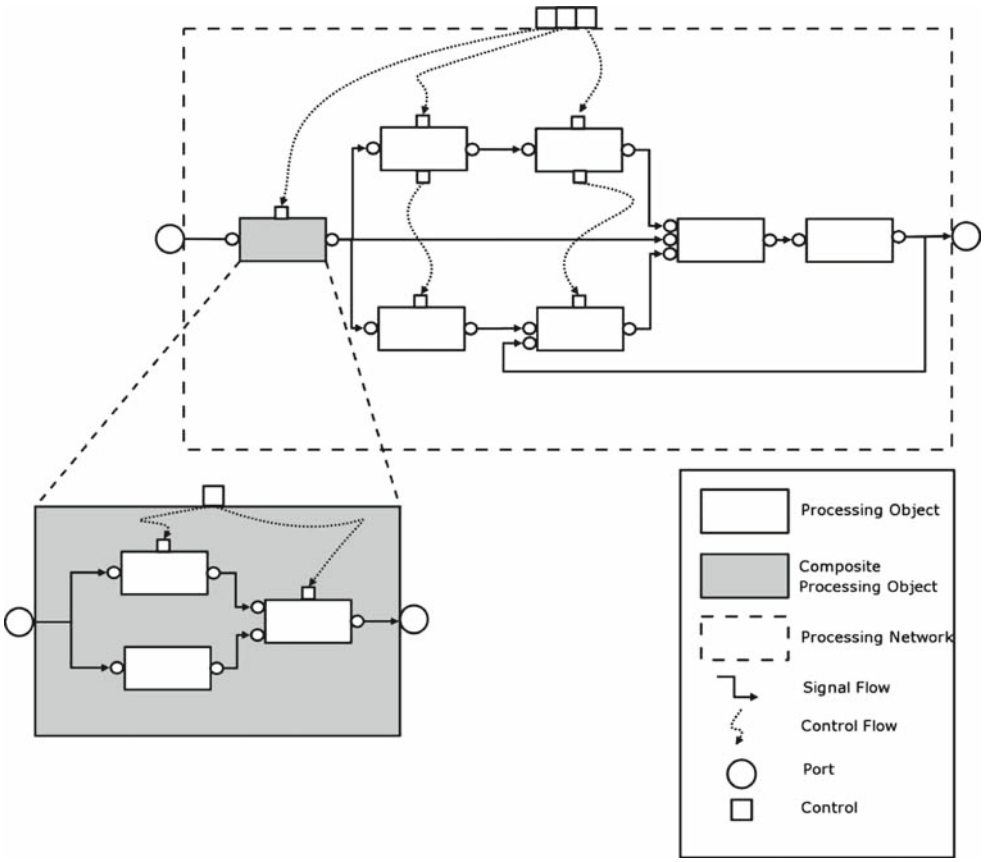

Fig. 2 Graphical model of a 4-MPS processing network. Processing objects are connected through ports and controls. Horizontal left-to-right connections represents the synchronous signal flow while vertical top-to-bottom connections represent asynchronous control connections

section, we provide a brief outline of the metamodel, see [5] for a more detailed description.

The 4 MPS metamodel is based on a classification of signal processing objects into two categories: *Processing* objects that operate on data and control, and *Data* objects that passively hold media content. Processing objects encapsulate a process or algorithm; they include support for synchronous data processing and asynchronous event-driven control as well as a configuration mechanism and an explicit life cycle state model. On the other hand, data objects offer a homogeneous interface to media data, and support for metaobject-like facilities such as reflection and serialization.

Although the metamodel clearly distinguishes between two different kinds of objects the managing of data constructs can be almost transparent for the user. Therefore, we can describe a 4-MPS system as a set of processing objects connected in graphs called *Networks* (see Fig. 2).

Because of this the metamodel can be expressed in the language of graphical models of computation as a *context-aware dataflow network* (see [30] and [48]) and different properties of the systems can be derived in this way.

Figure 3 is a representation of a 4-MPS processing object. Processing objects are connected through channels. Channels are usually transparent to the user that should manage networks by simply connecting ports. However, they are more than a simple communication mechanism as they act as FIFO queues in which messages are enqueued (produced) and dequeued (consumed).

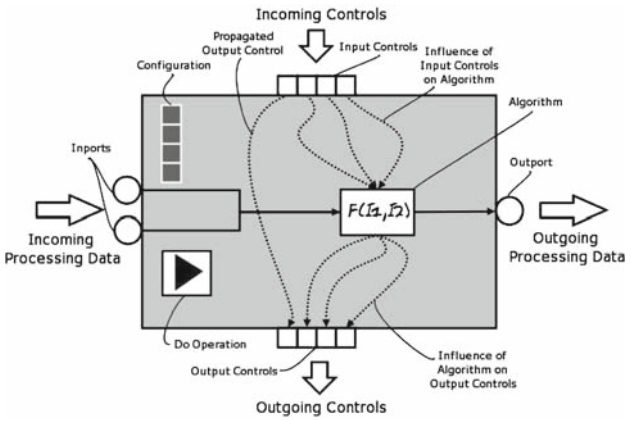

Fig. 3 A 4-MPS processing object detailed representation. A processing object has input and output ports and incoming and outgoing controls. It receives/sends synchronous data to process through the ports and receives/sends control events that can influence the process through its controls. A processing object also has a configuration that can be set when the object is not running

The metamodel offers two kinds of connection mechanisms: *ports* and *controls*. Ports transmit data and have a synchronous dataflow nature while controls transmit events and have an asynchronous nature. By synchronous, we mean that messages are produced and consumed at a predictable —if not fixed — rate.

A processing object could, for example, perform a low frequency cut-off on an audio stream. The object will have an input-port and an output-port for receiving and delivering the audio stream. To make it useful, a user might want to control the cut-off frequency using a GUI slider. Unlike the audio stream, control events arrive sparsely or in bursts. A processing object receives that kind of events through controls.

The data flows through the ports when a processing is triggered (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 (see the *Typed Connections* pattern in Sect. 2.3). Connections are handled by the *FlowControl*. This entity is also is responsible for scheduling the processing 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. Minimizing latency and securing performance conditions that guarantee correct output (avoiding underruns or deadlocks, for instance) are other responsibilities of the FlowControl.

2.1.1 Life-cycle and configurations

A 4-MPS processing object has an explicit lifecycle made of the following states: *unconfigured*, *ready*, and *running*. The processing object can receive controls and data only when running. Before getting to that state though, it needs to go through the *ready* having received a valid *configuration*.

Configurations are another kind of parameters that can be input to processing objects and that, unlike controls, produce expensive or structural changes in the processing object. 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 object when this is not in running state.

2.1.2 Static versus 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* at compile time, and *dynamically* at run time (see [17]). Static compositions in the 4 MPS metamodel are called processing composites while dynamic compositions are called networks.

Choosing between processing composites and networks is a trade-off between efficiency versus understandability and flexibility. In processing composites the developer is in charge of deciding the behavior of the objects at compile time and can therefore fine-tune their efficiency. On the other hand, networks offer an automatic flow and data management that is much more convenient but might result in reduced efficiency in some particular cases.

2.1.3 Processing networks

Nevertheless processing networks in 4MPS are in fact much more than a composition strategy. The network metaclass acts as the glue that holds the metamodel together. Figure 4 depicts a simplified diagram of the main 4MPS metaclasses.

