Visual prototyping for audio applications

David García Garzón

Master thesis submitted in partial fulfillment of the requirements for the degree:

Màster en Tecnologies de la Informació, la Comunicació i els Mitjans Audiovisuals

Supervisor: Xavier Serra

Department of Information and Communication Tecnologies
Universitat Pompeu Fabra
Spain
September 2007

Abstract

The goal of the proposed master thesis is to define an architecture that enables visual prototyping of real-time audio applications and plug-ins. 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. Specifically, this research will address the problem of having to bind interactive user interface to a real-time processing core, when both are defined dynamically, the set of components is extensible, it allows bidirectional communication of arbitrary types of data between the interface and the processing core, and, it still fulfils the real-time requirements of audio applications.

Acknowledgements

I would like to thank everyone who helped to make this dissertation possible. But it is a long list, so I am sure I will miss someone. In any case the list should definitely start wit Xavier Amatrian and Pau Arumí. Most of the work explained here is the result of our joint effort to build something we could be proud of sharing it with the world. I am also very grateful to Xavier Serra who make that possible and let me be part of the MTG for so many time, one of the most interesting and dynamic research groups I could be part of.

This thesis is also built on the work of some friends of mine who also contributed to the CLAM project: Maarten Deboer, Miguel Ramírez, Xavier Rubio, Enrique Robledo, Albert Mora, Ismael Mosquera, Sandra Gilabert, Xavier Oliver... I wanted to have an special mention to Josep Blat who helped us to keep CLAM alive.

I am also grateful to many people on the UPF who supported me in many aspects such as Perfecto Herrera, Emilia Gómez, Oscar Celma, Fabien Gouyon, Rafael Ramírez, Sergi Jordà, Jordi Massaguer, Lars Fabig, Sam Roig, Bram de Jong and many others.

I would like to thank a lot of people I met at the Queen Mary University of London, specially to Juan Bello, Christopher Harte, Sammer Abbdallah, Christian Landone and Kate Noland. I am also grateful to the people at PLoP community, specially to Ralph Johnson, for providing us with so many insights on paper writing and software development.

Most of the work described in this thesis and the thesis itself has been done using free software. I am very grateful to the free software and open source community and to the people that boost it with an special mention to the Linux Audio Community which have built up a nice platform for creativity and the foundation for all our work.

I am indebted to my parents who supported and loved me all those years. And, finally I couldn't stand so much work without the love and affection of Marta, Brenda, Kimi, Isa, Esteban, Silvia, Javis, Nico, Miriam, Duke and the rest of people who share wine (and patxaran!) with me. Thanks to all of you.

Some of the work explained in this thesis has been partially funded by the Agnula European Project (IST-2001-34879), by the SIMAC European Project (IST-507142) and by the Generalitat de Catalunya (exp. 200/05 ST).

To Raquel (1976-1992)

I've been looking so long
at these pictures of you...

Contents

List of Figures

Chapter 1

Introduction

This thesis addresses the general problem of how to efficiently develop audio and music software. This work proposes an architecture that enables visual prototyping of real-time audio applications. We analyze implementation issues that developers have to deal when developing different archetypes of audio applications, such as user interface communication, multi-threading or real-time restrictions, and the means that software engineering provides to handle such complexities.

This chapter describes the context of this work, including its motivation, the research context, a summary of the work, and a description of the content of this thesis.

1.1 Motivation

I've been working on the Music Technology Group of the Universitat Pompeu Fabra for six years. That dynamic research group leaded by Xavier Serra is involved in the development of innovative audio and music technology. One of my main function there was providing software engineering support to other researchers, by porting, testing and optimizing code, integrating systems and developing research prototypes. Those tasks provided me the opportunity to be involved in the development of very different kinds of audio applications: synthesizers, audio effects, authoring tools, music information retrieval systems, plug-ins, web applications...

One of the main outputs of that activity has been the CLAM framework, co-developed with Xavier Amatriain, Pau Arumí and others. The original

intent of the framework was to join development efforts among different teams within the group. Every team was implementing similar algorithms and utilities and the framework was a common place to share them. This way, testing and porting would be centralized and integration would be easier. But we paid our initial design inexperience. The framework was powerful enough to build the diverse set of applications we required to but it was hardly usable by non-experts. Moving existing code to the framework was a hard task.

Thus, over the last years, we focused on simplifying the framework usage and providing tools to easy develop with it. In this context, the work presented in this thesis represents an step beyond on what similar frameworks are able to do: An architecture to visually build full real-time audio applications.

1.2 The problem

There is a long way from the conception of a novel audio processing algorithm until it becomes an end-user product. In a very rough generalization we can say that the typical process has two development levels. On the first level, several versions of the processing algorithm are evaluated and their parameters are tunned in order to get optimal results. Because this stage requires flexibility, processing algorithms researchers often use scripting languages such as Matlab. In a later stage, algorithms are integrated into a end-user product. Here computational performance is often a requirement and the algorithm is usually ported to a low level compiled language such as C or C++. This language port carries not just costs and time, but also risks due to translation errors or to the evolution divergence, that usually happens after the translation, between the product and the research versions of the algorithm. Another factor that make this two stage development hard is the fact that an audio end-user product must address a lot of low level platform issues that can be out of the scope of an algorithm demonstrator. In order to have a quality end-user product, several refinement iterations are needed on the interface, obtaining user feedback. This not only makes the process longer but also leads to algorithm modifications not foreseen during the research level. This research proposes means to reduce the gap between the algorithm conception and a working end-user application.

Both industry and academy can get the benefits of such research. In the industry, reducing the time of the overall process, the so called 'time to market'[?], gives a clear advantage over competitors[?]. This fact, that is already true for the traditional market, becomes even more vital for success in the context of the fast paced technology market. Academy could also benefit of having end-user applications of their technology. When conducing user experiments, a end-user application is more comfortable to use for the subjects of the experiment. End-user application is also a good technology demonstrator. And working toward end-user application eases the technology transfer to the society.

1.3 The proposed solution

So, how to address this problem? Proper development environment may increase development productivity and thus, reduce the time to market[?]. Development frameworks offer system models that enables system development dealing with concepts of the target domain. Eventually, frameworks 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 frameworks such as PD [?], Marsyas [?], Open Sound World [?] and CLAM [?].

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 interface builders which gives a similar flexibility for the user interface than the one data-flow tools provide to build the processing core. Examples of such environment 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 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 mostly related to the communication between the processing core and the user interface.

The proposed research is to define an architecture that provides the logic to bind a data-flow definition built with a visual data-flow editor to a user interface defined with a visual GUI builder in order to build a full featured

real-time audio applications.

Challenges to be addressed are two fold. On one hand, the architecture should solve all the programming issues that developers of real-time audio applications currently have to face, and do it in a transparent and generalized way. On the other hand we should face new challenges introduced by the prototyping architecture itself.

Real-time audio programming issues are discussed in detail in chapter **??**. Examples of issues to be solved are multi-platform audio devices access, communication between real-time and non-real-time threads, latency reduction, jitter handling...

Some of the issues that the prototyping architecture introduces are related to the fact that we need to locate and bind unknown elements of two two dynamically created structures. When programming an audio applications the developer has direct access to the objects to relate and their interface. The prototyping infrastructure should discover which elements are meant to be related. Moreover, in order to allow extensibility, the architecture should not limit the kind of elements to deal in both the processing and the user interface sides.

1.4 Scope of the solution

The proposed architecture provides the following features:

- Building of full featured applications with a custom user interface
- 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
- Plug-in extensibility for:
  - Processing units
  - Graphical elements for data visualization and edition
  - Data tokens types
  - System connectivity back-ends (ASIO, JACK, OSC, VST, LADSPA...)

The set of applications we want to support are real-time processing applications which has a simple application logic. That is, just the application logic needed for starting and stopping a processing algorithm, configuring it, connecting it to the system streams (audio, MIDI, files, OSC, plug-in system...), visualizing the inner processing data and controlling parameters while running.

Given that limitation, the architecture to define would not claim to 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 and the work on this thesis should help to define abstractions that would help to develop visual frameworks in the future. The architecture just will claim building applications such as synthesizers, real-time music analyzers or audio effects.

Also by 'visual prototyping' we are not referring to a complete visual language that could allow build real-time system without programming. We just meant that the developer should address just the novel processing and interface components. Once all components are available, the full application can be built with visual builders.

1.5 Methodology

1.5.1 Goals

We present here an overview of the goals of this thesis:

- Review the state of the art in software engineering tools to fasten the development and criteria on how to evaluate them
- Review the state of the art of audio software engineering to get insight on solutions to common issues on audio applications.
- Abstract a model to describe the internal structure of a audio application so that such model could help to define the issues that need to be solved.
- Define an architecture that would enable visual development of audio applications

- Iteratively implement it while addressing different use cases
- Provide a qualitative analysis on which are the benefits of using such architecture and a review of the limitations

1.5.2 Process

An actual implementation of the architecture is needed to be able to evaluate its feasibility and usefulness. As most authors in recent software engineering literature suggest [**?**][**?**], early generalizations may lead to over-design. They recommend iterative work on the implementation to get to a proper generalization. Thus, the work presented in this thesis is the result of a iterative process of refining by considering different use cases and addressing different features in a incremental pace.

In this iterative process, several parallel activities have taken place (see figure **??**). While some activities sought the goal of having a more usable framework, others dealt with coming up with the appropriate abstractions and reusable constructs that can be reused beyond the framework. Two of such abstractions are an object oriented meta-model for multimedia processing, described in section **??** and more deeply in [**?**], and a pattern language for data-flow systems, described in section **??** and more deeply in [**?**] and [**?**].

1.5.3 Evaluation

Evaluation is a tricky problem in Software Engineering. An ideal environment would be having the same system developed in different ways just changing the aspects of the process to be evaluated and comparing how each aspect affects the development efficiency. That evaluation method is not viable because building complex systems is expensive, and, even in this ideal environment, we are not taking into account human factors that would make two identical experiments differ. Human factors forces us to use an statistical approach and, thus, evaluation would require more cases.

Because the clean room approach is not viable, the classical approach is to analyze the development process of existing real projects. This approach is very limited within the world of the proprietary software. The set projects a researcher is able to analyze tends to be very limited due to corporate and organizational boundaries and confidentiality requirements. Reproducing

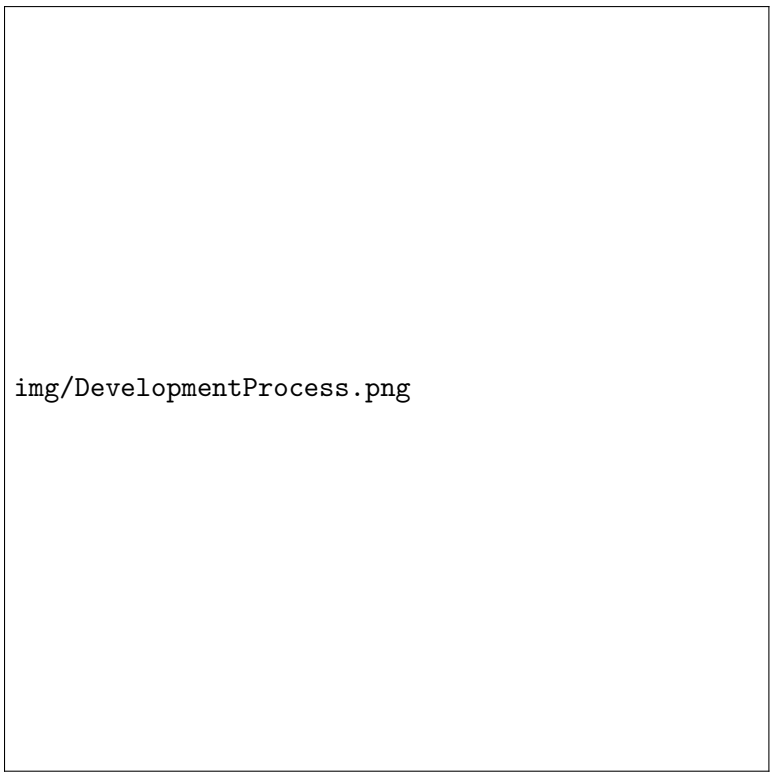

Figure 1.1: Parallel activities on the iterative inception process.

results is even harder as it requires peer researchers to setup a similar set of data. Moreover, accessible data on the development of such projects has a high risk of bias as actors of the process tend to hide things that does not work well. Fortunately, the large availability of open source projects and the visibility they offer to their development process give us a chance to obtain more significant metrics. Robles[**?**] addresses different ways of exploiting such data sources in order to reach insightful conclusions about the development and evolution of existing free and open source software.

The proposed evaluation method is to use the architecture to build several audio and music applications. Some of them from scratch and some of them reimplementations of existing open source software. Then we'll evaluate the effectiveness of the architecture either by comparing the programming effort to the original one when available, or applying some systematic qualitative criteria which are given along the chapter **??**.

But in order to rely in such evaluation, there are some aspects that must be considered carefully. Most of the expected efficiency boosts rely on components reuse. Of course, reuse is viable when the component already exists. So we should provide a criteria to estimate the likelihood for a given component to be already present and evaluate the development cost of such component consistently.

Another aspect to consider is the fact that a reimplementation would not need the exploration process the first implementation had. So, both development processes won't be comparable. The solution for this issue can be either considering metrics that evaluates just the final artifacts, or trying to reproduce the exploration process, which can be also valuable.

In summary, the proposed methodology is to iterate through the following steps

- Analyze the development process of a set of existing open source audio applications.
- Adjust the definition of the architecture so that would support their construction.
- Implement such architecture.
- Implement the analyzed audio application by using the architecture.

- Compare development processes or final artifacts and extract conclusions.

1.6 Summary of contributions

The proposed work would lead to the following outcomes:

- A survey on existing audio applications and an analysis on their main traits and design issues to be solved.
- The definition of an architecture that would allow to develop real-time audio and music processing applications.
- A concrete implementation of such architecture within the CLAM framework.
- An implementation of audio related widgets for the Qt framework that could be reused.
- Re-implementations of existing open source audio applications by using the implemented architecture.
- A systematic qualitative analysis of several facets of the tool.
- A quantitative analysis of the development process of a set of existing open source audio software contrasted with the development process that uses the architecture.

1.7 Thesis organization

This chapter has introduced the context of this work and it has set its goals and methodologies.

Chapter 2 describes related work about tools and methods to make the software development more efficient. It also explains how other authors have faced the the specific issues of audio software engineering and some of the tools that are available for such domain.

Chapter 3 does a domain analysis on the audio applications family of systems. The goal is to obtain a set of abstraction and related engineering concerns to be applied to analyse the engineering needs of a given audio application.

Chapter 4 describes the prototyping architecture at different levels of detail.

Chapter 5 evaluates the architecture by analyzing how it performs in several real use cases.

Finally, Chapter 6 includes conclusions, the main contributions and further perspectives of research.

Chapter 2

Related Work: Application Development

This chapter reviews in the state of the art of application development and more concretely in audio application development. First, we review several general approaches that enable the efficient development of applications such as the use of frameworks, visual languages or domain specific languages. Then we introduce two concrete technologies such as data-flow languages to build audio processing applications, and visual GUI builders. We also review the state of art in the evaluation of such tools. Finally we review literature about specific engineering concerns and existing development environments for the audio domain.

2.1 The evolution of frameworks

Tools are one of the factors of the development process that can be modified to get an impact on its efficiency. Frameworks are very valuable tools to consider since they let you reuse both design and code. Roberts and Foote [**?**] define a framework as "a set of classes that embodies an abstract design for solutions to a family of problems". Other definition of framework, which emphasizes more the means than the goal is "a reusable design of all or part of a software system described by a set of abstract classes and the way instances of those classes interact"[**?**]. The later definition spots the fact that frameworks are not just a collection of classes but also a design on how they collaborate.

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. The first object-oriented frameworks to be considered as such are the MVC (Model View Controller) for Smalltalk [**?**] and the MacApp for Apple applications[**?**]. Other important frameworks from this initial phase were ET++[**?**] and Interviews. Most of these seminal frameworks were related to graphics or user interfaces.

Roberts and Johnson [**?**] explain the evolution patterns of a development framework. According to them, frameworks should follow certain evolution which involves incremental abstraction and refinement. First stages of a framework should consist in very few and simple abstractions from several existing applications. On later stages, the abstraction would be so high that the developer would be able to build a system just by using a domain specific language or visual builder.

A good visual builder should allow a domain expert with no programming knowledge to build a system just by drawing the components together just by using graphical conventions of the domain.

2.2 Visual programming languages

A visual programming language (VPL), is a programming language which uses relations and placement of objects in a 2D screen to articulate the execution of programs as opposed to textual programming languages (TPL) which uses linear text streams. Some illustrative general purpose VPLs are ProGraph and LabView.

Often their superiority to text based programming languages is defended with arguments such as:

> Pictures are superior to tests in a sense that they are abstract, instantly comprehensible and universal. [**?**]

Some critical authors call those claims the *superlativist hypothesis* (VPL are better than TPL) and the *accessibility hypothesis* (information in a diagram is instantly comprehensible an universal). Menzies [**?**] and others warn that those claims are based on intuition and lack of scientific criteria. Menzies reports that studies comparing visual programming languages

with textual programming languages often reach contradictory conclusions in similar conditions.

An study of Green, Petre and Bellamy [**?**] rejected both claims. They rejected the accessibility hypothesis by observing that novices found more troubles reading a visual program than experts. They also rejected the superlativist claim by demonstrating that for some tasks TPLs outperformed VPLs. To explain why VPLs perform differently for different programming problems, Green and Petre formulated the *match-mismatch conjecture* that states that the efficiency of a visual language depends on how well it maps the problem.

Moher et al [**?**] go further by stating that the effectiveness of a language not only depends on the target program but also on which task is to be done with the program. An illustrative example is forward versus backward reasoning. Some notation could facilitate the forward reasoning, given some inputs which is the output, but it could make backward thinking hard, which are the inputs that gave that output. Backward reasoning is very useful for some programming tasks such as debugging.

In a later study, Green and Petre [**?**] used a relation of different cognitive dimensions used to evaluate notations[**?**] to evaluate visual and text languages. Such dimensions were:

- Closeness of mapping
- Viscosity
- Hidden dependencies
- Hard mental operations
- Premature commitment
- Secondary notation
- Visibility
- Consistency
- Progressive evolution

The *closeness of mapping* is the distance between the domain and the programming worlds. The need of using programming specific entities, forces the programmer to think in programming terms instead of domain terms.

The *viscosity* is the effort that user has to do to effect a small change. It is normally measured as the number of primitive actions performed, but as it is task dependant, it is hard to evaluate unless there is a clear trend. They identify the layout as a source of viscosity in visual languages.

The *hidden dependencies* evaluates how a part of a program is affected by the changes of another in a way that is not obvious to the programmer. For example, function side effects. Green et al. notice that such dependencies are harder to highlight on textual languages but that they also can be found on visual languages.

The *hard mental operations* criteria evaluates how hard is to understand a combination of primitives. For example, the amount of work to understand a conditional expression could be different if we represent it as English text, as a truth table or as a logic gates circuit. Also the kind of reasoning to perform affects the amount of work.

The *premature commitment* criteria tells whether the language forces taking decisions before the required information is available. For example, visual languages tend to force the user to guess the layout of the final design.

The *secondary notation* criteria evaluates which are the alternative means that programmers can use to communicate aspects of the program that are not explicit on the language notation. For example, although they are not required to, textual programmers indent to denote structure, insert blank lines to group code or use naming conventions to denote aspects of the named entities. Green spots the opportunities of visual languages on adding such secondary notation but also notes that mastering secondary notations is hard to novices.

The *visibility* criteria evaluates which is the required cognitive work to make a required aspect of the program accessible. Total visibility is not convenient but accessing a non-visible aspect should not be hard.

The *consistency* criteria evaluates how easy is to infer one part of the language by knowing the other part. Consistency is a key aspect to facilitate learning.

The *progressive evolution* criteria evaluates whether the programming is able to test a part of the program before having it completed.

Of course, all those criteria are still subjective and very dependant on the task and on the programmer skills. But at least they provide a systematic and multi-faceted way of evaluating languages.

Previously cited works concluded that the effectiveness of visual languages is dependant on the task. Burnett et al. [**?**] say that most visual languages, even being successful for the task, have to face what they called the *scaling-up problem*. As the problems to solve get bigger and more complex, several issues tend to appear when using visual languages. They identified such issues and collected common solutions when available. The issues were classified as representation issues or programming issues. Representation issues include the availability of static representation, the effective use of the screen, or the suitability of the documentation facilities. On the other hand, programming language issues include procedural abstraction, interactive visual data abstractions, type checking, persistence and efficiency.

On a later publication [**?**], Burnett describes how visual languages will require, or will enable, the development of new software engineering tools. She explicitly talks about new documentation and code comprehension tools, and new testing, debugging and reusing tools.

My own opinion is that, textual languages are often specifications which are implemented by several vendors who provide their own development environment. The development environment is not part of the language but affects a lot to its usability. Conversely, visual languages definition are often coupled to the tool, and most of the previous concerns about the usability of the language refer to such tool.

This coupling to the tool, leads us to a new drawback with visual languages. Languages coupled to a tool are often coupled to a single vendor. The only rare case is UML and related standardization method, but UML standard is not executable although some trends tend to use it as executable specification [**?**].

2.3 Domain specific languages

Domain-specific languages (DSL), as opposed to general purpose languages (GPL), are languages that are restricted to a concrete domain. Deursen et al define them with the following statement:

> A domain-specific language (DSL) is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain. [**?**]

The actual promise of DSLs is the focused expressive power. They allow solutions expressed in the idiom and at the level of abstraction of the problem domain. They are not usually focused on execution, instead they tend to be declarative[**?**] on the facts of the domain. Some of the benefits of using DSLs are:

- Domain experts can understand, validate, modify and often develop DSL programs.
- DSLs capture domain knowledge directly so there is no specification to program mismatch.
- DSLs allow optimization and validation at the domain level.[**?**]

But there also some drawback on their usage such as:

- It might not be a DSLs available for the target domain.
- There is a cost on designing, implementing and maintaining a DSLs.
- They need user training.
- Their use may imply an efficiency loss compared to hand-coded software.

According to Deursen and Klint [**?**], the creation of a DSL typically involves the following steps:

- Analysis:
  1. Identify the problem domain
  2. Gather relevant knowledge in this domain
  3. Cluster this knowledge in a handful of semantic notions and operations on them
  4. Design a DSL that concisely describes applications in the domain
- Implementation
  1. Construct a library that implements the semantic notions
  2. Design and implement a compiler which translates DSL programs to a sequence of library calls

The key aspect of the process is the analysis. Neighbors [**?**] identifies the role of the *domain analyst* which is similar to the *system analyst* but instead of producing one single system is able to support the development of families of related systems. This require expertise having built several applications within the domain as a system analyst.

We can merge the idea of visual programming languages with the idea of domain specific languages. Domain-specific visual languages should be able to use visual domain specific notations to specify a family of programs. By addressing a concrete family of programs and by using domain-specific notations, the problem mismatch, introduced in section **??**, could be reduced. Also by using different language to specify different aspects of a program could reduce the task dependant mismatch.

In the following sections we present two common visual approaches to address different aspects of an audio application. On one side, data-flow languages to address the processing aspect, and the user interface visual builder to produce user interfaces.

2.4 Data flow languages

Data-flow models have a long tradition on system engineering. Signal processing area does an extensive use of them.

Visual builders that follow the data-flow paradigm are often called data-flow languages. Several of such data-flow languages exists for the audio and music domain. Beside being close to signal processing experts domain, data flow languages has more advantages. Firstly, being visual languages, a developer can get, at a glance, insight of the structure of the system. Data-flow languages also make more difficult to generate syntactically badly built systems. The language that such syntax generates is large enough to express a wide set of systems. Large interconnected systems are hard to understand visually, but the black box idea enables grouping a subset of interconnected subsystem as a subsystem itself and thus the implementation can be more scalable. And last but not least, having strict interfaces between subsystems, eases to reuse them in a different system.

On the other hand, data-flow languages just describe the data dependencies. Procedural details of the modules and their semantics need to be indicated using a secondary notation such as different iconic representation,

naming, port coloring...

Amatriain [**?**] described a Metamodel for Multimedia Systems (4MS), an object oriented model to model audio and music data flow systems. Arumi [**?**] compiled a set of design patterns that addressed several design challenges one can find when trying to implement data-flow systems on the audio domain.

2.5 Visual user interfaces builders

So, data-flow is successful on providing a design language for application processing algorithms. What about building products up to the public? Commonly, audio and music products need a user interface to give the user control over the application and to provide feedback on what's happening on the system. So, that is a two fold function: control and visualization.

Often data-flow prototyping tools offer integrated controls and visualizations to plug into the data-flow. So, you might consider releasing the data-flow prototyping tool as the product. But, that will blur the functional intent of your product. Although this kind of interface could be perfectly suited for power users, it gives too much access to the inners of your product: User interface elements for data-flow building are adding noise to the user interface elements that the user is intended to use, that is control and visualization user interface.

A proper user interface can be prototyped visually. In fact, user interface domain was one of the first domains to be provided of visual builders [**?**]. Visual interface building consists on visually setting the layout of the set graphical interface elements and setting their static properties. Some limited dynamic behaviour can be specified by using an event language [**?**]

This kind of prototyping shares a lot of the advantages with the data-flow based prototyping for the processing core but for the user interface domain. The resulting system is also a visual combination of the domain entities, which can be extended by the developer.

But visual user interface builder does not solve the full application building. It just solves the layout of graphical elements, their static properties and some responses to events that can be solved within the interface. Application logic is to be implemented by hand using the low level language the prototyping tool translates the prototype into.

2.6 Evaluating tools

Most of the tools and techniques presented before promise some benefit on the development process. But in the past some promising tools did not succeed even they provided some clear benefit.

Myers et al. [**?**] analyzes past trends in user interface tools and identifies the traits that made each trend successful or not. They state that such traits can be used as criteria to foresee whether a current or future tool is going to be successful.

One of the main criteria is the *target problem*. The key element to know if a tool is at least promising or not is to analyze whether the target problem is a key problem on development or not, and whether the tool address it thoroughly and effectively. But that's not the only criteria in order to make the tool successful.

Other two important and related criteria are the *threshold* and the *ceiling* of the tool. The threshold indicates how hard is to learn the tool. The ceiling indicates how far you can go with it. The ideal tool would have a low threshold and a high ceiling. But those two concepts are closely related. When designing a tool often happens that by raising the ceiling we are also raising the learning threshold and the other way around, when lowering the threshold we are reducing the ceiling. This happens in a natural way as the tool is likely to put more elements into the game in order to be able to model a wider range of applications.

Myers observes that a cost-benefit analysis is not enough to justify a high threshold. Most users will not get pass it. He also gives two means to get high ceilings without raising the threshold too much: One is offering a trap door, which does not affect the usability of the regular tool. The other is offering an smooth path which allows progressive raising of both the threshold and ceiling.

Other interesting criteria is the *path of least resistance*. This criteria tells that a tool has more chance to succeed when eases more the proper way of doing things than the dirty or unsuitable one. That applies to both the resulting product and the development artifacts. For example, toolkits made it easier to reuse than to build components from scratch and to have a consistent look and feel, visual builders made it easier to separate the application logic and the layout logic, and event languages made it easier to

build mode free interface than modal ones.

The last but not least criteria is the *moving target*. Technologies evolve fast and tools are reactive, the problem first appears and then someone thinks on a tool to address it. Also mastering the tools is something that may take long, so the target problem could have loose its importance before a critical mass of developers effectively use the tool that address it.

They observe that, in the case of user interface tools, for some years there has been an anomaly in this criteria due to the standardization of the desktop user interface. This exceptional situation let the tools mature. They warn that this situations is likely to change in the following years with the appearance of the ubiquitous computing and recognition interfaces.

2.7 Audio Software Engineering

Developing audio and music software implies addressing some specific issues. Extensive literature exists which analyzes the different software engineering aspects of audio applications. This section does an overview on it.

Pennycook [**?**] describes the challenges of developing interfaces for musicians as they must support creativeness instead of coercing it. He does a survey on several user interfaces for audio and music software. The reviewed software is now obsolete but some of the insights are still valid. He identifies several categories: composition and synthesis languages, graphic score editors, performance instruments, digital audio processing tools, and computer aided instruction on music systems.

In several papers, Dannenberg et al. [**?**] [**?**] [**?**] [**?**], analyzes software engineering concerns in real-time multimedia systems including the handling of incoming events, timing, low latency and other more general engineering concerns such as portability, reliability and ease of development.

> Real-time systems are commonly regarded as the most complex form of computer program due to parallelism, the use of special purpose input/output devices, and the fact that time-dependent errors are hard to reproduce. [**?**].

Dannenberg notes that the application should not wait for input as time and data dependent computations must take place, so he proposes a event driven architecture which inverts the control flow: instead of the program

asking for incoming events, the systems calls the program whenever an event comes. This introduces new concerns on preemption and multi-threading communication. He also spots the problem of memory management, as the costs of standard memory allocation and deallocation is not deterministic. He proposes preallocation of memory and an algorithm to handle such memory in real-time conditions.

Hardware abstraction and portability is other source of engineering issues. Bencina [?] abstracts common services to be provided by audio devices under the PortAudio API. Scavone [?] offers similar services under a object oriented API.

Audio analysis software have different needs than real-time software. While not having to deal with real-time restrictions they have to deal with more complex processing flow which is harder to generalize. Tzanetakis and Cook [?] describe architectural needs of audio analysis applications for audio information retrieval (AIR) presenting a general architecture to fit such needs.

2.8 Development environments for the audio domain

This section will give a brief survey of existing frameworks and environments for audio processing. Most of these environments are extensively reviewed in [?].

The current arena presents a heterogeneous collections of systems that range from simple libraries to full-fledged frameworks and development environments. 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 we will start listing the relevant environments not only for audio but also for image and multimedia:

- *Multimedia Processing Environments*: Ptolemy [?], BCMT [?], MET++ [?], MFSM [?], VuSystem [?], Javelina [?], VDSP [?]
- *(Mainly) Audio Processing Environments*: CLAM [?], The Create Signal Library (CSL) [?], Marsyas [?], STK [?], Open Sound World

(OSW) [?], Aura[?], SndObj [?], FORMES [?], Siren [?], Kyma [?], Max [?], PD[?]

- *(Mainly) Visual Processing Environments*: Khoros-Cantata [?] (now VisiQuest), TiViPE [?], NeatVision [?], AVS [?], FSF [?]

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 G. Tzanetakis is probably the most important framework in this sub-category as it has been used extensively in several Music Information Retrieval systems [?].

   (b) Synthesis Oriented: Audio processing frameworks that focus on generating output audio from input control signals or scores. STK by P. Cook [?] 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 [?] and CSL [?] are in a similar position, having in any case some advantages and disadvantages. CLAM, the target framework of the experiments in this thesis, 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 [?] 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 [?] family or Kyma[?].

4. *Music languages*: In this category we find different languages that can be used to express musical information (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 base 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 [**?**].

2.9 Summary, current directions and hypothesis

In this chapter we reviewed existing literature in several areas. On one side we considered means to make the development of general applications more efficient. We observed that frameworks and domain-specific languages succeeded on addressing a concrete domain, while visual language just give some benefit if are suited to the target program and to the development task. We reviewed literature about two domain-specific visual languages: data-flow systems and visual interface builders. We saw that they performed their task perfectly but there is a gap between them that still is not covered by such tools.

A first hypothesis of this thesis is that we can formulate an architecture which fills the gap between both visual builders to build a full audio application without text programming.

On the other hand, we did a review on existing literature about the concrete engineering concerns of audio applications and how to address them.

A second hypothesis of this thesis is that such issues can be systematically addressed and in some cases automatically covered given a high level description of the application logic requirements.

Chapter 3

Audio Applications

This chapter does a domain analysis on the family of systems that we call audio applications. This analysis covers the set of applications to be modelled visually by the prototyping architecture, real-time audio applications, but also extends to some other applications whose development could benefit from such prototyped components.

To get into that point, we analyze common and specific aspects of audio applications, and their related implementation issues. Such aspects range from data exchanges with the outside world, data and time dependencies, in-memory representations, user interface... Then, to probe the validity of such abstraction, we use them to describe some common archetypes of audio applications. Finally we define, based on such abstractions, which are the target applications of visual prototyping and how visual prototyped components could become the building blocks of some other audio applications.

3.1 Environment data sources and sinks

This section describes an abstraction how an audio application sees the environment provided by the system.

Normally an audio application is such because it deals with some audio related data sources and sinks: soft synthesizers receive MIDI events coming from other sequencer application, and they send an audio stream to an audio device, while a karaoke application reads time aligned lyrics from a file and displays it with synchronized coloring while it reproduces the song. Audio applications may interact with different sources and sinks of such audio

related data. Each one has different requirements on how to be handled:

- Audio streaming
- Asynchronous control events (MIDI, OSC...)
- Serialized information (audio files, meta-data...)

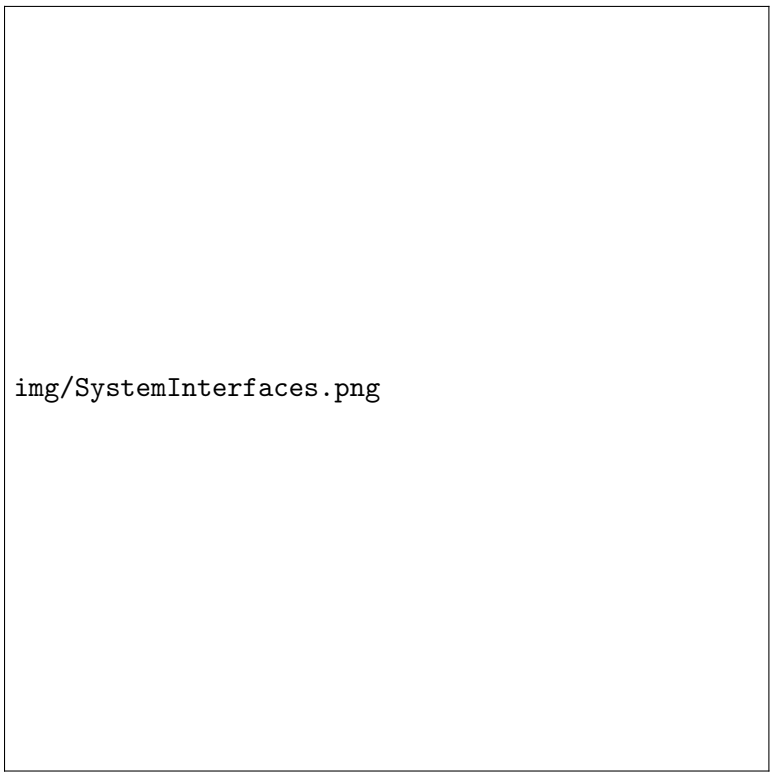

Figure 3.1: Audio applications takes and feeds data in different forms with the system and with the other applications. Communication have different requirements depending on the type of interface.

3.1.1 Audio Streams

The most common data communication of audio application with the outside world are audio streams. Traditionally, audio streams directly come from (or go to) the audio device driver offered by the system. See figure **??**. Several programming interfaces for audio devices are available. Most

of them are platform dependant such as ALSA (Linux), OSS (Unix), Audio Core (MacOs), WMME, DirectSound, ASIO... Some libraries, such as PortAudio[**?**] and RTAudio[**?**], provide a cross-platform view of them all by abstracting common features such as device enumeration, selection, and setup.

Figure 3.2: Device drivers audio streams: Traditionally, audio streams are provided by system device drivers.