Networks offer an interface to instantiate new processing objects given a string with its class name using a processing object *factory* and a plug-in loader. They also offer interface for connecting the processing objects and, most important, they automatically control their firing.

This firing scheduling can follow different strategies by either having a static scheduling decided before hand or implementing a dynamic scheduling policy such as a *push strategy* starting firing the up-source processings, or a *pull strategy* where we start querying for data to the most downstream processings. As a matter of fact, these different strategies depend on the topology of the network and can be directly related to the different scheduling algorithms available for dataflow networks and similar graphical models of computation (see [37] for an in-depth discussion of this topic). In any case, to accommodate all this variability the metamodel provides for different FlowControl sub-classes which are in charge of the firing strategy, and are pluggable to the network processing container.

2.2 Towards a pattern language for dataflow architectures

As explained in the previous section, the general 4 MPS metamodel can be interpreted as a particular case of a *Dataflow Network*. Furthermore, when reviewing many frameworks and environments related to CLAM (see Sect. 5) we also uncovered that most of these frameworks end up offering a variation of dataflow networks.

While 4 MPS offers a valid high-level metamodel for most of these environments it is sometimes more useful to present a lower-level architecture in the language of design patterns, where recurring and non-obvious design solutions can be shared. Thus, such pattern language bridges the gap between an abstract metamodel such as 4 MPS and the concrete implementation given a set of constraints.

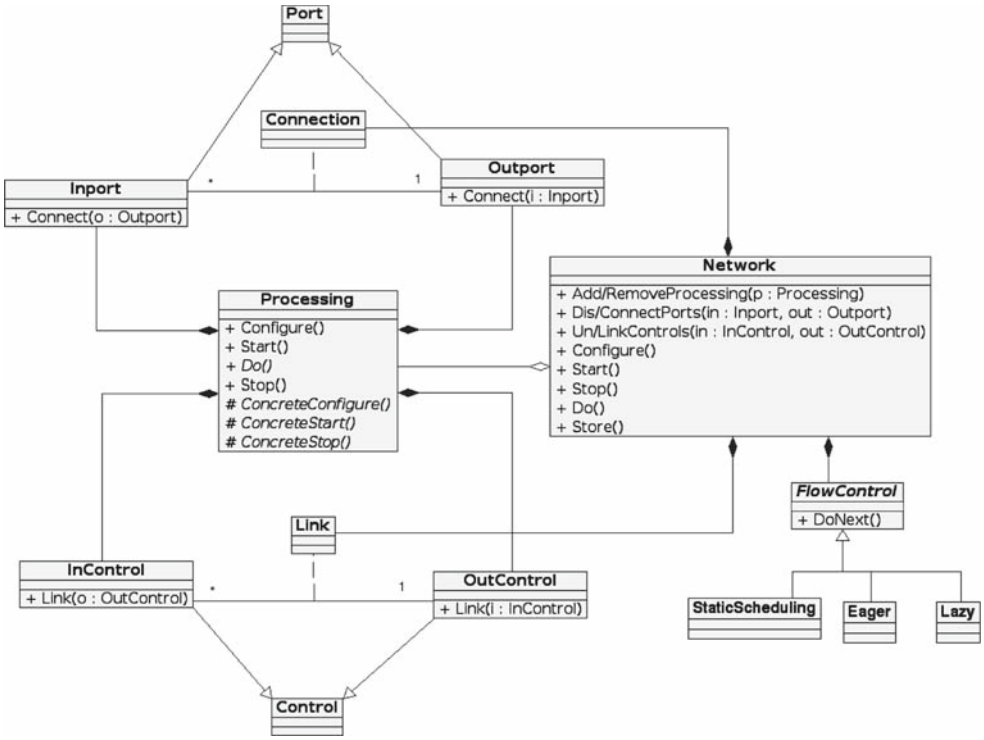

Fig. 4 Participant classes in a 4-MPS network. Note that a 4-MPS network is a dynamic run-time composition of processing objects that contains not only processing instances but also a list of connected ports and controls and a flow control

Patterns provide a convenient way to formalize and reuse design experience. However, neither dataflow systems nor other audio-related areas have yet received many attention on domain-specific patterns. The only previous efforts in the audio domain that we are aware of are several music information retrieval patterns [11] and a catalog with six real-time audio patterns presented in a workshop [55]. In the general multimedia field there are some related pattern languages like [33] but these few examples have a narrower scope than the one here presented.

There have been previous efforts in building pattern languages for the dataflow paradigm, most noticeably the one by Manolescu [34]. However, the pattern language here presented is different because it takes our experience building generic audio frameworks and models [3,54] and maps them to traditional graphical models of computation. The catalog is already useful for building systems in the multimedia domain and aims at growing (incorporating more patterns) into a more complete design pattern language of that domain.

In the following, we offer a brief summary of a complete pattern language for dataflow architecture in the multimedia domain presented in [10]. As an example we also include the more detailed description of two of the most important patterns in the catalog.

All the patterns presented in this catalog fit within the generic architectural pattern defined by Manolescu as the dataflow architecture pattern. However, this architectural pattern does not address problems related to relevant aspects such as message passing protocol, processing objects execution scheduling or data tokens management. These and other aspects are addressed in our pattern language, which contains the following patterns classified in three main categories:

- *General dataflow patterns* address problems of how to organize high-level aspects of the dataflow architecture, by having different types of module connections. Semantic ports addresses distinct management of tokens by semantic; driver ports addresses how to make modules executions independently of the availability of certain kind of tokens; stream and event ports addresses how to synchronize different streams and events arriving to a module; and, finally, typed connections addresses how to deal with typed tokens while allowing the network connection maker to ignore the concrete types.
- *Flow implementation patterns* address how to physically transfer tokens from one module to another, according to the types of flow defined by the *general dataflow patterns*. Tokens life-cycle, ownership and memory management are recurrent issues in those patterns. Cascading event ports addresses the problem of having a high-priority event-driven flow able to propagate through the network. Multi-rate stream ports addresses how stream ports can consume and produce at different rates; multiple window circular buffer addresses how a writer and multiple readers can share the same tokens buffer.

and phantom buffer addresses how to design a data structure both with the benefits of a circular buffer and the guarantee of window contiguity.
– Network usability patterns address how humans can interact with dataflow networks. Recursive networks makes it feasible for humans to deal with the definition of large complex networks; and port monitor addresses how to monitor a flow from a different thread without compromising the network processing efficiency.

Two of these patterns, typed connections and port monitor, are very central to CLAM since they enable its rapid prototyping features that will be reviewed at 4, and so, we provide here a summarized version of these patterns. Complete versions of these and the rest of the patterns in the catalog can be found in [10].

2.3 Pattern: Typed connections

Context: Most multimedia data-flow systems must manage different kinds of tokens. In the audio domain, we might need to deal with audio buffers, spectra, spectral peaks, MFCC's, MIDI, etc. And you may not even want to limit the supported types. The same applies to events (control) channels, we could limit them to floating point types but we may use structured events controls like the ones in OSC [53].

Heterogeneous data could be handled in a generic way (common abstract class, void pointers, etc.) 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: Connectible 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 is cost-sensitive and should avoid dynamic type checking and handling.
– Connections are done in run-time by the user, so mismatches in the token type should be handled.
– 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, more efficient way by knowing the concrete token type rather than just knowing an abstract base class.
– The collection of token types evolves and grows and this should not affect the infrastructure.
– A connection maker coupled to the evolving set of types is a maintenance workhorse.
– A type could be added in run time.