A special kind of application, the audio plug-in, does not take the audio streams from audio devices but from a host application. An audio plug-in is an independently distributed library that can be loaded by any host application. See figure **??**. The audio plug-in concept had been used by several audio software as a way of extending their capabilities. The concept gained popularity with the publication of API specifications such as VST that allowed not just developing plug-ins but also hosts applications that could load such plug-ins, fostering the reuse of plug-ins across applications. After VST, many other standards appeared such as the free software based

LADSPA and DSSI, or the Audio Units standard on Mac.

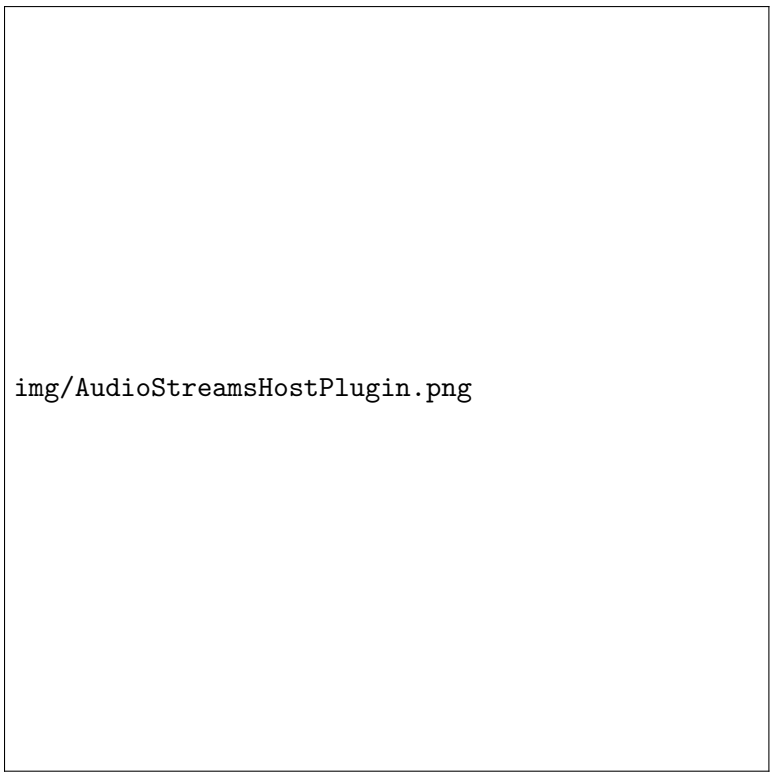

Figure 3.3: Host application is the audio stream provider of the audio plug-ins it loads.

Plug-ins have an asymmetrical relation with their host. Conversely, inter-process audio communication standards such as JACK[**?**] enable pear-to-pear communications among applications: Any conforming application can use any other as sink or source for its audio streams. See figure **??**. This allows the connected applications overcome the inherent limitations of being a plug-in. They can have their own application logic and they can communicate with several applications, not just the host.

Other application use the network as source or sink of audio streaming. Internet audio broadcast applications are examples of application using the network as audio sink. Also most modern audio players are able to play streaming audio coming from the network. Network streaming has to deal with the fact that a given audio frame is not guaranteed to get its destiny in a bounded interval of time. Some internet protocols such as RTP[**?**] deal

Figure 3.4: Inter-application audio communication APIs allow use other applications as source and sinks for audio streams.

with such limitations but other applications need to reuse regular protocols. Software layers such as GStreamer [1] provide a layer that abstracts the idiosyncrasy of such sinks and sources offering application a continuous stream.

Audio streams are accessed through an application programming interface (API). Two common styles of API's are commonly used. In a blocking audio API, the application ask for reading or writing an audio chunk and blocks until the input audio is available or the output audio is required and fed into the device. Of course, the audio application must handle other actions while waiting for the devices, so normally the audio is handled by an independent thread. Callback or event based API's, instead, tell the system to call a given application function (the callback or the event handler) whenever the audio stream is available to be accessed. PortAudio, RTAudio, and ALSA support both styles of API's.

One of the main traits of audio streams is that they provide or require data at a fixed pace. If data is not read or written at such pace, the application misbehaves. The time slot is equivalent to the time duration of the data that is to be written or read (see figure **??**). *Read over-runs* happen whenever the application is not able to read a data source within its time slot while, *write under-runs* happen whenever the application is not able to write into a data sink within its time slot. Both are named generically *x-runs* and they normally are perceived as hops or clicks on the sound.

Even when the mean throughput of the CPU is enough to execute the whole processing in less time than the audio duration, the processing may require CPU peaks that go beyond the time slot as in figure **??**(b). Also, in systems not enabled for real-time, other threads, processes or even the operating system, may take the time slot, not letting the audio processing to be done on time (figure **??**(c)). In both cases, the effect of x-runs can be minimized by adding extra buffering. Figure **??**(d), shows how buffering absorbs the x-runs at the cost of latency, that is, the time period since an input event happens until one listens its effects on the output. But as figure **??**(e) shows, the number and the relevance of the x-runs to be absorbed depends on the size of the buffering which in turn increases the latency. For applications not requiring low latency, buffering can be very high. But as discussed later, on some applications high latencies are a problem and a

[1] http://gstreamer.freedesktop.org/

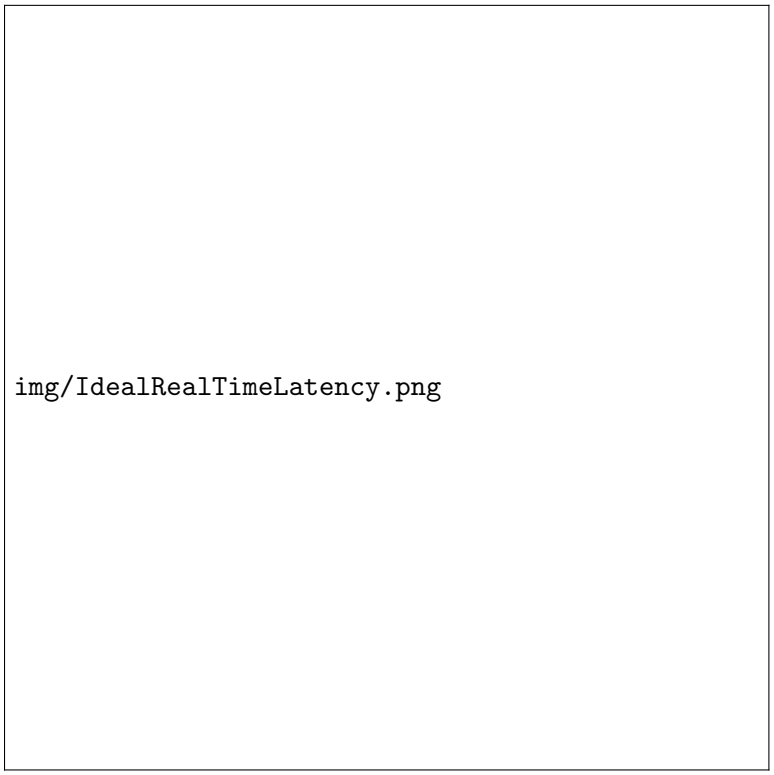

Figure 3.5: Real-time processing requires the processing of a block to be executed within the duration of such a block.

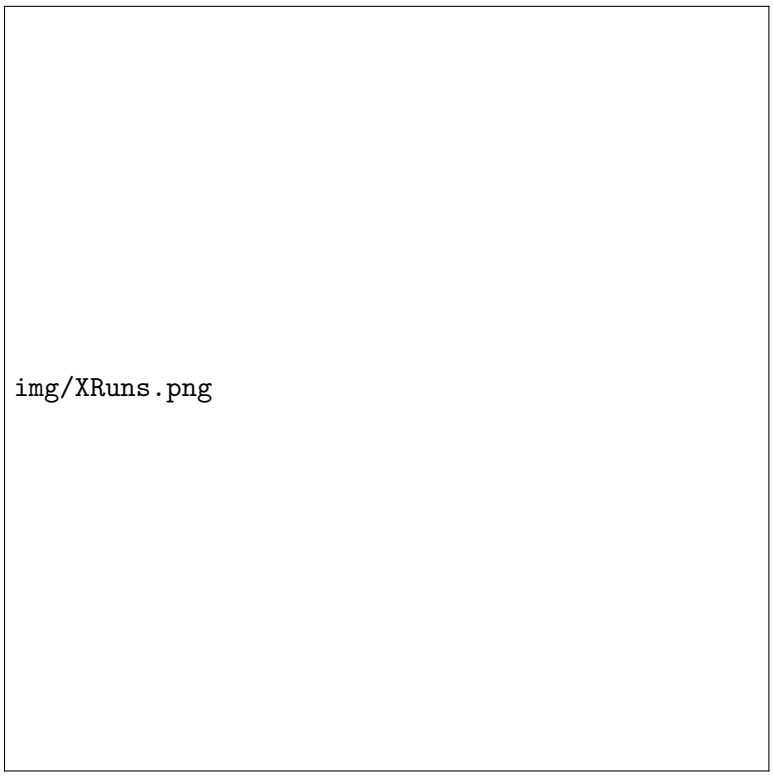

Figure 3.6: Time-lines for (a) a well behaving real-time execution, (b) x-run due to a processing peak, (c) x-run due to non-real time scheduling, (d) one period buffering adding latency but absorbing x-runs c and d, and (e) one period buffering not able to compensate several adjacent x-run conditions.

trade-off is needed.

In this section, we have been discussing about audio streams. But, of course, most of those considerations can be extended to any synchronous data transfer not being audio. For example, a visualization plug-in may receive from the host player spectra to be visualized at a constant pace.

3.1.2 Asynchronous control events

A different, but also very common, data transfer between audio applications and the outside world are asynchronous control events such as MIDI events [**?**] sent or received by external MIDI devices, or OSC [**?**] messages sent or received by local or remote applications.

They are asynchronous because in contrast with audio streams, they are not to be served in a continuous pace. Receiving audio applications don't know before hand when events will happen. This implies a different kind of problems. Control events may come in bursts, so, when an application also has to deal with real-time synchronous stream, we can get into problems if serving a control event is too expensive. Also, while you can implicitly know the time position of audio streams, just by their ordering and the sampling period, there is no implicit way to know the timing of control events.

Whenever an audio application deals with control events, latency is a concern in any dependant audio stream. For example, a MIDI controlled software synthesizer will misbehave if a note starts too much time later than the user pressed the key on the MIDI keyboard. Live performance is hardly affected by this delay. Brandt and Dannenberg consider acceptable latency of some commercial synthesizers between 5 and 10 ms which is comparable to delays due to acoustic transmission [**?**].

An additional concern with control events is the *jitter*. When we have two dependant input and output audio streams, the latency is constant and depends just on the buffering sizes [2]. When an output audio stream depends on an input control, latency may be variable producing an effect called *jitter* (see figure **??**). Some studies reported the minimal noticeable jitter on a 100 ms rhythmic period is 5 ms[**?**], 1 ms [**?**] and between 3 and 4 ms [**?**]. Of course, not all events has the same effect on the audio. The ones in the studies were onsets of a percussive sound. Jitter perception in other kind of

[2] It also can be variable due to poorly designed audio devices.

events such as continuous control may be less sensitive.

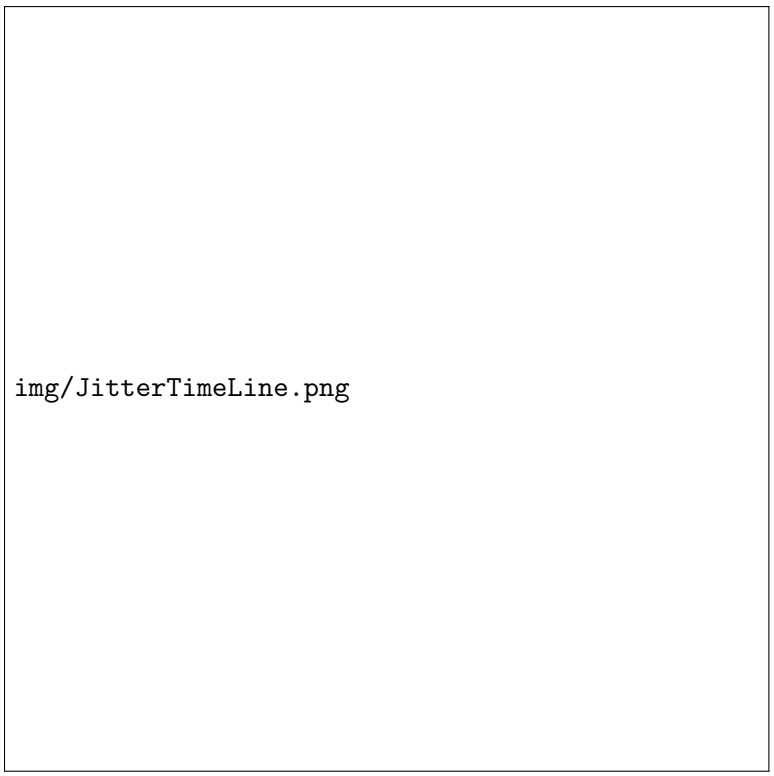

Figure 3.7: Different latencies on serving control events results on control jitter.

Dannenberg describes a solution to the jitter problem [**?**]. If we can get time-stamped events from the device, we can set a compromise maximum latency, and applying all the events with constant delay from the time-stamp. Time-stamping also gives problems when the time-stamp clock and the clock reference for real-time audio does not match. Adriaensen [**?**] proposes a solution to correct the clock mismatch.

3.1.3 File access

Another important outside information flow is the one involving files: Mainly audio files but also files containing meta-data, configuration, sequencing descriptions...

File access has two main traits. On one hand, and in contrast to audio streams and control events, they do not impose any real-time or timing re-

strictions on how to access data. The application can access file information without any time order restriction, and it can spend many time as it needs to do a computation related to a time slot. On the other hand, accessing files is not real-time safe, in the sense that there is no guaranties of reading or writing a file block within a bounded period of time. Thus, if a real-time process must access a file for reading or writing, it will need intermediate buffering, and dealing the file access in a different thread or after the real-time task in blocking API's. In this case, the buffering does not add latency to the output as file access is not restricted in time.

In summary, if we just deal with files, none of the previous real-time concerns apply, but as we relate them with data sink and sources imposing real-time restrictions we need to separate them with a buffer.

3.2 Real-time and off-line execution

Data and time dependencies an application has on external sinks and sources limit the kind of processing the application is able to do. Also, the processing an application is required to do, limits the data and time dependencies the application can have on external data sinks and sources.

For instance, consider an effect plug-in (figure **??**). An effect plug-in continuously takes sound coming from an input audio stream, modifies it and sends continuously the resulting sound to an output audio stream. Time-data dependencies of such architecture forbids the plug-in to do any transformation that requires information from the future of the stream. It is limited to *streaming processing*. A very simple streaming process is an amplification effect which multiplies every sample by a constant. No future data is required to apply the amplification to a sample.

On the other hand, audio normalization is a very simple example of transformation requiring future data, a *non-streaming processing*. Normalization consists on applying an audio the maximum gain without clipping. Applying the gain is a streaming processing but to know the gain you need to know the maximum level which might be beyond the current processed audio sample. Thus, the full audio excerpt must be analyzed before we can compute the output of the first sample. This is incompatible with the timing and data dependencies imposed by the audio plug-in architecture. [3]

[3]Notice that internal application buffering could add some limited look-ahead and

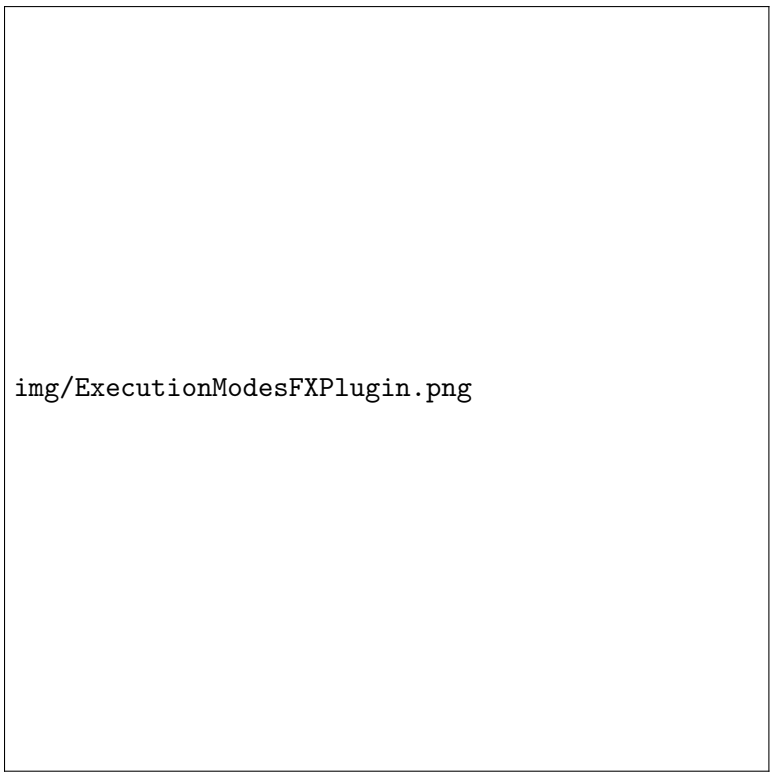

Figure 3.8: Data-time dependency in an effect plug-in

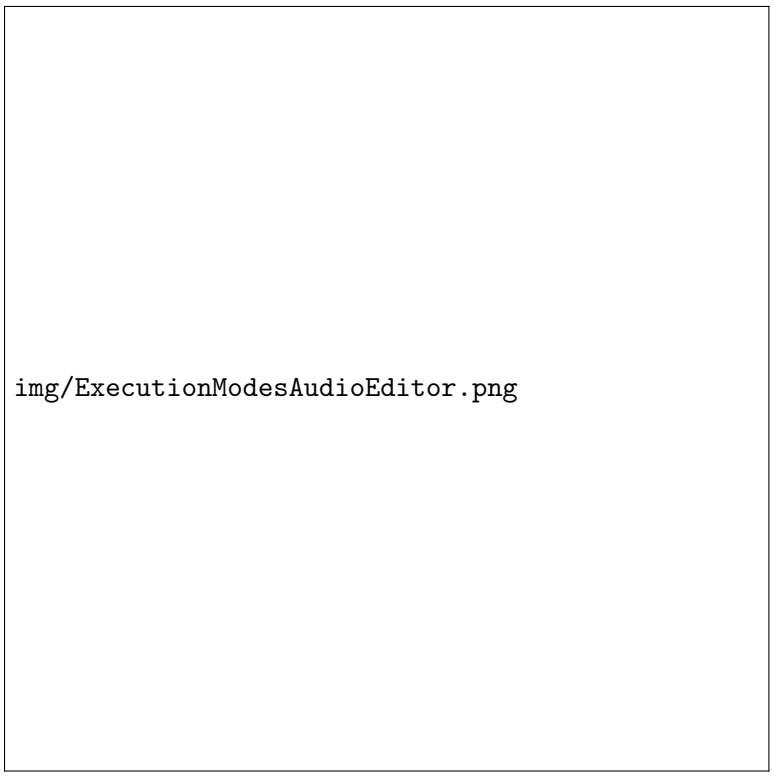

Figure 3.9: Data-time dependency in an audio editor. The internal audio representation breaks data-time dependency between audio devices, enabling non-streaming algorithms but disabling real-time processing.