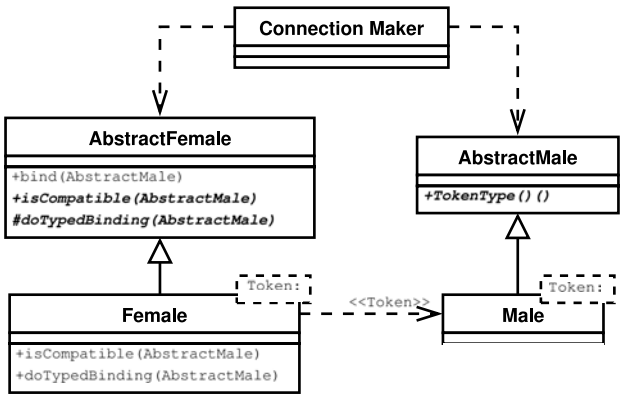

Fig. 5 Class diagram of a canonical solution of typed connections

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

The class diagram for this solution is shown in Fig. 5.

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: The solution implies that the connection maker is not coupled to token types. Just concrete modules are coupled to the token types they use.

Type safety is ensured 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 by accessing semantic type information. For example, implementations of the bind method could check that the size and scale of audio spectra match.

2.4 Pattern: Port monitors

Context Some multimedia 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 process has real-time requirements. This normally requires splitting visualization and processing into different threads, where the processing thread is scheduled as a high-priority thread. But because the non real-time monitoring must access to the processing thread tokens some concurrency handling is needed and this often implies locking in the two threads.

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 (i.e. The process result must be calculated in a given time slot)
- Visualizations must be fluid; that means that it should visualize on time and often but it may skip tokens
- 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. Internally, they have a lock-free data structure which can be simpler than a lock-free circular buffer since the visualization can skip tokens.

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 (Fig. 6).

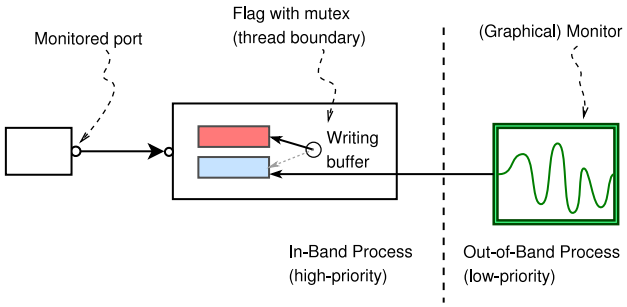

Fig. 6 A port monitor with its switching two buffers

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, as it only lasts a single flag switching.

In any case, the visualization thread may suffer starvation risk. Not because the visualization thread will be blocked but because it may be reading always from the same buffer. That may happen if every time the processing thread tries to switch the buffers, the visualization is blocking. Experience tell us that this effect is not critical and can be avoided by minimizing the time the visualization thread is accessing tokens, for example, by copying them and release.

2.5 Patterns as a language

Some of the patterns in the catalog are very high-level, such as 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 Fig. 7. The arcs between patterns mean "enables" relations: introducing a pattern in the system enables other patterns to be used.

The catalog shows how to approach the development of a complete dataflow system in an evolutionary fashion without the need to do *big up-front design*. The patterns at the top of the hierarchy suggest to start with high level decisions, driven by questions like: "do all ports drive the module execution?" And "does the system have to deal only with stream flow or also with event flow?" Then move on to address issues related to different token types such as: "do ports need to be strongly typed while connectible by the user?", or "do the stream ports need to consume and produce different block sizes?", and so on. On each decision, which will introduce more features and complexity, a recurrent problem is faced and addressed by one pattern in the language.

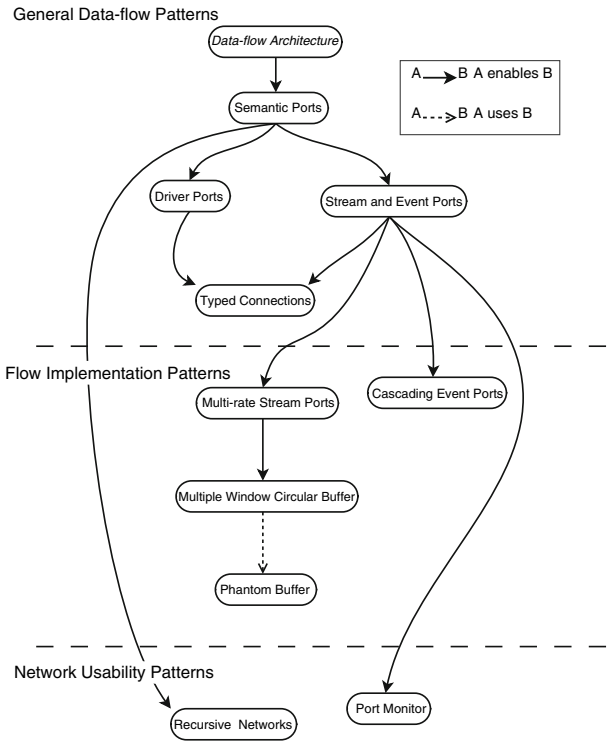

Fig. 7 The multimedia dataflow pattern language. High-level patterns are on the *top* and the *arrows* represent the order in which design problems are being addressed by developers

The above patterns are inspired by our experience in the audio domain. Nevertheless, we believe that those have an immediate applicability in the more general multimedia domain.

As a matter of fact, all patterns in the "general dataflow patterns" category can be used on any other dataflow domain. Typed connections, multiple window circular buffer and phantom buffer have applicability beyond dataflow systems. And, regarding the port monitor pattern, although its description is coupled with the dataflow architecture, it can be extrapolated to other environments where a normal priority thread is monitoring changing data on a real-time one.

Most of the patterns in this catalog can be found in many audio systems. However, examples of a few others (namely, multi-rate stream ports, multiple window circular buffer and phantom buffer) are hard to find outside of CLAM so they should be considered innovative patterns (or proto-patterns).

3 CLAM components

As seen in Fig. 8 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 [42]. It is black-box because already built-in components included in the repositories can be connected with minimum or no 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.

Apart from the kernel, CLAM includes a number of tools for services such as audio input/output or XML serialization and a number of applications that have served as a testbed and validation of the framework.

In the following, we will review the CLAM infrastructure, repositories, and its tools.

3.1 The infrastructure

The CLAM infrastructure is a direct implementation of the 4 MPS metamodel, which has already been explained in Sect. 2.1.

Indeed, the metaclasses illustrated in Fig. 4 are directly mapped to C++ abstract classes in the framework (note that C++ does not accept metaclasses naturally). These metaclasses are responsible for the white-box or extensible behavior in the framework. When a user wants to add a new processing or data to the repository a new concrete class needs to be derived from these classes.

3.2 The repositories

The *processing repository* contains a large set of ready-to-use processing algorithms, and the *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 categories such as Analysis, ArithmeticOperators, or AudioFileIO.

Although the repository has a strong bias toward spectral-domain processing because of our research 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 data repository we offer the encapsulated versions of the most commonly used data types such as audio, spectrum, 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.