To execute a normalization we should have to relax time-data dependencies between input and output audio streams. For example, an audio editing tool (figure **??**), can apply a normalization by doing it off-line. *Off-line execution* means that the execution of the process is not bound to any source or sink with real-time restrictions. To achieve that, the audio editor makes use of in-memory representation of the full audio. In-memory representation offers random access to time associated data. File access also would play the role, as explained in section **??**, but, it would need be on a different thread and communicating to the real-time playback using a ring buffer as in figure **??**.

Figure 3.10: Data-time dependency in an file player. File reading is isolated in a different thread using a buffer to avoid file system access to block the throughput. This buffering has no effect on real-time latency.

We can get more insight from the normalization example. The com-

extend the future scope at the expense of latency. But this is not enough to solve the normalization problem since we need to know the full excerpt.

putation of the normalization factor (the inverse of the maximum absolute sample value) can be computed in streaming. Being streaming could mean, for example, that it could be computed within the audio capture process. The process has no frame a frame output but a summary output. But still the output can be computed in a progressive way.

On the other hand, given the normalization factor, applying the gain is also a streaming process. We can see that summary computations can be streaming processes, but processes with dependencies on summary computations can not be streaming. We can see also that some non-streaming processes can be expressed by a set of streaming processes communicating summary outputs.

One conclusion of that is that if we have a way of expressing streaming algorithms, such as data-flow systems where the tokens are at frame level, some non-streaming algorithm could be built by connecting streaming algorithms which share tokens at song level. This is a useful feature to be able to reuse components built for a real-time system in an off-line system.

Like audio streams, input control events must be applied on time and output control events must be sent on time. So dependencies on asynchronous control events also imposes streaming processing.

3.3 Processing multiple audio items

Not always the dimension driving the execution of an audio related process is the time. Often, the main driving dimension is the audio item, meaning for example a song within a play list or an audio sample in a sample collection. Some examples for such processing is:

1. A multimedia player computing the normalization factor for all the files in the collection.
2. A learning algorithm accessing a lot of files to learn some concept.
3. A song version aligner taking two versions and locating the correspondence points.
4. A similarity algorithm comparing full collections with a reference item.
5. A collection visualization tool **??**, computing the distance between each pair of audio items.

Notice that every example has a different access pattern on the multiple items. 1 and 2 just applies some off-line process to a sequence of items. While 1 has an output for each file (the gain), 2 has an overall output for all the collection (the learned model). Notice that this has some parallelisms on summary computations along time. Examples 3, 4 and 5 have a core process which has two input items, and the main difference among them is how the items are feed to that core process.

3.4 Graphical user interface

The graphical user interface is an actual part of the application. But if we consider it as such, related information flow with the outside would consist just on drawing primitives and low level input events which are not attractive for this discussion. If we consider the full interface as an outside element (with some privileges), information flow will consist in audio relevant information, and this will enrich the description on what is happening with audio applications.

In real-time systems, a sane design places the GUI and the audio processing in separate threads. Having the GUI in a low priority thread helps to meet the real-time requirements by allowing the real-time processing thread to pop up every time it is required. Of course thread separation requires inter-thread communication. In the case, of real-time applications, such inter-thread communication must be lock-free for the real-time thread. Valois describes several lock free structures [**?**]. The commonly used one for thread communication in audio is the lock-free FIFO queue by Fober et al. [**?**] [**?**].

In off-line systems, thread separation is also useful because off-line processing may require too long periods in which the GUI would not respond. But thread separation is not the only solution for off-line systems. The alternative can be a rendez-vous policy: periodically the processing tasks let the user interface to respond any pending incoming user interface event. This simplifies the system not having to deal with thread management and inter-thread communication issues.

User interaction is closely related to the application logic. We identified several archetypical user interaction atoms. Depending on the application logic, several of those atoms will be required, and each interaction atom will

bring its own implementation concerns. The user interaction atoms are:

- Instant data monitoring
- Real-time control sending
- Transport control and feedback
- Visualization of data along time
- Edition of data along time
- Visualizing or managing audio items collections
- Configuration and setup

Following sections analyze the traits of each interaction atom and its implementation requirements.

3.4.1 Instant data monitoring

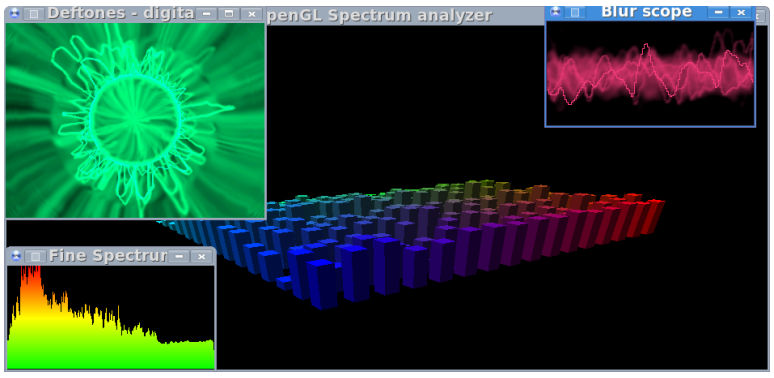

Figure 3.11: Examples of instant data monitoring: XMMS visualization plug-ins taking instant PCM and spectrum data.

Often audio applications need to display some data that is tied to an instant of the streaming audio. The vu-meter on an audio recorder application, or oscilloscopes and spectrum views on multimedia players (figure **??**) are common examples.

The common trait of such views is that they require regular updates of information coming from the processing thread. Thus, a thread safe communication is needed.

Also, visualizing every processing data is not required. Indeed is not even feasible as the processing token cadence is at the order of 10ms and the screen retrace at the order of 100ms. This, and the fact that the reading thread has no real-time constraints, will allow us to use a very simple thread safe communication mechanism instead of the typical lock free structures. Such mechanism is explained with more detail at section **??**.

3.4.2 Real-time control sending

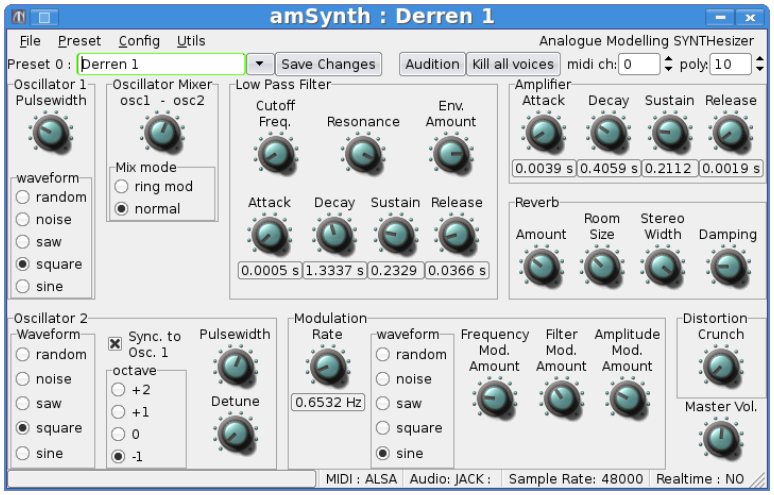

Figure 3.12: Some control knobs in amSynth, a software synthesizer.

A very common user interface functionality in audio applications is to send asynchronous control data that modifies processing in real-time. For instance, the knobs of a real-time software synthesizer, or the mixer slider of a DAW[4] which sets the recording gain of an input channel.

Such events coming from the interface should be taken as they where a real-time asynchronous event source. The problem here is that the latency of user interface events is huge because the user interface is in a non-real-time priority thread. Moreover such latency is very variable, because it heavily depends on the queue of graphical events to be served and that causes jitter. Often user interface events are not time stamped so it is difficult to apply them a constant latency to avoid that jitter. And when they are time-stamped they often use a different clock than the audio clock so the drift

[4]DAW stands for Digital Audio Workstation, instances are Ardour, Cubase, ProTools...

should be compensated.

3.4.3 Transport control and feedback

A set of user interactions affect the time the audio application is handling. Simple transport actions such as *play* and *stop* but also moving the play-head of an audio view or setting cue points, play regions and loops in a time line.

Figure 3.13: The transport control and feedback interface in Ardour 2.

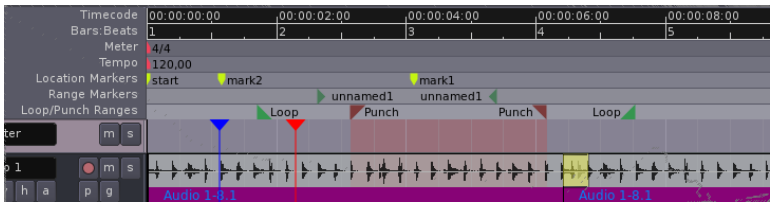

Figure 3.14: Detail of the Ardour 2 time bar. The red triangle is the play head indicating the currently playing position. The blue triangle is the cue position where the play head goes back when the playback ends. Loop defines a region to play and punch the region where the saving is active. Range and position markers let save them for later reuse.

At the same time, audio applications often gives feedback of the current transport status for example by moving the play head position, a transport indicator or by displaying the current time.

Transport interactions have sense when performing real-time tasks. But they also have their counter part in off-line processing. Play and stop have similar requirement to launching and aborting an off-line processing. Also, a progress indicator may have very similar behaviour to a transport indicator. But off-line execution might not be a linear time execution so in this case progress indicator might be something different than time.

When more than one audio item are involved, often audio item change is another transport control interaction for controlling and having feedback on. Transport changes can be handled in a similar way control sending is

done. The specifics for transport is that such interaction may need more interaction than simply sending events to the processes.

3.4.4 Visualization of data along time

Non-Instant data-time visualizations display data bound to time along the time. The most clear example is waveform visualization but also spectrograms or even the structural representation of a song. Such visualizations need a data source that is not real-time such as file access or in-memory representation.

Being just visualization the only interaction with the audio application are updates when other process changes such data.

When programming such views, other process modifying the in-memory representation could be an issue, but as the interface thread does not modify the representation, normally lock-free communication mechanism are not needed.

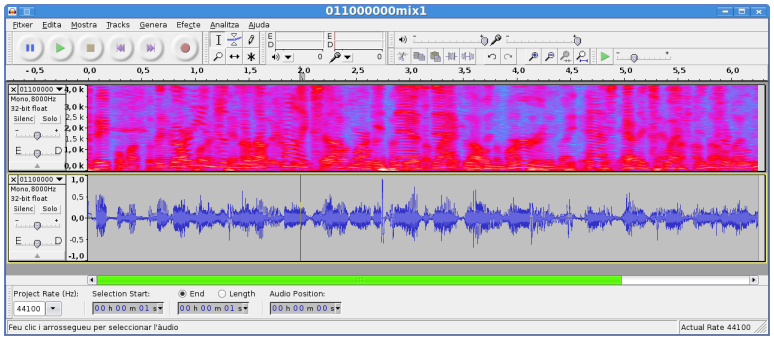

Figure 3.15: Several views from Audacity: Spectrogram and waveform.

3.4.5 Edition of data along time

Another common interaction between the GUI and the application is direct manipulation of time related data. For example, placing and editing notes in an score editor, arranging patterns in a sequencer or placing and moving waves along tracks in a digital audio workstation.

In edition, concerns applied to non-instant data-time visualisation also applies here: They require in-memory representation of the data, and data updates to the visualization. Besides that, edition is not just passive and

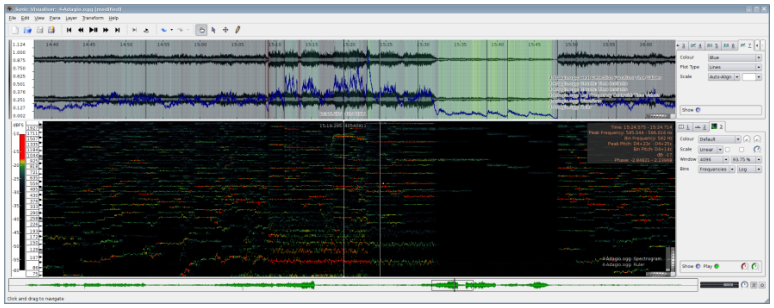

Figure 3.16: Sonic Visualisers presents static data along the time in many different ways.

it also modifies the in-memory representation, such changes could interfere with some running real-time or off-line process. Again, lock-free thread communication is needed.

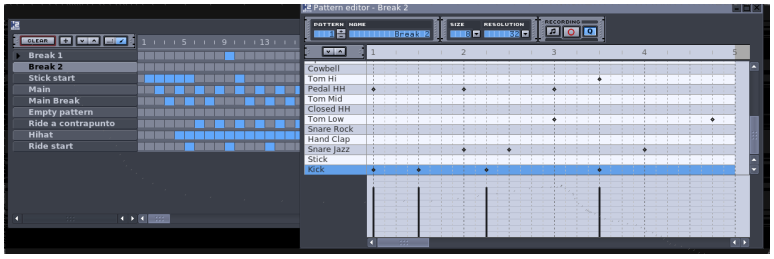

Figure 3.17: Hydrogen drum machine allows editing when the events will happen along the time.

3.4.6 Managing multiple audio items

Often audio applications have to deal with a collection of audio items and offers the user views to visualize or manage such collection. Simple examples could be a play list editor (figure **??**) or a sample collection browser. But such views could be more elaborated [**?**] as shown in figure **??**.

Also applications has views or edition interface for information related to a single audio item as a whole. For example, an application could allow to edit the ID3 tags of a song or adding custom labels. It could also display context information: artwork, artist biography and other meta-data. Such information is time independent, those components just need to know when

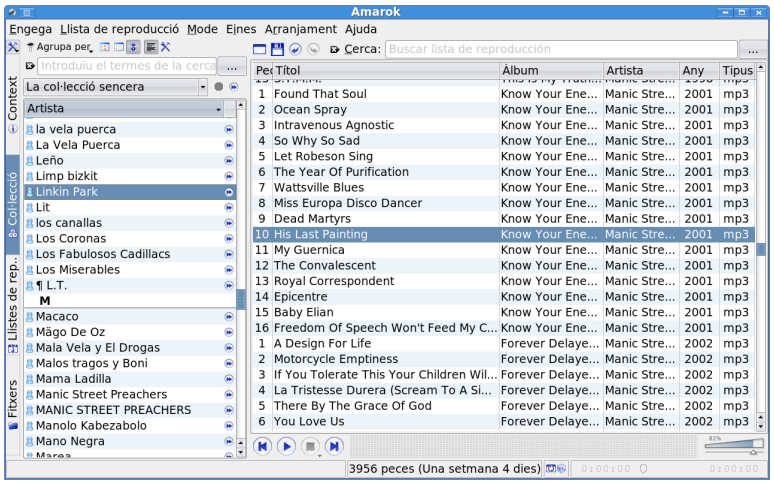

Figure 3.18: Amarok multimedia player showing the collection browser (left) and the play list editor (right).

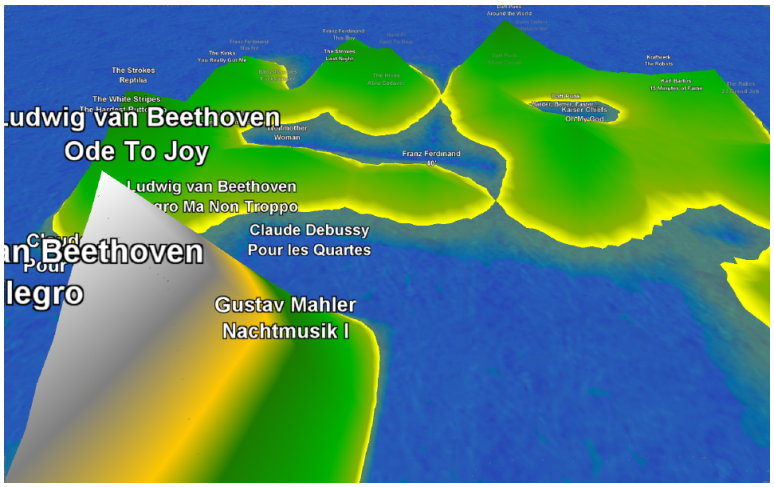

Figure 3.19: Three dimensional interface for exploring collections

they item information must be feed.

Because audio items visualization and management doesn't impose any real-time constraint, this kind of interaction is feasible for web applications. Some illustrative web applications dealing with audio items are Foafing The Music[**?**][5], Last FM [6], and Music Brainz [7].