3.3 Tools

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

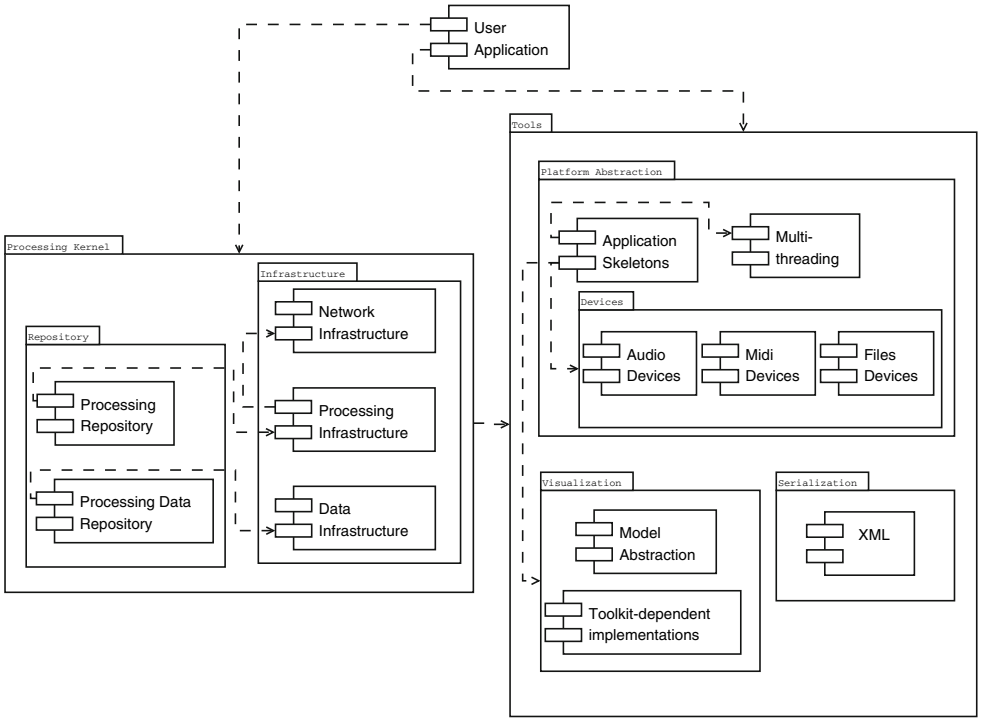

Fig. 8 CLAM components. The CLAM framework is made up of a processing kernel and some tools. The processing kernel includes an Infrastructure that is responsible for the framework white-box behavior and repositories that offer the black boxes. Tools are usually wrappers around pre-existing third party libraries. A user application can make use of any or all of these components

All these tools are possible, thanks to the integration of third party open libraries into the framework. Services offered by these libraries are wrapped and integrated into the metamodel so they can be used as natural constructs (mostly processing objects) from within CLAM. In this sense, one of the benefits of using CLAM is that it acts as a common point for already existing heterogeneous services [4].

3.3.1 XML

XML is used throughout CLAM as a general purpose storage format in order to store objects that contain data, descriptors or configurations [23]. In CLAM a spectrum as well as a network configuration can be transparently stored in XML. This provides for seamless interoperability between applications allowing easy built-in data exchange.

3.3.2 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. In CLAM this is accomplished through the visualization module, which includes many already implemented widgets offered for the Qt framework. The more prominent example of such utilities are the *port monitors*: Widgets that can be connected to ports of a CLAM network to show its flowing data. A similar tool called plots is also available for debugging data while implementing algorithms.

3.3.3 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, audio file formats, MIDI input/output, or SDIF file support, can be added to an application and then used on different operating systems with exactly the same code and always in observing the 4MPS metamodel.

3.4 CLAM applications

The framework has been tested on —but also its development has been driven by —a number of applications. Many of these applications were used in the beginning to set the domain requirements and they now illustrate the feasibility of the metamodel, the design patterns and the benefits of the framework. In the following, we will present some of these applications.

3.4.1 Spectral modeling analysis/synthesis

One of the main objectives when starting CLAM was to develop a replacement for a similar pre-existing tool.

This application (see GUI in Fig. 9) is used to analyze, transform and synthesize back a given sound. For doing so, it uses the sinusoidal plus residual model [7]. The application

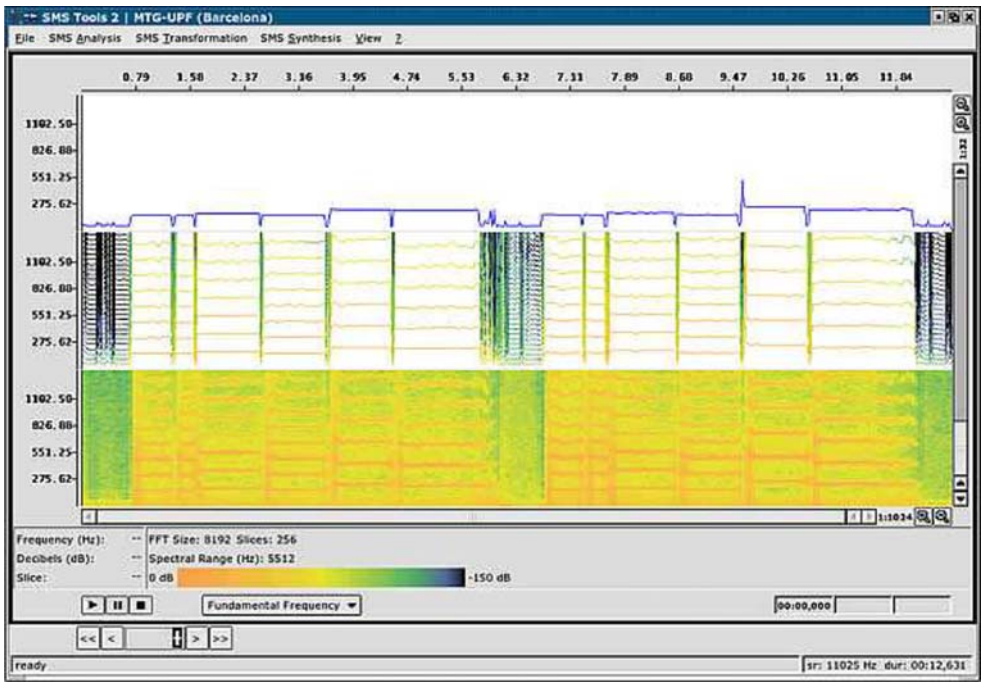

Fig. 9 The SpectralTools graphical user interface. This application can be used not only to inspect and analyze audio files but also to transform them in the spectral domain

Fig. 10 Editing low-level descriptors and segments with the CLAM annotator. This tool provides ready-to-use descriptors such as chord extraction and can also be used to launch custom algorithms

reads an XML configuration file, and an audio file (or a previously analyzed SDIF file). The input sound is analyzed, transformed in the spectral domain according to a transformation score and then synthesized back.

3.4.2 The annotator

The CLAM Annotator [9] is a tool for inspecting and editing audio descriptors (see Fig. 10). The application can be

used as a platform for launching custom extraction algorithms that analyze the signal and produce different kinds of descriptors. It provides tools for merging and filtering different source of descriptors that can be custom extractor programs or even remote sources from web services.

Descriptors can be organized at different levels of abstraction: song level, frame level, but also several segmentations with different semantics and attributes. The descriptors can be synchronously displayed or auralized to check their correctness. Merging different extractors and hand-edited ground truth has been proved very useful to evaluate extractors' performance.

3.4.3 Others

Many other sample usages of CLAM exist apart from the main applications included in the repository and described above.

For instance, *SALTO* is a software-based synthesizer [25] that implements a general synthesis architecture configured to produce high quality sax and trumpet sounds. *SpectralDelay*, also known as CLAM's Dummy Test, was the first application implemented in the framework. It was chosen to drive the design in its first stages. The application implements a delay in the spectral domain: the input audio signal can be divided with CLAM into three bands and each of these bands can be delayed separately.

The repository also includes many smaller examples that illustrate how the framework can be used to do particular tasks ranging from a simple sound file playback to a complex MPEG7 descriptor analysis.

Finally, other CLAM applications are not included in the repository for a variety of reasons, namely:

- The application has not reached a stable enough status. This is the case for many student projects that although interesting are not mature enough to be in the main repository. CLAM has been used to create a guitar effect that uses neural networks to learn its sound, a real-time fingerprint algorithm or a networked and distributed audio application.
- The application is mostly done with an artistic goal and therefore not very general purpose. For instance, Rappid [43] was a testing workbench for the CLAM framework in high demanding situations. Rappid implemented a simple time-domain amplitude modulation algorithm but any other CLAM based algorithm can be used in its place. Rappid was tested in a live-concert situation when it was used as an essential part of a composition for harp, viola and tape, presented at the Multiphonies 2002 cycle of concerts in Paris.
- It is a third-party application. Applications such as an Italian speech synthesizer have been developed by third parties and therefore not integrated into the repository.
- The application is protected by a non-disclosure agreement. Because CLAM has a double-licensing scheme, it can also be used in proprietary software.

4 CLAM as a rapid prototyping environment

So far we have seen that CLAM can be used as a regular application framework by accessing the source code. Furthermore, ready-to-use applications such as the ones presented in the previous section provide off-the-self functionality.

But latest developments have brought *visual building* capabilities into the framework. These allow the user to concentrate on the research algorithms and not on application development. Visual building is also valuable for rapid prototyping of applications and plug-ins.

CLAM's visual builder is known as the NetworkEditor (see Fig. 11). It allows to generate an application —or only its processing engine— by graphically connecting objects in a patch. 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.

Having a proper development environment is something that may increase development productivity. Development frameworks offer system models that enable system development dealing with concepts of the target domain. Eventually, they provide visual building tools that also improve productivity [24]. In the audio and music domain, the approach of modeling systems using visual dataflow tools has been widely and successfully used in system such as PD [40], Marsyas [46], and Open Sound World [14]. 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 workflow.

User interface design is supported by existing toolboxes and visual prototyping tools. Examples of such environments which are freely available are Qt Designer, FLTK Fluid or GTK's Glade. But such tools just solve the composition of graphical components into a layout, offering 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 section describes an architecture that addresses this gap and enables fully visual building of real-time audio

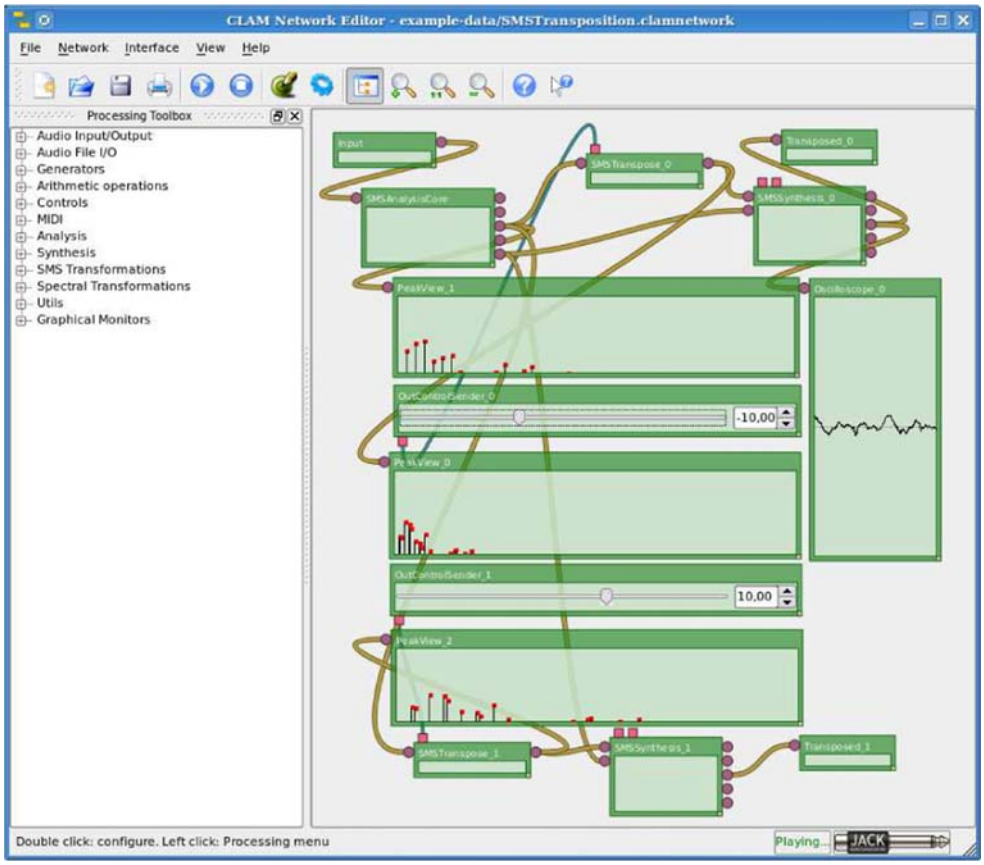

Fig. 11 NetworkEditor is the visual builder of the CLAM framework. It can be used not only as an interactive multimedia dataflow application but also to build networks that can be run as stand-alone applications embedded in other applications and plugins

processing applications by combining visual dataflow tools and visual GUI design tools.

4.1 Target applications

The set of applications the architecture is able to visually build includes real-time audio processing applications such as synthesizers, real-time music analyzers and audio effects and plugins (Fig. 12).

The only limitation imposed on the target applications is that their logic should be limited to 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, etc.), visualizing the inner data and controlling some algorithm parameters while running. Note that these limitations are very much related to the explicit life-cycle of a 4MPS Processing object outlined in Sect. 2.1.1.