3.4.7 Configuration and setup

Besides multi-item interactions, other set of interactions that are time independent are the ones for configuring and setup the application. For example, setting up a process before launching it, or setting up audio and event streams.

3.5 Audio application archetypes

This section identifies some of the functional roles of existing and foreseen audio applications, and describes them in terms of data transfers with their environment, internal representations and execution modes. The goal is to demonstrate that the former abstractions are enough to describe the internal application structure of a relevant group of audio applications.

The roles described below are very abstract. Actual audio software may be a combination of several of those roles.

3.5.1 Software synthesizers

Software synthesizers generate sound in response to incoming events. The application logic of this kind of software requires very few interactions with the user:

- Setting up the sound by controlling parameters, sound banks, presets...
- Receiving control events from the GUI (when available).

The main process of a software synthesizers is real-time constrained by a link between incoming control events stream and the audio output stream. Thus, events should be time-stamped and be applied with a constant delay.

[5] http://foafingthemusic.iua.upf.edu
[6] http://www.lastfm.com
[7] http://musicbrainz.org.

Some software synthesizes deal with input coming from the user interface. Such input events as explained in section **??** need an special care.

Also, the synthesizer could have some instant view of the produced audio. Figure **??** shows a software synthesizer called Salto which displays some instant views related to the produced sound.

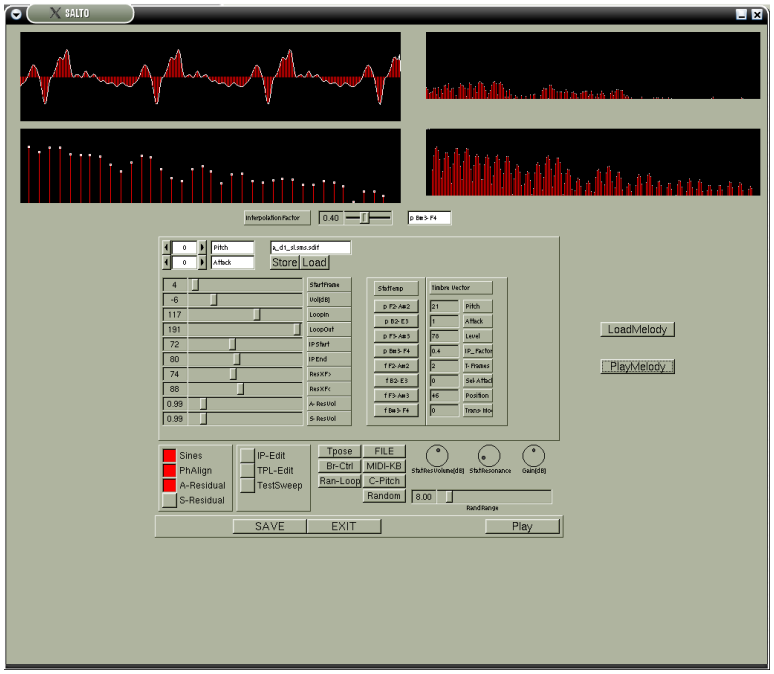

Figure 3.20: Salto, a software synthesizer developed at the UPF, which uses spectral models to synthesize sax and trumpet sounds.

Off-line synthesizers also exist but they are not as common as real-time ones. They are used when real-time rendering is not feasible, for example when complex generation models are used, such some physical modeling algorithms. In this case the control events come in a file representations. For example, in CSound the sound generator setup comes in the *orchestra file* while the control events are specified on the *score file*. MIDI files are also common event source for off-line synthesis.

3.5.2 Sound effects

Sound effects take the audio input and perform modifications to produce an output. They share most characteristics with sound generator but they also

must deal with the input stream which is real-time linked with the output. Recent audio effects tend to be audio plug-ins as this makes them more reusable.

Off-line processing is more common in processor than with sound generators. Nice examples of off-line effect processor are ecasound[8] and SoX [9].

3.5.3 Sound analyzers

Sound analyzers take some input sound and they extract a different representation. Such representation could have different uses. Real-time analyzers are used to give visual feedback on an audio stream. They are also used as control mean in performances. For example, we could detect the chords of the audio in order to perform an accompaniment arpeggio. This kind of tasks require real-time restrictions between the input and the output data.

Typical sound analyzers require few interaction:

- Analysis parameter setup
- Output visualization

Off-line sound analyzers are more frequent than real-time ones often simply because the processing load is to heavy for real-time requirements, but also because non-streaming processing is needed. For example, some analysis done in music information retrieval need having a global view of the excerpt or they have dependencies on summary results. Also music information retrieval systems (described below) require the output of analysis performed to many audio items, and such a batch analysis is done off-line.

3.5.4 Music information retrieval systems

Music information retrieval systems deal with a lot of audio items. In order to deal with such amount of audio material they use summarized information taken with some batch analysis process. So the abstractions on supporting multiple audio items in processing (section **??**) and interface (section **??**) can be applied to off-line analysis for music information retrieval.

[8] http://eca.cx/ecasound/
[9] http://sox.sourceforge.net/

3.5.5 Audio authoring tools

Audio authoring tools are applications such as Sweep, Audacity, GoldWave, CoolEdit... They respond to the schema previously shown on figure **??**. They have an in-memory representation of the audio and perform off-line processing to it.

Common interactions in this kind of software are:

- Recording audio
- Loading and saving files
- Transport management (play, stop, range selection...)
- Launching and cancelling off-line processes
- Playing audio

3.5.6 Sequencers

Sequencers are programs that allow the users to edit a representation of the structure of a song. Such structure contains data which is used to control internal or external sound producers, typically a synthesizer. They often have a recording mode to input events from an external device such a MIDI keyboard.

Free software applications that includes such a role are Rosegarden, MuSe, NoteEdit, Hidrogen, CheeseTracker... All those application use different paradigms on how the user edits the structure: score, piano role, pattern sequencer, tracker... Even thought, the concerns on edition are nearly the same.

Sequencers have to deal with several user interactions:

- Selecting or configuring the sound producers.
- Recording and editing events into the song structure
- Edit the song structure
- Transport control

This software has an in-memory representation of the events along the time. In normal playback, the real-time constraints apply to the control

sending. In-memory representation of the events enables using a forward scheduling. In recording mode, if available, often the input events are required to be forwarded to the output so they can have effect on the sound producer and provide feedback to the performer user. Because that, in that case, there is a real-time link between received and sent control events.

3.6 Real-time audio applications

Previous section has provided some insight on the inner structure of several archetypes of audio applications. In this section, we give deeper insight on the subset of real-time audio applications: How can they be modelled in a general way, which roles can they perform, and which are the main implementation concerns. We also explain how components of a real-time audio applications can be reused in other contexts.

Figure **??**, is a generalization of the family of applications that we call real-time audio applications. That is the set of applications whose visual building is addressed in this work. Such applications include *a single streaming process* that may deal with:

- several input and output audio streams,
- several input and output event streams (MIDI, OSC...), and,
- several input and output file streams.

The application could provide user interface for

- controlling transport (start, stop the process and seek on the file sources,
- having feedback on the transport state,
- instant data visualization,
- sending control events from the interface,
- setting up the sinks and sources to deal with (not in the figure).

This description fits the needs of some of the roles described on the previous section: real-time software synthesizers, real-time sound effects, and real-time audio analyzers.

Figure 3.21: A generic schema of a real-time application. The application contains a single streaming process which may take and send data from and to audio streams, control event streams or files. The application interface may provide instant data visualization, GUI control event sending and transport control and feedback.

Some other application archetypes such as authoring tools, sequencers, music information retrieval system and off-line systems are outside this description. The key features that make them out are:

- Non-streaming processing
- In-memory representations
- Multiple audio item processing
- Any other application logic that is more complex than binding to external streams, starting, stopping, control the transport on file based streams, sending control events and visualizing instant data.

Anyway, in section **??**, we explain how components built with the prototyping architecture can be helpful to build applications outside this scope.

3.7 4MPS Meta-model

This section goes into the details of the *streaming processing* element of the real-time application schema shown in the figure **??**. We describe a domain-specific meta-model called 4MPS. 4MPS is the conceptual framework for the data-flow language to be used in the prototyping architecture for the audio processing.

The Object-Oriented Metamodel[10] for Multimedia Processing Systems, 4MPS 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 section **??**) 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 section we provide a brief outline of the metamodel, see [**?**] for a more detailed description.

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

The 4MPS 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 4MPS system as a set of Processing objects connected in graphs called *Networks* (see Figure **??**).

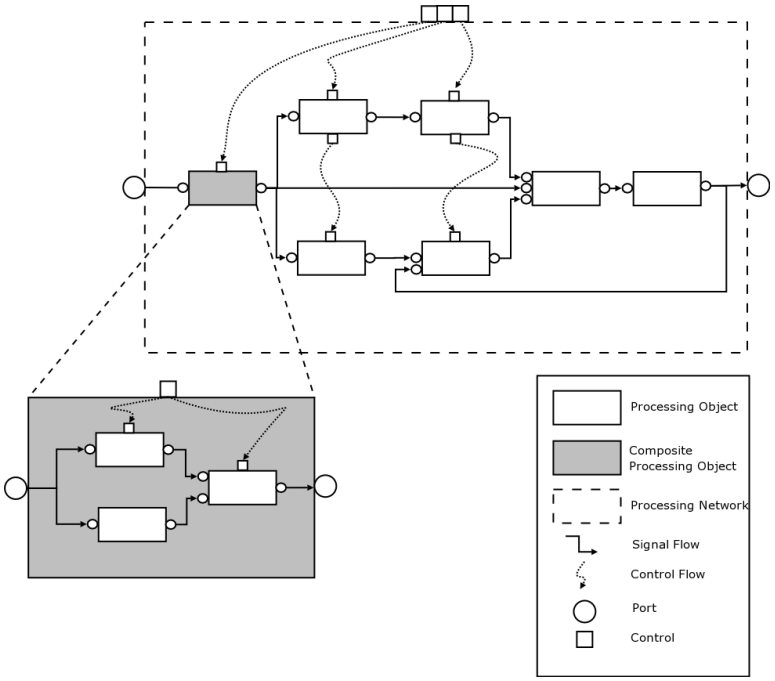

Figure 3.22: Graphical model of a 4MPS processing network. Processing objects are connected through ports and controls. Horizontal left-to-right connections represents the synchronous signal flow while vertical top-to-bottom connections represent asynchronous control connections.

Because of this the metamodel can be expressed in the language of graphical models of computation as a *Context-aware Data-flow Network* (see [**?**] and [**?**]) and different properties of the systems can be derived in this way.

Figure **??** is a representation of a 4MPS 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 FIFI queues in which messages are enqueued (produced) and dequeued (consumed).

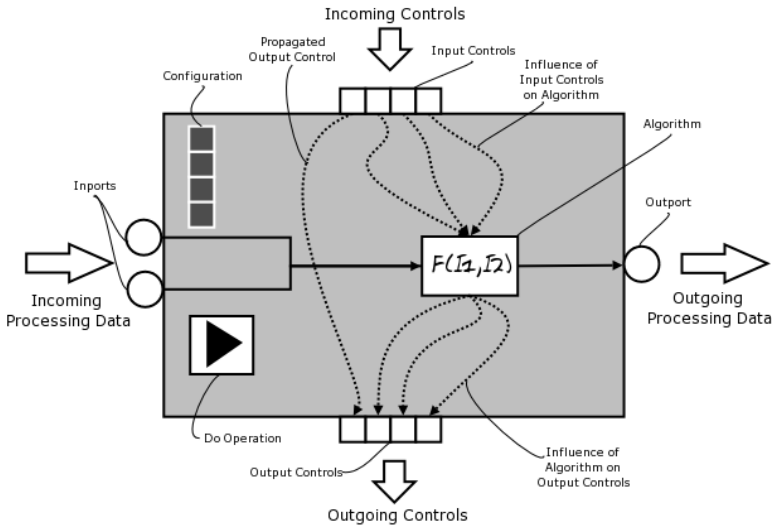

Figure 3.23: 4MPS Processing object detailed representation. A Processing object has input and output ports and incoming and outgoing controls. It receives/sends synchronous data to process through the ports and receives/sends control events that can influence the process through its controls. A Processing 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 data-flow 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 out-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 section **??**). 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.

3.7.1 Life-cycle and Configurations

A 4MPS 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.

3.7.2 Static vs. dynamic processing compositions

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

This process, known as composition, can be done in two different ways: *statically* at compile time, and *dynamically* at run time (see [**?**]). Static compositions in the 4MPS 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.

3.7.3 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 **??** 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 down-stream 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 data-flow networks and similar graphical models of computation (see [**?**] 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.

3.8 Towards a Pattern Language for Data-flow Architectures

3.8.1 Introduction

As explained in the previous section, the general 4MPS Metamodel can be interpreted as a particular case of a *Data-flow Network*. Furthermore, when reviewing many frameworks and environments related to CLAM (see

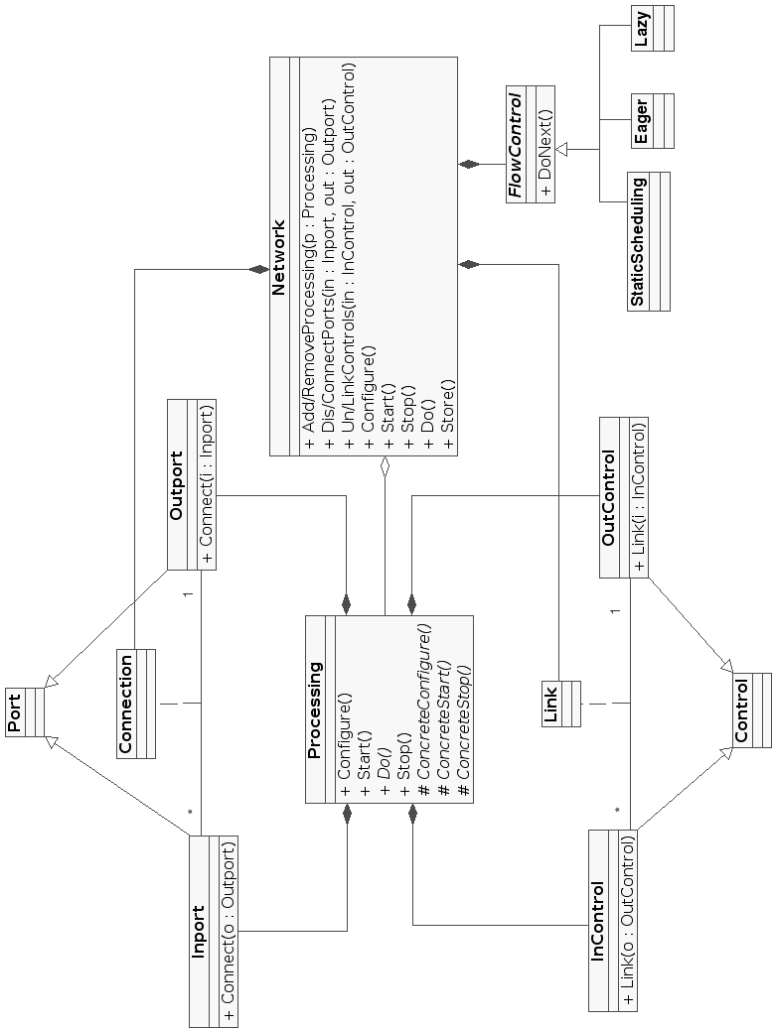

Figure 3.24: Participant classes in a 4MPS Network. Note that a 4MPS Network is a dynamic run-time composition of Processing objects that contains not only Processing instances but also a list of connected Ports and Controls and a Flow Control.

section **??**) we also uncovered that most of these frameworks end up offering a variation of data-flow networks.

While 4MPS 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 4MPS and the concrete implementation given a set of constraints.

Patterns provide a convenient way to formalize and reuse design experience. However, neither data-flow 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 [**?**] and a catalog with 6 real-time audio patterns presented in a Workshop [**?**]. In the general multimedia field there are some related pattern languages like [**?**] but these few examples have a narrower scope than the one here presented.

There have been previous efforts in building pattern languages for the data-flow paradigm, most noticeably the one by Manolescu [**?**]. However, the pattern language here presented is different because it takes our experience building generic audio frameworks and models [**?**] [**?**] 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 paragraphs we offer a brief summary of a complete pattern language for Data-flow Architecture in the Multimedia domain presented in [**?**]. 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 Data Flow 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 Data-flow Patterns** address problems of how to organize

high-level aspects of the data-flow 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 data-flow 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 data-flow 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 the implementation of the visual prototyping architecture. Thus, 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 [**?**].

3.8.2 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... 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 [**?**].

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

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.

Solution

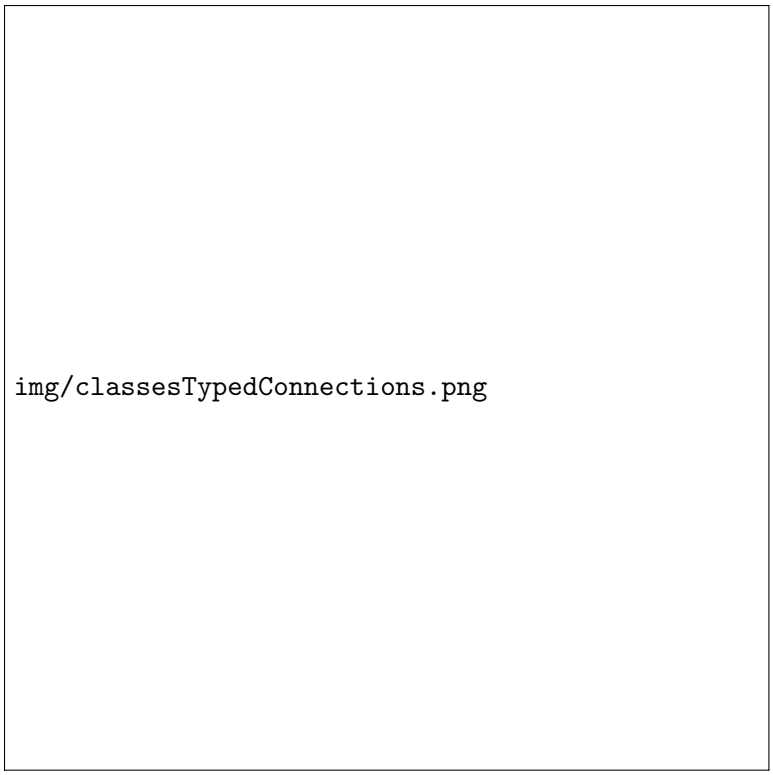

Figure 3.25: Class diagram of a canonical solution of Typed Connections

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. 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 figure **??**.