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

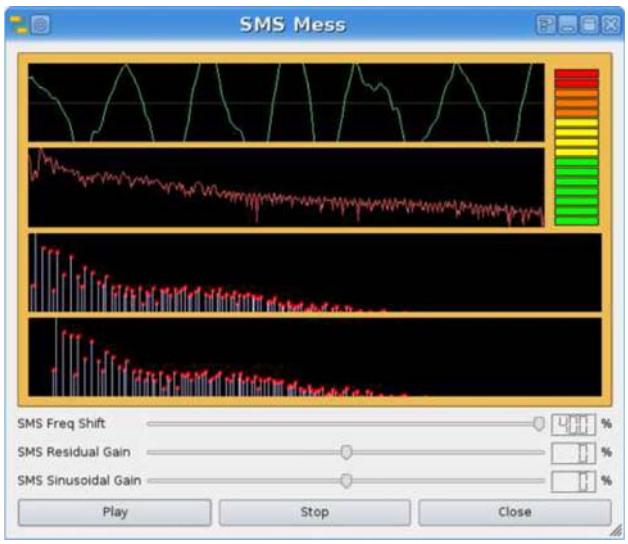

Fig. 12 An example of a rapid-prototyped audio effect application: Pitch transposition. This application, which can be prototyped in CLAM in a matter of minutes, performs an spectral analysis, transforms the audio in the spectral domain, and synthesizes back the result. Note how, apart from representing different signal components, three sliders control the process interacting directly with the underlying processing engine

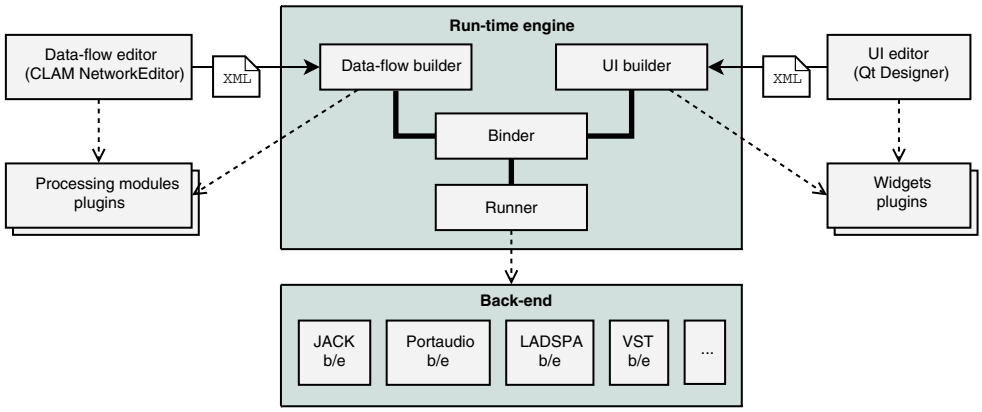

Fig. 13 Visual prototyping architecture. The CLAM components that enable the user to visually build applications

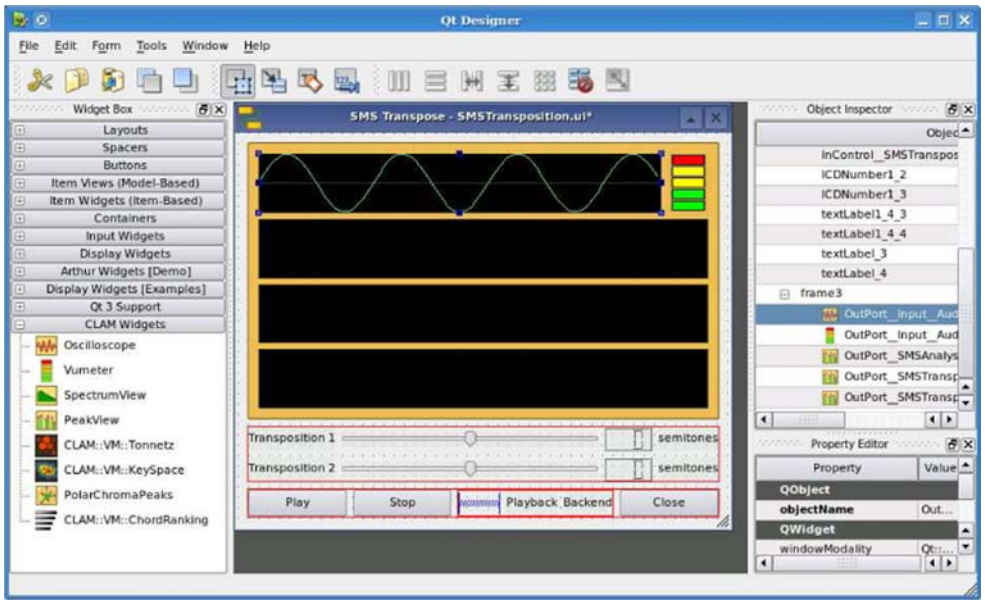

Fig. 14 Qt Designer tool editing the interface of an audio application

the architecture would help to build important parts of such applications.

The architecture provides the following features:

- Communication of any kind of data and control objects between GUI and processing core (not just audio buffers).
- The prototype can 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 back-ends (JACK, ALSA, PORTAUDIO, LADSPA, VST, AudioUnit, etc.)

4.2 Main architecture

The proposed architecture (Fig. 13) 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 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 are later 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 14 shows a Qt Designer session designing the interface for an audio application, which uses some CLAM data objects related widgets provided by CLAM as a Qt widgets plugin. Note that other CLAM data related

widgets are available on the left panel list. For example to view spectral peaks, tonal descriptors or spectra.

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 instantiated user interface by using introspection capabilities.

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.

4.3 Run-time engine

If only a dataflow visual tool and a visual interface designer was provided, some programming would still be required to glue it all together and launch the application. The purpose of the run-time engine, which is called prototyper in our implementation, is to automatically provide this glue. Next, we enumerate the problems that the run-time engine faces and how it solves them.

4.3.1 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. The complexity to be addressed is how to do such task when the elements of such structure are not known before hand because they are defined by add-on plugins. [3]

Both CLAM and Qt frameworks provide object factories that can build objects given a type identifier. Because we want interface and processing components to be expandable, 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 a creator on the factory at plugin initialization.

In order to build up the components into an structure, both frameworks provide means for reflection so the builder can discover the properties and structure of unknown objects. For instance, in the case of processing elements, the builder can browse the ports, the controls, and the configuration parameters using a generic interface, and it can guess the type compatibility of a given pair of ports or controls.

[3] Note that this is a recurring issue in audio applications where the use of plug-ins is common practice.

4.3.2 Relating processing and user interface

The run-time engine must relate components of both structures. For example, the spectrum view on the Transposition application (second panel on Fig. 12) needs to periodically access spectrum data flowing by a given port of the processing network. The run-time engine first has to identify which components, are connected. Then decide whether the connection is feasible. For example, spectrum data cannot be viewed by an spectral peaks view. And then, perform the connection, all that without the run-time engine knowing anything 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 mentioned in Sect. 2.2. In a nutshell, this design pattern allows to establish a type dependent connection construct between two components without the connector maker knowing the types and still be type safe. This is done by dynamically check the handled type on connection time, and once the type is checked both sides are connected using statically type checked mechanisms which will do optimal communication on run-time.

4.3.3 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 [34]. 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 Sect. 2.2, are needed.

A port monitor is a special kind of processing component which does double buffering of an input data and offers a thread safe data source interface for the visualization widgets. A flag tells which is the read and the write buffer. The processing thread does a try lock to switch the writing buffer. The visualization thread will block the flag when accessing the data but as the processing thread just does a 'try lock', so it will just overwrite the same buffer but it won't block fulfilling the real-time requirements of the processing thread.

4.3.4 System back-end

Most of the application logic is coupled to sinks and sources for audio data and control events. Audio sources and sinks depend on the context of the application: JACK, ALSA, ASIO, DirectSound, VST, LADSPA, and so on. 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 back-end plugins to deal with this issues.

Back-end plugins address the often complex back-end setup, relate and feed sources and sinks in a network with real system sources and sinks, control processing thread and provide any required callback. Such plugins, hide all that complexity with a simple interface with operations such as setting up the back-end, binding a network, start and stop the processing, and release the back-end.

The back-end 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 back-end. Back-end plugin system also provides interface to cover such functionality.

5 The big picture

This article would be incomplete without at least a short overview of existing environments that somehow share objectives or constructs with CLAM. In the following we will outline similar frameworks and tools and highlight what makes CLAM unique.

5.1 A (very brief) survey of existing audio environments

In this section, we will give a brief survey of existing frameworks and environments for audio processing. Although, because of space constraints, the survey is not much more than an ordered list, we believe that it is already valuable to have all these references listed in such a way. Most of these environments are extensively reviewed in [3].

The current arena presents heterogeneous collections of systems that range from simple libraries to full-fledged frameworks.[4] Unfortunately, it is very difficult to have a complete picture of the existing environments in order to choose one or decide designing a new one.

In order to contextualize our survey and because the 4 MPS metamodel is valid for any kind of multimedia processing system, it is best to first start with a list of relevant environments not only for audio but also for image and multimedia:

- *Multimedia Processing Environments*: Ptolemy [27], BCMT [35], MET++ [1], MFSM [20], VuSystem [31], Javelina [26], VDSP [36]
- *(Mainly) Audio Processing Environments*: CLAM [6], The Create Signal Library (CSL) [39], Marsyas [45], STK [16], Open Sound World (OSW) [14], Aura[18], SndObj [29], FORMES [15], Siren [38], Kyma [44], Max [41], PD[40]
- *(Mainly) Visual Processing Environments*: Khoros-Cantata [56] (now VisiQuest), TiViPE [32], NeatVision [51], AVS [47], FSF [21]

If we now focus in the audio field, we can further classify the environments according to their scope and main purpose as follows:

1. *Audio processing frameworks*: software frameworks that offer tools and practices that are particularized to the audio domain.
   (a) *Analysis oriented*: Audio processing frameworks that focus on the extraction of data and descriptors from an input signal. Marsyas by Tzanetakis [45] is probably the most important framework in this sub-category as it has been used extensively in several music information retrieval systems [45].
   (b) Synthesis oriented: Audio processing frameworks that focus on generating output audio from input control signals or scores. STK by Cook [16] has already been in use for more than a decade and it is fairly complete and stable.
   (c) General purpose: These audio processing frameworks offer tools both for analysis and synthesis. Out of the ones in this sub-category both SndObj [29] and CSL [39] are in a similar position, having in any case some advantages and disadvantages. CLAM, the framework developed presented in this article, should be included in this sub-category.
2. *Music processing frameworks*: These are software frameworks that instead of focusing on signal-level processing applications they focus more on the manipulation of symbolic data related to music. Siren [38] is probably the most prominent example in this category.
3. *Audio and music visual languages and applications*: Some environments base most of their tools around a graphical metaphor that they offer as an interface with the end user. In this section, we include important examples such as the Max [41] family or Kyma [44].
4. *Music languages*: In this category we find different languages that can be used to express musical information

[4] Note that throughout the article we are adhering to the most commonly accepted definition in which a *framework* is defined as "a set of classes that embodies an abstract design for solutions to a family of problems" [28], we will use the term *environment* otherwise.

(note that we have excluded those having a graphical metaphor, which are already in the previous one). Although several models of languages co-exist in this category, it is the Music-N family of languages the most important one. *Music-N languages* languages are based their proposal on the separation of musical information into statical information about *instruments* and dynamic information about the *score*, understanding this score as a sequence of time-ordered note events. Music-N languages are also based on the concept of *unit generator*. The most important language in this category, because of its acceptance, use and importance, is Csound [49].

5.2 CLAM in the big picture

Although as shown earlier, many other related environments exist there are some important features of our framework that are worth pointing out and make it somehow different[5] (see [3] for an extensive study and comparison of most of them):

1. All the code is *object-oriented* and written in C++ for efficiency. Although 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 [22] and C++ idioms [2], good development practices like test-driven development [12] and refactoring [19], 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. It integrates a large number of already existing third party tools under a common metamodel.
5. It 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).
6. It is *cross-platform*.
7. The framework can be used either as a regular C++ *library* or as a *visual prototyping tool*.
8. It has been formalized through a metamodel and a pattern language so the lessons learned can be used beyond CLAM.

[5] Note that most of these features can be individually traced in many other environments. It is the inclusion of all of them what makes CLAM unique.

6 Conclusions

CLAM already has a long life as it has been developed for the past 6 years. During this period of time, and even before reaching its first stable release, the framework has proved useful in many applications and scenarios.

In this article, we have presented a thorough review of CLAM features as in its 1.0 release. We have focused especially on the latest developments, which drive the framework in a more *black-box* direction by presenting a rapid-prototyping visual builder that allows to build efficient stand-alone applications and plugins.

And although CLAM is valuable in itself, as an example of a well-designed multimedia application framework, we have also shown why many of its constructs and design decisions can be shared beyond the framework itself. In the 4 MPS metamodel we offer a domain-specific language that is shared by many multimedia processing applications and in the dataflow pattern language we detail more fine grain constructs that are also found in many environments.

Acknowledgments The CLAM framework has been developed at the Universitat Pompeu Fabra thanks to the contribution of many developers and researchers. A non-exhaustive list of contributors should at least include Maarten de Boer, Miguel Ramírez, Xavi Rubio, Ismael Mosquera, Xavier Oliver, Enrique Robledo, and our students from the Google Summer of Code.

References

1. Ackermann, P.: Direct manipulation of temporal structures in a multimedia application framework. In: Proceedings of the 1994 ACM Multimedia Conference, 1994
2. Alexandrescu, A.: Modern C++ design. Addison–Wesley, Pearson Education, New York (2001)
3. Amatriain, X.: An object-oriented metamodel for digital signal processing with a focus on audio and music. PhD thesis, Universitat Pompeu Fabra, Barcelona, Spain, 2004
4. Amatriain, X.: Clam: a framework for audio and music application development. IEEE Softw. **24**(1), 82–85 (2007)
5. Amatriain, X.: A domain-specific metamodel for multimedia processing systems. IEEE Trans. Multimed. **9**(6), 1284–1298 (2007)
6. Amatriain, X., Arumi, P.: Developing cross-platform audio and music applications with the CLAM Framework. In: Proceedings of International Computer Music Conference, 2005
7. Amatriain, X., Bonada, J., Loscos, A., Serra, X.: DAFX: Digital Audio Effects (Udo Z+áalzer ed.), chapter Spectral Processing. pp. 373–438. Wiley, New York (2002)
8. Amatriain, X., de Boer, M., Robledo, E., Garcia, D.: 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, New York (2002)
9. Amatriain, X., Massaguer, J., Garcia, D., Mosquera, I.: The clam annotator: a cross-platform audio descriptors editing tool. In: Proceedings of the 2005 International Symposium on Music Information Retrieval, ISMIR '05, 2005