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.

3.8.3 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.

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.

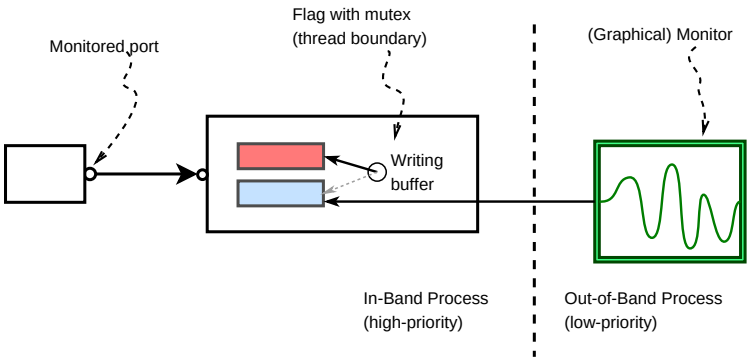

Figure 3.26: A port monitor with its switching two buffers

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.

3.8.4 Patterns as a language

Some of the patterns in the 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 **??**. 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 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 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

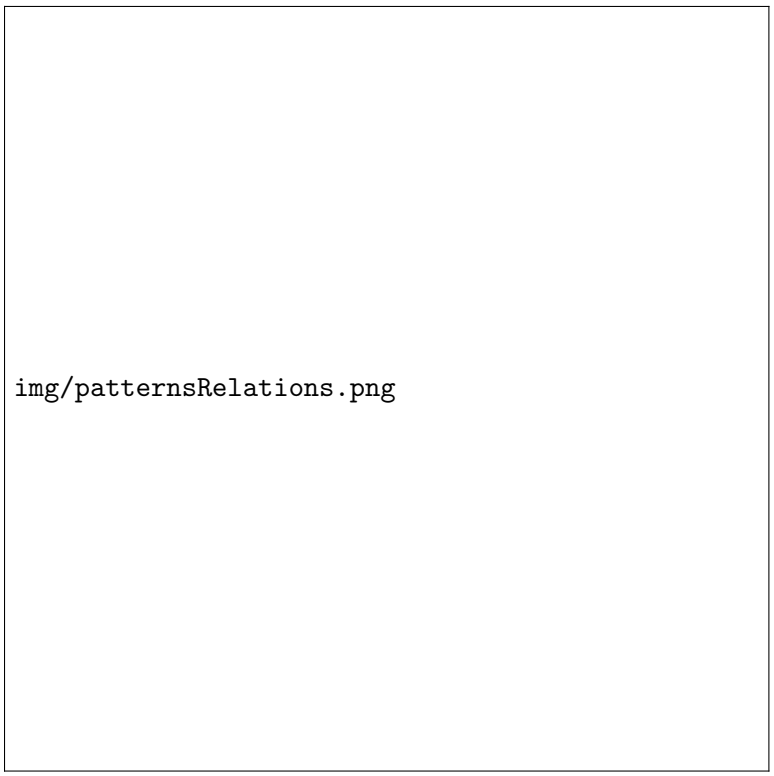

Figure 3.27: The multimedia data-flow pattern language. High-level patterns are on the top and the arrows represent the order in which design problems are being addressed by developers.

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.

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 data-flow patterns" category can be used on any other data-flow domain. Typed Connections, Multiple Window Circular Buffer and Phantom Buffer have applicability beyond data-flow systems. And, regarding the Port Monitor pattern, though its description is coupled with the data-flow 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).

3.9 Summary

This chapter has done a domain analysis of the program family which includes the applications related to audio and music. Some abstractions has been extracted such the ones addressing the system data input and output, user interface interactions, several processing regimes and the implementation issues and solutions related to them. Then we used such abstractions to model several archetypical audio applications, and finally, we also used such abstraction to define the sub-family of applications that are to be visually built by the prototyping architecture. The abstractions also gave some guidelines on how the prototyped components could be used in the set of applications that fall away of the selected sub-family.

Chapter 4

The prototyping architecture

This chapter describes the details of the prototyping architecture. Firstly, an overview describes the relations and functionalities of the different architectonic elements. Then, it explains how each architectonic element address the different issues that are required to fulfill the functionality.

4.1 Requirements

The family of applications the architecture is able to visually build includes real-time audio processing applications as defined in **??**. This include some applications archetypes such as real-time software synthesizers, real-time music analyzers (figure **??**) and audio effects and plugins (figure **??**).

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...), 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 section **??**.

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.

Besides that, the architecture provides the following features:

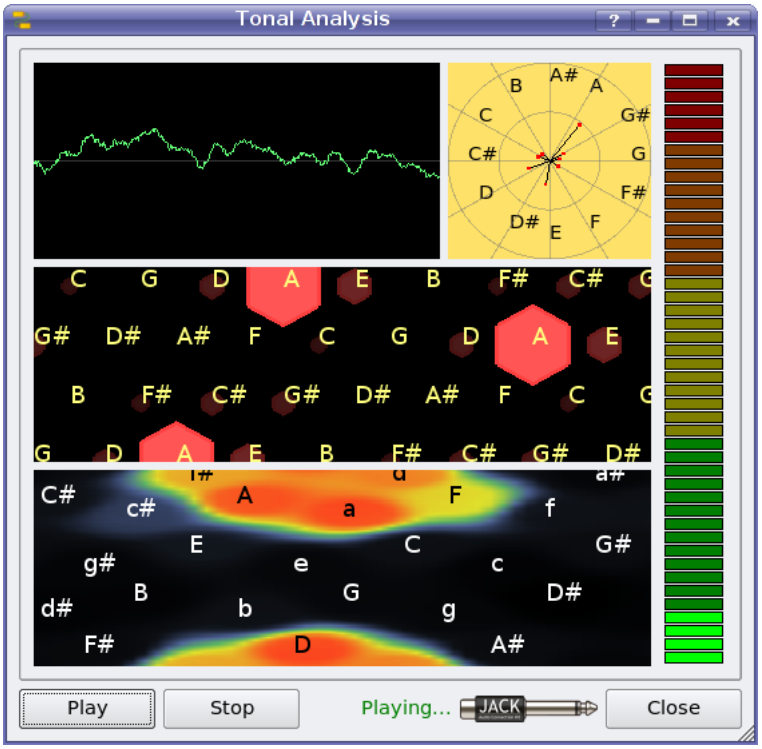

Figure 4.1: An example of audio analysis application: Tonal analysis with chord extraction. This application can be prototyped in CLAM in a matter of minutes and is able to analyze and incoming audio and extract and represent its chords and tonal components.

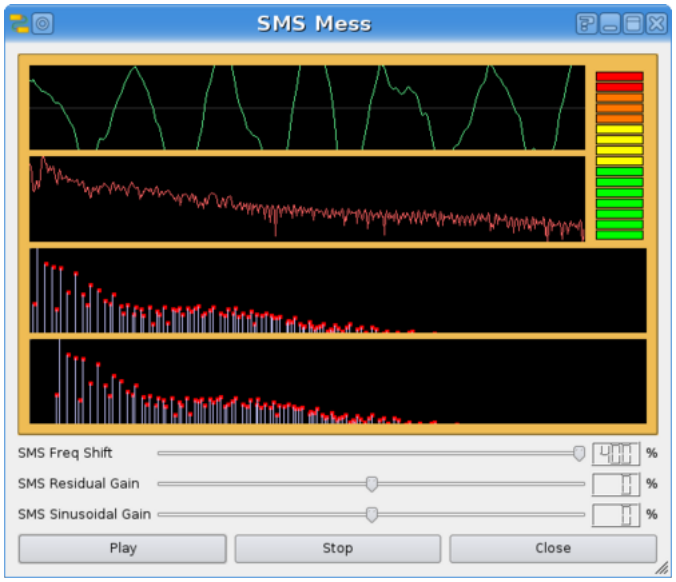

Figure 4.2: An example of a rapid-prototyped audio effect application: Pitch transposition. Note how in this application apart from representing different signal components three sliders control the process interacting directly with the underlying processing engine.

- 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...)

4.2 Main architecture

The proposed architecture (figure **??**) has three main components:

- A visual tool to define the audio processing core,
- a visual tool to define the user interface, and
- a run-time engine that joins both elements into a running application.

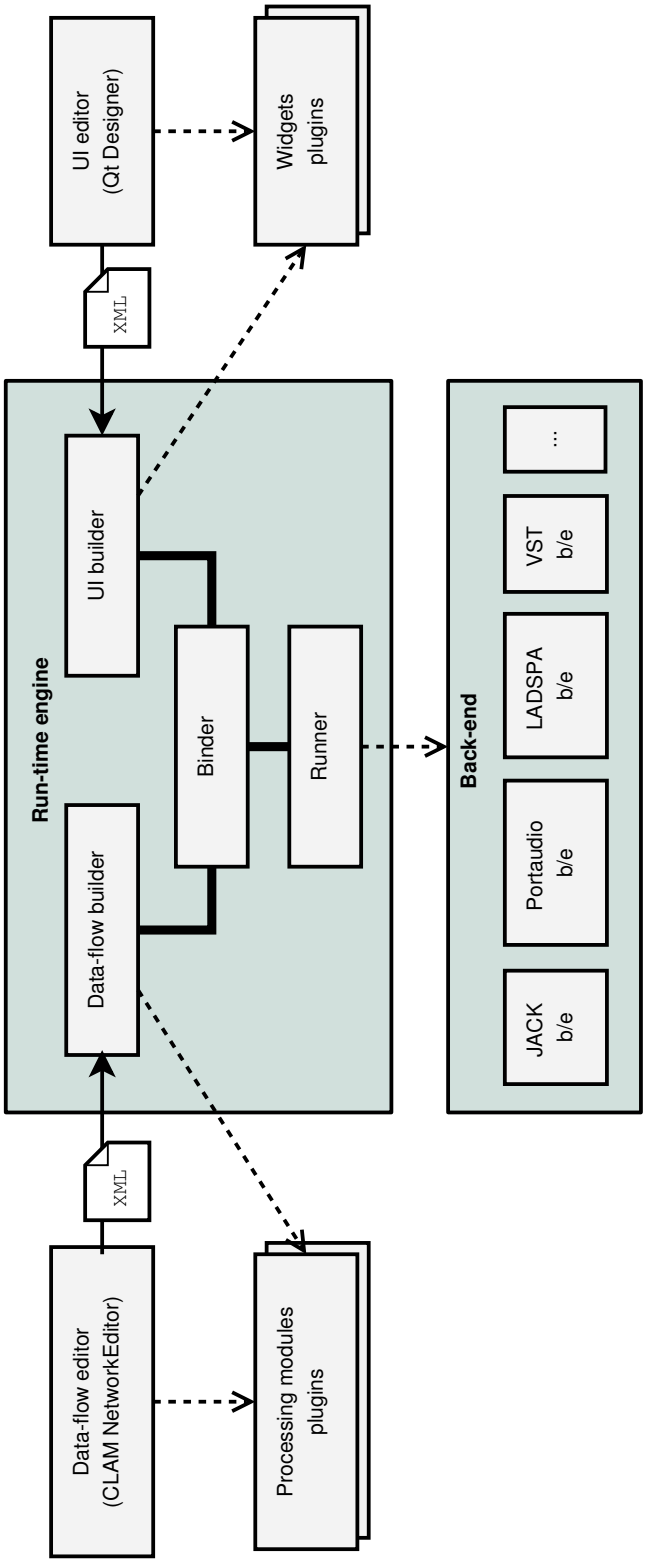

Figure 4.3: Visual prototyping architecture. The CLAM components that enable the user to visually build applications.

The key element is the run-time engine. It 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.

4.3 Visual builders

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 **??** 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.

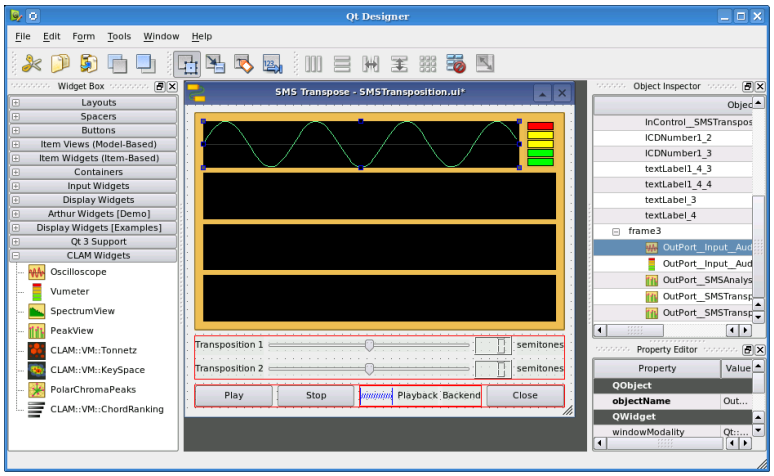

Figure 4.4: Qt Designer tool editing the interface of an audio application.

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

instantiated user interface by using introspection capabilities.

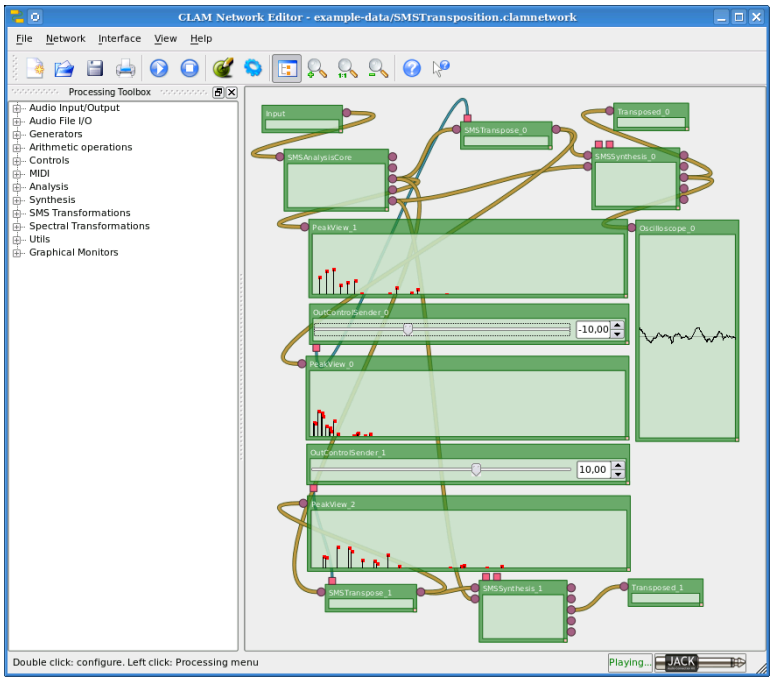

Figure 4.5: NetworkEditor is the visual builder of the CLAM framework. It can be used not only as an interactive multimedia data-flow application but also to build networks that can be run as stand-alone applications embedded in other applications and plugins.

Analogously, CLAM Network Editor (figure **??**) 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.

4.4 Run-time engine

If only a data-flow 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.

4.4.1 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. 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.

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.

4.4.2 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 figure **??**) 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 can not 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 section **??**. In a nutshell, 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. 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.

4.4.3 Thread safe communication in real-time

One of the main issues that typically need extra effort while programming is multi-threading. It is hard to program and to debug. 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 [**?**]. Being in different threads, safe communication is needed, but traditional mechanisms for concurrent access are blocking and the processing thread can not be blocked. Lock-free structures are overkill as conditions are loose: The reading thread (visualization) may block and even lose tokens. Indeed the refresh rate of the screen is orders of magnitude greater than most token rates. Thus, new solutions, as the one proposed by the Port Monitor pattern in section **??**, 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.

4.5 Component extensibility

The architecture provides means to extend its capabilities by adding plug-ins. Such extensions cover different aspects of the architecture: Processing

modules, widgets, extended data types, binders...

The shared mechanism of those plug-in is that they populate a singleton structure such a factory or a meta-data dictionary by creating at global scope object whose constructor does the population. As library plug-ins are loaded by CLAM all those objects are created and their constructor register their load onto the singleton. We use a C++ idiom that ensures that the singleton is created before any use [**?**].

Besides processing components and widgets there are other plug-ins which are worth to explain in detail.

- System Back-ends
- Binding plugins
- Type plugins

4.5.1 Binding Plug-ins

Binding plug-ins are used by the run-time engine to locate special widgets that need to be related somehow with the network. 'Special' means any criteria on type, name, properties or any other aspect accessible with Qt introspection capabilities. Binding plug-ins are the ones that bind port monitors and control senders, but also transport buttons, back-end and playback indicators...

4.5.2 Type Plug-ins

The Typed Connections pattern does not require to register any type. If two ports share the token type they can be connected. Anyway type plug-ins can be defined to increase the services for that type. Currently, such services just include port coloring.

4.5.3 System back-ends

Real-time audio applications may work in a heterogeneous set of contexts: System interfaces to the audio devices such as ALSA, OSS, CoreAudio, WMME, DirectSound or ASIO, wiring API's such as JACK, or plugins systems such as LADSPA, VST, VSTi, DSSI, Audio Units... Audio applications

have to perform a set of tasks that tightly depend on such a context. Managing threads, providing callbacks, exploring devices, or feeding data from and to the application.