10. Arumi, P., Garcia, D., Amatriain, X.: A dataflow pattern language for sound and music computing. In: Proceedings of Pattern Languages of Programming (PloP 06), 2006
11. Aucouturier, J.: Ten experiments on the modelling of polyphonic timbre. PhD thesis, University of Paris 6/Sony CSL Paris, 2006
12. Beck, K.: Test Driven Development by Example. Addison–Wesley, New York (2000)
13. Burbeck, S.: Application programming in smalltalk-80: how to use model-view-controller (mvc). Technical report, Xerox PARC, 1987
14. Chaudhary, A., Freed, A., Wright, M.: An open architecture for real-time audio processing software. In: Proceedings of the Audio Engineering Society 107th Convention, 1999
15. Cointe, P., Briot, J.P., Serpette, B.: Object-Oriented Concurrent Programming, chapter The FORMES Language: a Musical Application of Object Oriented Concurrent Programming. MIT Press, Cambridge (1987)
16. Cook, P.: Synthesis Toolkit in C++. In: Proceedings of the 1996 SIGGRAPH, 1996
17. Dannenberg, R.B.: Combining visual and textual representations for flexible interactive audio signal processing. In: Proceedings of the 2004 International Computer Music Conference (ICMC'04) (2004)
18. Dannenberg, R.B.: Combining visual and textual representations for flexible interactive audio signal processing. In: Proceedings of the 2004 International Computer Music Conference (ICMC'04) (2004)
19. Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D.: Refactoring: improving the design of existing code. Addison–Wesley, New York (1999)
20. François, A.R.J., Medioni, G.G.: A modular middleware flow scheduling framework. In: Proceedings of ACM Multimedia 2000, pp. 371–374, Los Angeles, CA, November 2000
21. François, A.R.J., Medioni, G.G.: A modular software architecture for real-time video processing. In: IEEE International Workshop on Computer Vision Systems, pp. 35–49. Vancouver, B.C., Canada, July 2001
22. Johnson, R., Gamma, E., Helm, R., Vlissides, J.: Design Patterns—Elements of Reusable Object-Oriented Software. Addison–Wesley, New York (1996)
23. Garcia, D., Amatrian, X.: 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
24. Green, T.R.G., Petre, M.: Usability analysis of visual programming environments: a "cognitive dimensions" framework. J. Vis. Lang. Comput. **7**(2), 131–174 (1996)
25. Haas, J.: SALTO—a spectral domain saxophone synthesizer. In: Proceedings of MOSART Workshop on Current Research Directions in Computer Music, Barcelona, Spain, 2001
26. Hebel, K.J.: The well-tempered object. musical applications of object-oriented software technology, chapter Javelina: An Environment for Digital Signal Processor Software Development. pp. 171–187. MIT Press, Cambridge (1991)
27. Hylands, C. et al.: Overview of the Ptolemy Project. Technical report, Department of Electrical Engineering and Computer Science, University of California. Berklee, CA, USA (2003)
28. Johnson, R.E., Foote, J.: Designing Reusable Classes. J. Object Oriented Program. **1**(2), 22–35 (1988)
29. Lazzarini, V.: Sound processing with the SndObj Library: an overview. In: Proceedings of the 4th International Conference on Digital Audio Effects (DAFX '01), 2001
30. Lee, E.A., Park, T.: Dataflow process networks. In: Proceedings of the IEEE, vol. 83, pp. 773–799 (1995)
31. Lindblad, C.J., Tennenhouse, D.L.: The VuSystem: A Programming System for Compute-Intensive Multimedia. IEEE J. Sel. Areas Commun. **14**(7), 1298–1313 (1996)
32. Lourens, T.: TiViPE—Tino's visual programming environment. In: Proceedings of the 28th Annual International Computer Software and Applications Conference (COMPSAC'04), pp. 10–15, 2004
33. Lucid, H., Huljenid, D.: Developing multimedia services using high-performance concurrent communication design patterns. In: Proceedings of the 7th International Conference on Telecommunications. ConTEL 2003, 2003
34. Manolescu, D.A.: A dataflow pattern language. In: Proceedings of the 4th Pattern Languages of Programming Conference, 1997
35. Mayer-Patel, K., Rowe, L.: Design and performance of the Berkeley continuous media toolkit. In: Proceedings of Multimedia Computing and Networking 1997, pp. 194–206, San Jose, CA (1997)
36. Mellinger, D.K., Garnett, G.E., Mont-Reynaud, B.: The Well-tempered Object Musical Applications of Object-Oriented Software Technology, chapter Virtual Digital Signal Processing in an Object-Oriented System. pp. 188–194. MIT Press, Cambridge (1991)
37. Parks, T.M.: Bounded Schedule of Process Networks. PhD thesis, University of California at Berkeley, 1995
38. Pope, S.T.: Squeak: Open Personal Computing and Multimedia, chapter Music and Sound Processing in Squeak Using Siren. Prentice- Hall, Englewood Cliffs (2001)
39. Pope, S.T., Ramakrishnan, C.: The Create Signal Library ("Sizzle"): Design, Issues and Applications. In: Proceedings of the 2003 International Computer Music Conference (ICMC '03), 2003
40. Puckette, M.: Pure data. In: Proceedings of the 1996 International Computer Music Conference, pp. 269–272 (1996)
41. Puckette, M.: Max at seventeen. Comput. Music J. **26**(4), 31–43 (2002)
42. Roberts, D., Johnson, R.: Evolve frameworks into domain-specific languages. In: Proceedings of the 3rd International Conference on Pattern Languages for Programming, Monticelli, IL, USA, September 1996
43. Robledo, E.: RAPPID: robust real time audio processing with CLAM. In: Proceedings of 5th International Conference on Digital Audio Effects, Hamburg, Germany, 2002
44. Scaletti, C., Johnson, R.E.: An interactive environment for object-oriented music composition and sound synthesis. In: Proceedings of the 1988 Conference on Objec-Oriented Programming, Systems, Languages, and Applications (OOPSLA'88), pp. 25–30, September 1988
45. Tzanetakis, G., Cook, P.: Marsyas3D: a prototype audio browser-editor using a large-scale immersive visual and audio display. In: Proceedings of the International Conference on Auditory Display (ICAD). IEEE, 2001
46. Tzanetakis, G., Cook, P.: Audio Information Retrieval using Marsyas. Kluewer, Dordrecht (2002)
47. Upson, C. et al.: The application visualization system: a computational environment for scientific visualization. IEEE Comput. Graph. Appl. **9**(4), 32–40 (1989)
48. van Dijk, H.W., Sips, H.J., Deprettere, Ed F.: On context-aware process networks. In: Proceedings of the International Symposium on Mobile Multimedia & Applications (MMSA 2002), December 2002
49. Vercoe, B.L.: CSound. The CSound Manual Version 3.48. A Manual for the Audio Processing System and supporting program with Tutorials, 1992
50. Weinand, A., Gamma, E., Marty, R.: Design and implementation of ET++, a seamless object-oriented application framework. Struct. Program. **10**(2) (1989)

51. Whelan, P.F., Molloy, D.: Machine Vision Algorithms in Java: Techniques and Implementation. Springer, Berlin (2000)
52. Wilson, D.A.: Programming With Macapp. Addison–Wesley, New York (1990)
53. Wright, M.: Implementation and performance issues with open sound control. In: Proceedings of the 1998 International Computer Music Conferen ce (ICMC '98). Computer Music Association, 1998
54. www CLAM. CLAM website: http://www.iua.upf.es/mtg/clam, 2004.
55. www Dannenberg. Dannenberg website: http://www.cs.cmu.edu/rbd/doc/icmc2005workshop/, 2004
56. Young, M., Argiro, D., Kubica, S.: Cantata: visual programming environment for the khoros system. Comput. Graph. **29**(2), 22–24 (1995)