The architecture solves that complexity by enabling *back-end plugins*. Back-end plugins encapsulate context dependant tasks into interchangeable objects, so that by selecting a different back-end plugin, the application can be run in a different context. This also has the side effect that enables the extension of the architecture to future execution contexts.

Thus, 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.

Some important aspects of the back-end plugin system should transcend to the user interface to provide a set of functionalities: choosing the back-end among the available ones, choosing the device binding to each source and sink, changing the back-end status (playing, stopped, paused...), and displaying back-end information, such the status, error conditions... The architecture also defines graphical elements that perform such function, which are connected upon binding.

4.6 Reusing components in non-real-time applications

Section **??**, defined what we consider a real-time applications which are the ones that can be modeled with the visual prototyping architecture. This disables the application of the visual prototyping to applications featuring:

- Non-streaming processing
- In-memory representations
- Multiple audio item processing
- Any other application logic that is more complex than binding to external streams, starting, stopping, control the transport on file based streams, sending control events and visualizing instant data.

Anyway, components developed in the present framework can be helpful to develop applications that fall outside of those limits. For example, being the sinks and sources a plug-in system the full application as is, can be used within a more complex host applications, as in the case of an audio authoring tool or a DAW system.

Also, if a regular toolkit is used for the user interface, such interface definition can be used and dynamically extended in a more complex application which introduces more application logic. All the means to bind the user interface and the processing algorithm in a transparent way are also available when programming.

Also, in-memory or file representation could serve as streaming source or sink for an streaming process, including the data source for instant views.

And finally the streaming processing components can be building blocks of more complex processing patterns. Section **??** explained a way of building a non-streaming processing by communicating summary computations among two different streaming processes. And streaming and non-streaming processes could be the core process scheduled for multiple audio item processing.

Examples of some of those adaptations can be found in some use cases explained in chapter **??**.

Chapter 5

Evaluation

This chapter evaluates the proposed work by analysing the actual implementation of several use cases for the prototyping architecture. For each case we do a short description of the problem, a memory of the development process, a summary of the developed processing and graphical components and a discussion on how the architecture performed. Use cases were approached in parallel to the architecture definition, development and refining so, some flaws detected in early use cases get fixed in later use cases.

At the end of the chapter we apply a set of systematic criteria to qualitatively evaluate the architecture and analyse the current limitations.

5.1 Use case: SMS processing

We used this first use case to implement the basics of the prototyping architecture elements. As both developments were very interleaved it can not be used to evaluate the efficiency of the process, just to explore its capabilities and evaluate the quality of the final product.

Spectral modeling, often referred as Spectral Modeling Synthesis or SMS[**?**], is a technique that gets an alternative representation of the sound which splits the signals into a set of sinusoids and an stochastic residual. SMS representations enables high level manipulations of the sound by manipulating both components independently[**?**] [**?**].

The key elements of the spectral modeling are the analysis and the synthesis. The analysis (figure **??**) is the process of obtaining the sinusoidal and the spectral components given an audio stream. The synthesis (figure **??**)

is the inverse process, reconstructing the original signal given the spectral components. Analysis and synthesis are streaming processes; each audio frame corresponds to a pair of spectral peaks and a residual spectrum.

Transformation processes can be performed in the middle. Most of the transformations are also streaming processes. But, for instance, SMS time stretch effect is a example of transformation that can not be executed in real-time conditions, because input and output streams are not synchronous in time. Anyway, the architecture could enable its implementation if the source of the target stream does not impose real-time constraints, such an audio file or a in-memory representation.

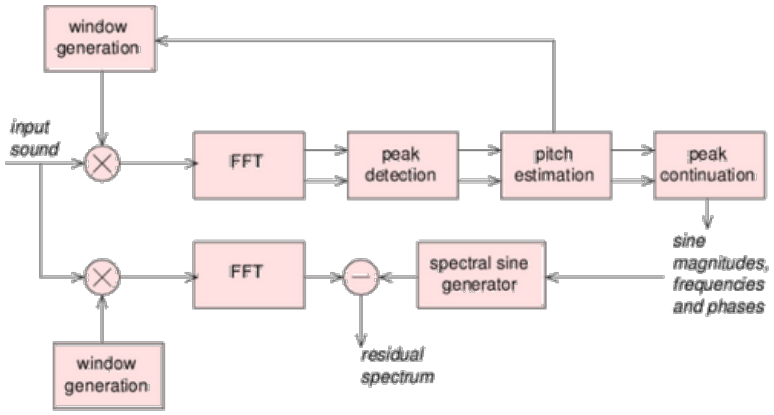

Figure 5.1: SMS Analysis block diagram.

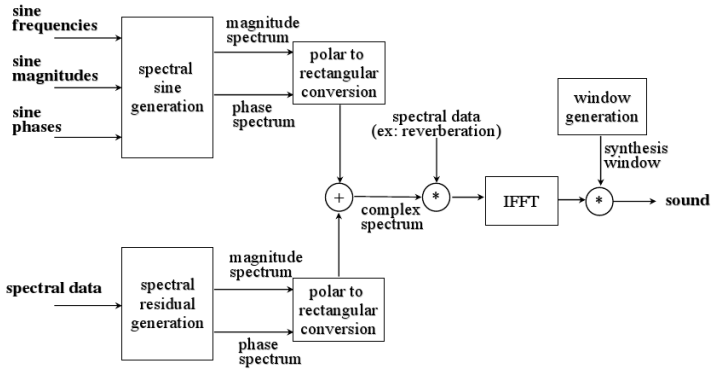

Figure 5.2: SMS Synthesis block diagram.

5.1.1 Development memory

At the time this experiment started, the CLAM framework already contained code for SMS technology. Indeed most of the processing modules in the CLAM were related to spectral modeling. But they were implemented in the context of an authoring tool, CLAM SMSTools, which did all the processing in off-line, and the implementation was not ready for stream processing. Existing modules at the time were tightly coupled to an in-memory representation of the full audio item, where the results of the analysis and successive transformations were dropped. So in order to move them to the prototyping platform we had to redesign them as streaming processes.

Removing references to the in-memory representation was a very hard process because *hidden dependencies* on the code existed and we often broke the code. We finally decided to revert all the changes and repeat the port with exhaustive back to back tests.

Communicated data object were audio buffers, spectra, spectral peaks and fundamental frequency candidates. In order to visualize them we developed a first version of the Port Monitor solution. After most of the basic elements were set up (analysis, synthesis and views) we started to build up applications. Mostly effects for instance:

- SMS Pitch Shift effect, already shown at figure **??**, which changes the frequency of the sinusoids.
- Gender Change effect, figure **??**, which turns male voices into female voices and female voices to male voices.

5.1.2 Interface components

As explained above, three new views based on port monitors had to be developed: The Oscilloscope, to view audio streams, the Spectrum View to see the residual component and other spectra, and the Spectral Peaks View to see the sinusoidal component.

Applications also required widgets to send control events of different kinds: Bounded floating point, boolean and enumerations. We reused the widgets provided by the Qt toolkit and implemented control sender processing modules to bridge the threading gap.

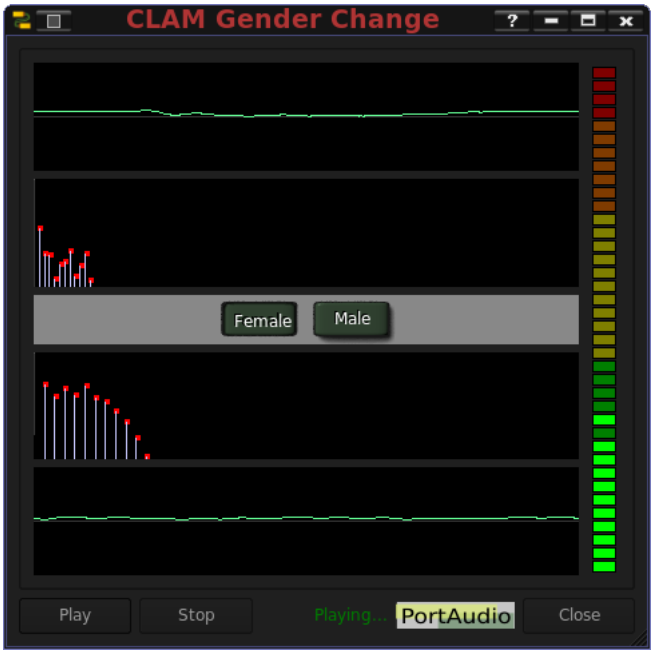

Figure 5.3: The GenderChange application based on SMS technology working on real-time.

5.1.3 Discussion

By forcing modules to work with ports in streaming, we decoupled the transformation modules from the handling of the in-memory structure. All the code to access on the structure a given data on a given instant of time was thrown away. By hiding the time handling details, the code of the transformation modules became more compact and understandable (a mean of 40 percent decrease in SLOC).

All the hidden dependencies were turned into module ports and control which made dependencies explicit both for the user at the network level while visually prototyping, and for the module maintainer on the C++ code.

So, migrating to the new architecture forced to refactor the code toward a good design: separation of concerns, definition of well defined interfaces... And we consider this a sign of having a good *Path of Lowest Resistance*.

5.2 Use case: Multi-faceted real-time Analysis

This small project started as some CLAM user asked on the mailing list for supporting MFCC [**?**] and LPC analysis on the CLAM Music Annotator. The CLAM Music Annotator is an authoring tool for doing off-line analysis and visualizing and editing the results.

Although user requested them for the Annotator, such analysis are able to be implemented as streaming and testing and assembling could be more easy visually so we first implemented them as a real-time analysis and then reused the components on the Annotator.

This use case is simple but interesting because introduced the need of reusing view components for different port data types.

5.2.1 Development memory

Again the components were already in CLAM but not ported to be usable from the Network Editor yet. Also there was a pair of programs that did the assembly of objects and the extraction for each analysis. So we ported the modules and assembled them within the Network Editor.

Also the visualizations were missing. Most of the data to visualize were float arrays, to be visualized as line or bar graphs. But the actual data on the port was not float array but more abstract types such as LP Models which contains two arrays, the filter coefficients and the reflection coefficients. In this case for example, we had to create two different Port Monitors getting the same type of port but offering different data with the same interface.

Then the widget that displays the Graph Bar could just take the array wherever it comes from, by using this uniform interface which also provided meta-data such as the labels or the numeric intervals to compute them, the units, the bounds of the function...

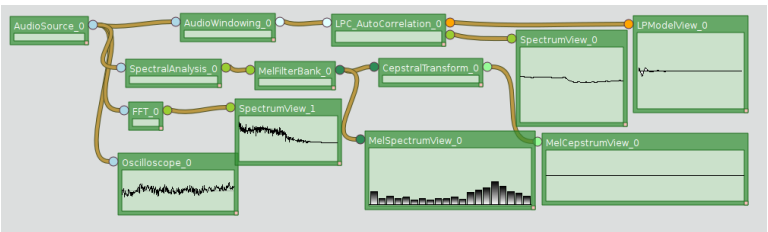

Figure 5.4: Processing components of the multifaceted real-time analysis.

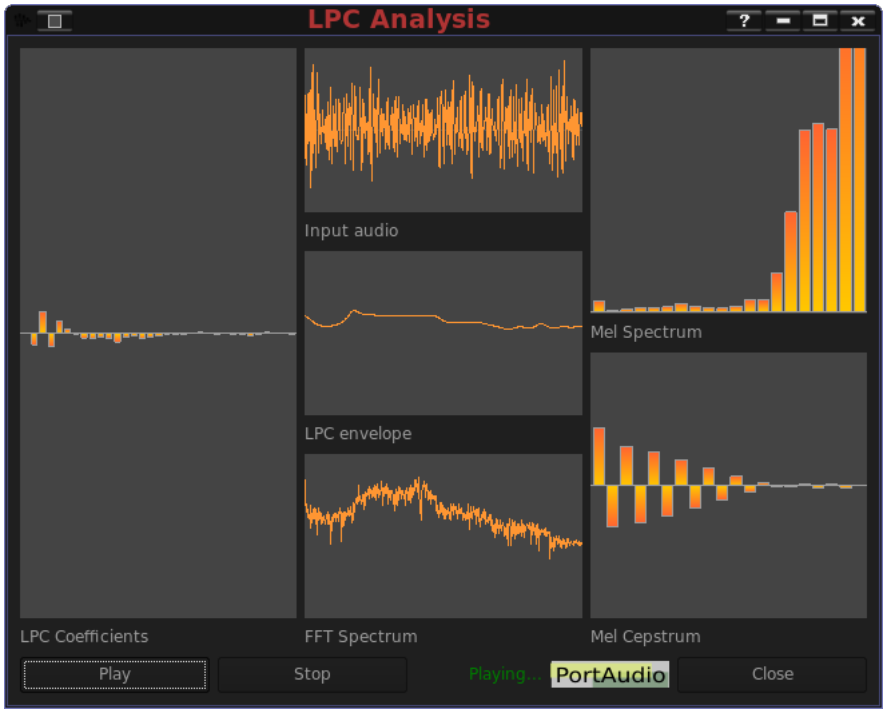

Figure 5.5: A multi faceted real-time analysis showing different views of the incoming sound: LPC coefficients, Oscilloscope, LPC envelope, FFT spectrum, Mel Spectrum and Mel Cepstrum.

In order to reuse the Bar Graph view on the Annotator we provided an implementation of the same interface but instead of taking the data from a port monitor, it retrieves the data from the in-memory representation of the off-line computed data.

5.3 Use case: Chord Extraction

This experience happened while the prototyping architecture implementation and concepts were usable but still incomplete and it provided useful insights of the requirements and the applicability. It is also useful to see the process, costs and benefits of porting a research Matlab code to the prototyping architecture. The developer was myself, so the results are not valid to analyze other factors such as framework usability.

The chord extraction in CLAM is an implementation of the algorithm described by Harte in [**?**] and it is based on Harte's existing Matlab im-

plementation. We modularized and ported the original Matlab implementation with a very close support of Harte and other colleges from his lab, we did some drastic optimizations and real-time adaptations, we created some instant-views for chord analysis, and adapted all to the prototyping architecture. An screen shot of the final application is shown at figure **??**.

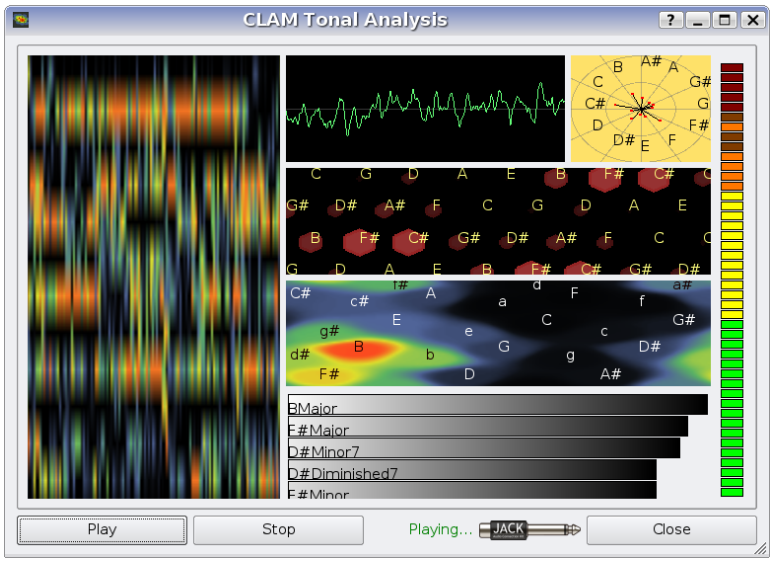

Figure 5.6: The 'Tonal Analysis' prototyped application.

5.3.1 Development memory

The code port followed a tight methodical process this time. Firstly I set up a battery of back to back tests to the full algorithm. Such back to back tests were useful to do refactoring on the original Matlab code. I also placed some back to back test on intermediate points of the algorithm. Once the original code was propped up, I started to refactor the Matlab code to extract little modules with strong cohesion. Every time a new module was identified, white box unit tests were created. Such unit test were used as a guide for test driven developing the module in C++. As more C++ modules were available, intermediate back to back references were used as input or output to back to back testing the aggregation of C++ modules.

Modularity made it also easier to profile the full algorithm to detect modules which were critical in performance. After doing some enhancements the code was optimized from lasting 30 times of the excerpt duration to last

a third of the excerpt duration. That allowed us to increase the temporal resolution of the algorithm while still being real-time, and it also allowed us to be less conservative with the frequency of execution of the tests.

Although the algorithm was optimized enough to be executed on real-time, one of the components was non-streaming and the execution had to be off-line. So the first integrated version of the algorithm, instead of being integrated into the prototyping architecture was integrated on the CLAM Annotator which allowed off-line execution.

Two instant views were added to the Annotator to visualize some of the algorithm outputs: the Key Space and the Tonnetz which I explain below. Having some views that had to be port monitors on the future spotted the need of setting an abstract interface so that the view could access transparently real-time computed data through a port monitor, or an in-memory non volatile representation such the one that used the Annotator.

Later on, I substituted the module that forced off-line execution, which was related to the tuning detection, and all the modules could be managed by the Network Editor. Back to back tests are no more useful when you are trying to enhance the algorithm. So, in order to check future improvements, a precision/recall test was set on the Beatles hand annotated collection.

With the new real-time incarnation of the algorithm, further improvements were tried. For example, considering the *none* chord with the same weight for all the pitches, helped to discard tagging as chord segments of the song were there was no real tonal. The network editor made easier to create and use alternative components to the existing ones. Also to fine tune parameters as they are part of the configuration. Moreover, new views of the intermediate data, such as the Polar Chroma Peaks and the Chord Ranking views, gave more insight about what was happening on the algorithm.

5.3.2 Processing components

Most of the components for the Tonal Analysis prototype had to be reimplemented from scratch taking the Matlab implementation as reference. The criteria to define the modules boundaries has been finding proper algorithm points where a representation change was produced or where a given representation suffered some clear operation. The modules we defined are the following ones:

Fourier Transform: Original code used the intrinsic FFT operation in

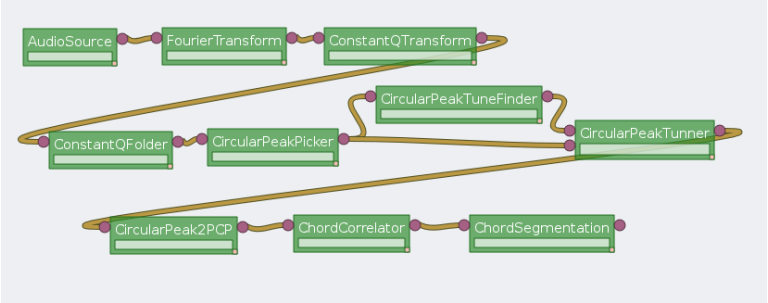

Figure 5.7: Processing modules of the tonal analysis

Matlab. For the C++ version, we used the 'Fastest Fourier of the West' [**?**] implementation which offers a great speed up.

Constant Q Transform: Takes a spectrum and multiplies it by a matrix to get a spectral representation where the distance among bins is proportional to the bin frequency. The output has the same number of bins to represent each semitone.

Constant Q Folder: Takes a constant Q transform and folds it into an octave so that the bins for the same pitch note on different octaves get added. That representation is called the chromagram.

Circular peaks finder: Takes the chromagram and finds the quadratic interpolated peaks. It is circular because it considers the first and last bins as adjacent bins.

Circular peaks tune finder: Folds the peaks into a semitone and add them considering them as phasors where the cents within the semitone sets the angle and the peak magnitude sets the module. The vector sum gives the instant tunning and the tonal level of the frame. A parameter controls the inertia, that is how much of the past tunning results affects the current tuning to be used.

Circular peaks tuner: Takes the circular peaks and the detuning and corrects them to be centered on the standard center.

Circular peaks to pitch profile: Adds all the peak magnitudes within a pitch into a pitch profile. A parameter let's choose whether the peaks are weighted the same or they are weighted depending on their distance to the pitch center.

Chord correlation: Correlates the pitch profile with a database of chord models and outputs the correlation for each chord model.

Chord segmentation: Takes the chord correlation and decides which is the current chord. Several parameters play there such as the minimum chord length, the minimum factor between the None chord and the first chord correlation in order the first chord to be considered...

5.3.3 Interface components

In this experiment we explored the addition of views that could be useful either to the development of the algorithm or to a final user. Often the results of the algorithm are not resolutive. In such cases we discovered that a proper visualization was more informative than the output of the segmentation algorithm.

Most of those views were also reused on the Annotator, by providing front-end to the in-memory representation that implemented the same data source interface than the Port Monitor, thus demonstrating that real-time user interface is also useful to authoring tools environments.

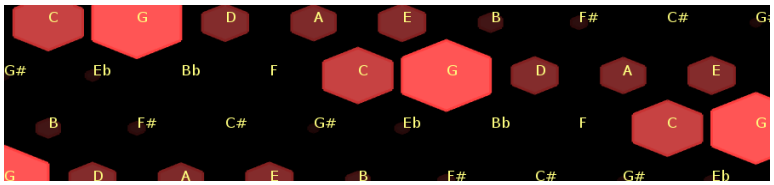

Figure 5.8: The Tonnetz view showing a C major chord.

The Tonnetz view (figure **??**) displays the intensity of each pitch note in musical meaningful arrangement inspired in Riemann[1] network representation of harmonic relations [**?**]. Notes are displayed as hexagonal tiles so that notes with harmonic relations are side by side. For instance, horizontal adjacent tiles form a series of fifth intervals, top-right adjacency means a major third interval while bottom-right adjacency means a minor third interval. Such distribution is interesting because common chord modes have a distinguishable shape (figure **??**).

The Key Space view [**?**] displays the probabilities of major and minor chords to be the sounding chord. It uses the chord correlation output. Chords are also placed so that chords with common pitches are placed together so that normally is seeing a colored stain on similar chords with a

[1] A 19th century music theorist

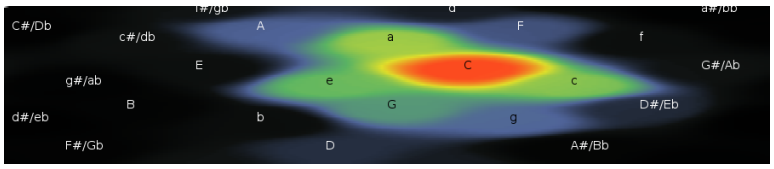

Figure 5.9: The Key Space view showing a C major chord. Minor chords labels are lower case.

prominent color spot on the center highlighting the most probable cord. See the figure **??**.

Figure 5.10: The Chord Ranking view.

The chord ranking view (figure **??**) displays as horizontal bars the correlation of the top most probable chords in a frame. It allows see other chords than major and minor shown in the Key Space.

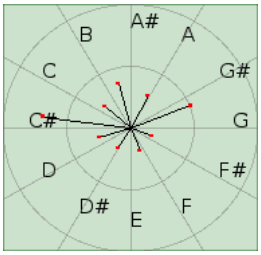

Figure 5.11: The Polar Chroma Peaks view.

The Polar Chroma Peaks view (figure **??**) represents the peaks detected on the chromagram displayed as fasors in the octave circle.

5.3.4 Discussion

This first experience gave us some insight on the new development framework.

First, porting has asserted to be a source of a lot of translation bugs. The testing propping up we set up detected and located a lot of them that

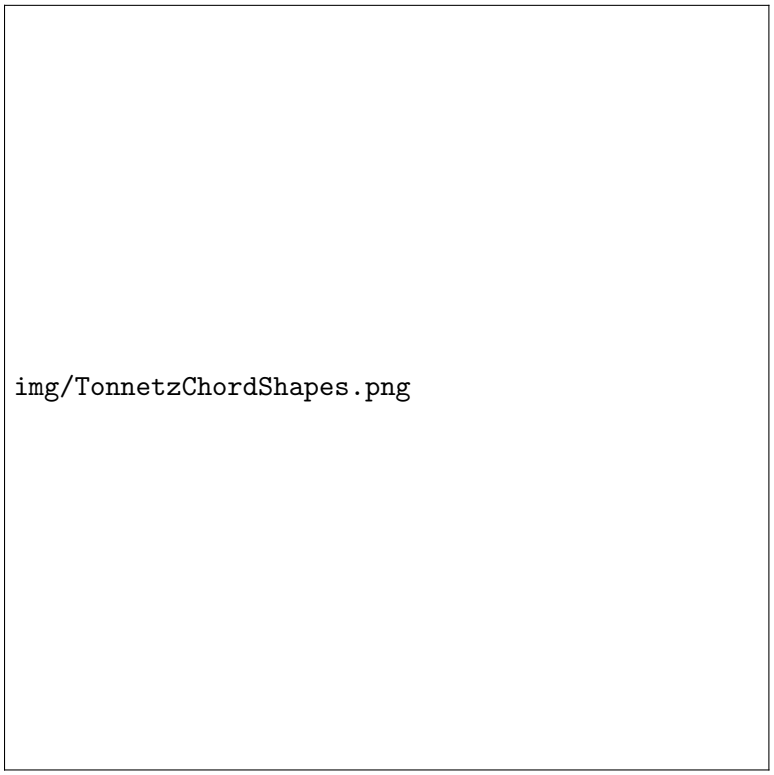

Figure 5.12: Different chord modes are seen as different shapes in the Tonnetz view

otherwise could not be detected until the final system was build, and at that moment, location would be very hard, and even more with the cross effect of different bugs acting at the same time. So, the porting method we used also proved to be useful.

Next, modularization helped in several aspects besides the testing. It helped to locate the performance edges by using profiling. It also helped to experiment with new alternative modules without throwing away the old implementation just by plug-in one or other module. Configuration

It was also proven that a network could be executed both in off-line and real-time modes. Also that instant views could be reused with minor effort in an authoring tool such the CLAM Music Annotator.

On the other hand the final application, although it was innovative and it impressed a lot of potential users, it had some usability concerns. We built a nice technology demonstrator but we failed to consider some aspects of its usage context. Such application was meant to be used for novice instrument players that wanted to know the chords of songs of their own personal collection. Instant views provide very volatile information so the user had to play back again the same excerpt once and again. The only way of controlling that was by accessing to the multimedia player that was piping audio to the analyzer. The application context switch was very inconvenient, as the user hands were often busy with the instrument.

That fact drove us to do a different version of the application that gets the audio from a file and controls the transport. This also solved another problem: the algorithm introduced very much latency into the output. By taking the input from a file, the analysis latency can be compensated by a delay.

Of course, because the components were shared, as the streaming version was still useful for other purposes, we still could keep both of them.

5.4 Use case: 3D room acoustics simulator

This is one of the first projects built once the prototyping architecture was mostly stable.

Some months ago, two contributors to the CLAM project, Pau Arumí and myself, were hired to implement some existing acoustics technologies. A new system to simulate the acoustics of sound emitter and receptors moving

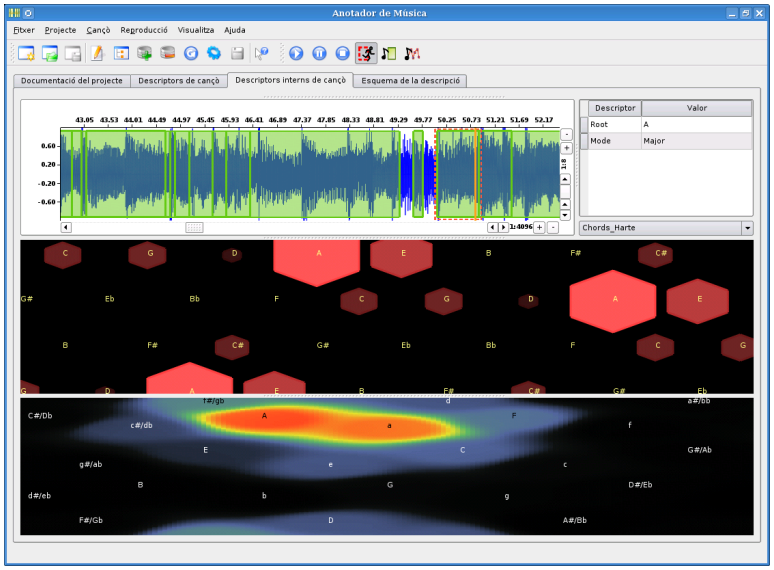

Figure 5.13: Integration of the chord extractor into the Music Annotator

within a virtual 3D scenario. After taking a look at the requirements we decided to implement most of the modules from scratch, and assemble them using the prototyping environment.

For this project we took two decisions:

- to reimplement basic spectral functionalities existing in CLAM to make them less complex, and
- to build up the components using the prototyping architecture.

5.4.1 Development memory

The system development was split in several phases:

- Solving the surround system (5.1 speaker components from pressure and velocity vector at the hot spot)
- Solving the real-time convolution
- Setting up a hyper-grid database of impulse response (an IR for each pair of 3D points; emitter and receiver)
- Solving the interpolation of the impulse response on the actual positions, taking the nearest points on the grid.

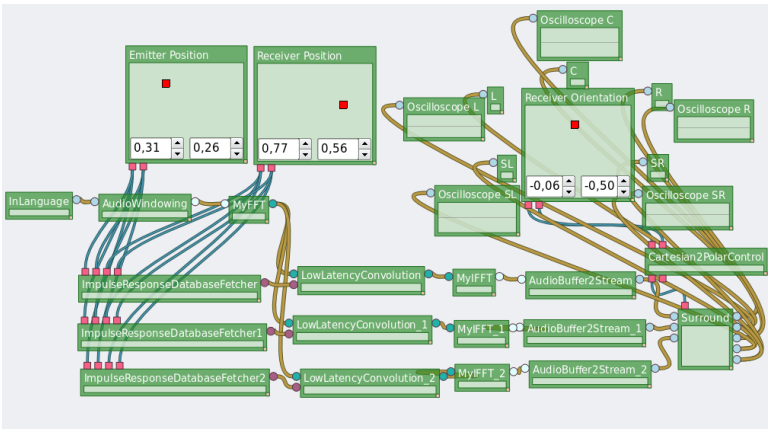

Figure 5.14: The virtual spacialization network.

Because each step is independent, each step could be addressed independently. This is a sign that the system eases *progressive evolution*.

5.4.2 Processing Components

Here we describe some of the components that composes the system:

Low Latency Convolution: This component does a convolution in an hybrid temporal-spectral domain. Hybrid temporal-spectral domain convolution means that it is a similar implementation than time domain convolution but multiplying and adding spectra from the discrete Fourier transforms (DFT) instead of multiplying samples. See figure **??**. By doing this, we get a compromise between the speed of spectral domain convolution and the low latency of time domain for long impulse responses. The latency of the system is the duration of a single frame.

Impulse Response Database Fetcher: Given a pair of emitter and receiver points generates an interpolated impulse response.

5.4.3 Interface components

In order to control the positions of the emitter and the receiver on the scene we developed a widget that sends two controls depending on a point within a surface. The same control also served to control the head orientation of the receiver.

This control served as a first use case of a widget having to be connected

Figure 5.15: Diagram of the low latency convolution processing component.

to two different connection points. This forced us to use a different property than the name to do the binding.

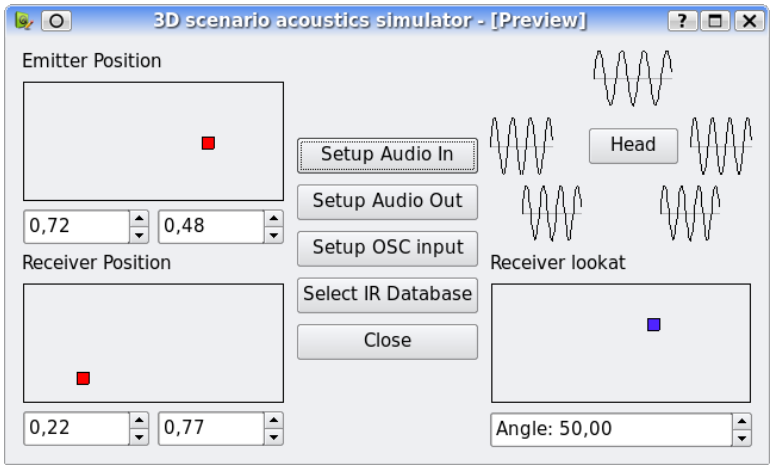

Figure 5.16: The interface for the 3D scenario acoustics simulation

5.4.4 Discussion

We built each of those components and test them independently, and assembling them to have the final system.

When reimplementing the spectral processing building blocks, having the basic implemented in such a modular way was very useful to experiment on which is the most efficient way of doing operations in the sense of which representation is harder to obtain and to operate on.

5.5 Overall discussion

This section summarizes the insights obtained from the previous experiments. To do so, we follow the systematic criteria enumerated in **??** and ??.

Ceiling and Threshold

The architecture lowers the learning threshold, by no requiring programming skills and hiding hard audio related implementation details to the designer. Still the architecture provides a relatively high complexity ceiling to the set

of applications that can be built just visually. But it also offers mechanism to raise such ceiling by extending existing components or by programming a more complex application logic.

Path of least resistance

A tool based on such architecture offers a *path of least resistance* which leads to good design decisions, for example, separating processing and interface in different threads, setting controlled real-time and thread-safe means for communicating them, modularizing processing and the visual elements, and reusing components among designs.

Predictability

The execution of the network is deterministic and predictable. Although the scheduling algorithm could take different orders of module firings because of dependencies, this should not affect the result.

Moving Target

Moving target criteria is something that could happen in the future but can not be predicted.

Closeness of mapping

The application to be built has been split in two domains. Each domain has been addressed by a visual domain-specific language which is close to the abstractions we use.

In the case of the processing domain the visual language we use the data-flow paradigm which is very close to the modular design often used in signal processing.

Moreover, most of the modules and the shared data are high level, so we are not dealing with low level programming constructs such as LabView, ProGraph or PD, encouraging the users to encapsulate that level of programming within a C++ programmed module.

On the interface domain, we are dealing with widgets, layouts and properties which are also entities, interface designers are used to.

Viscosity

Conceptually the data-flow interface has the same flaws that other visual languages in the sense that changes mean to lay out again the components. A lot of time is wasted in that kind of task. This is something that the Network Editor tool could solve by providing means of automatic layout.

Anyway, producing small changes is not hard conceptually. SMS example gave a lot of examples where going from one application to another was a matter of switching some processing modules.

Also new usability features has been added to leverage tedious tasks such as providing access to the port monitors that can be connected to a port in the contextual menu of the port, the ability of adding controls senders with a double click on the receiving control...

Hidden dependencies

As shown on the SMS use case, the 4MPS abstraction makes data dependencies explicit. This is good, but we also found on the system a very nasty hidden dependency: Whenever you change the name of a port or the name of a processing, the connections with the interface can not established. This is something to solve in the future.

In the case of the ports, which depends on the module implementation, this issue could be solved by providing port alias after a rename. Those port alias could be stored within the factory meta-data. Also a migration tool could be helpful. That tool could update such names, which corresponds to a migration alias, whenever an old network is loaded.

Renaming the modules is something that could be handled by synchronizing both prototyping tools.

Hard mental operations

Because the high level of the processing operations, hard mental operations are hardly found.

Secondary notation

Naming and layout are the currently available secondary notations on the processing visual building. They are not hard to master but they are hard

to keep. The layout because it is hard to keep when changes are introduced. The naming because the hidden dependency previously explained.

It would be interesting adding documentation facilities such the ones explained in [**?**].

Consistency

We have not detected any consistency flaw but in the naming of the actual names of the ports and module types. I mean, ports does not follow the same naming convention so it is hard to remember the port names. Of course, this is not a flaw of the architecture neither on the architecture implementation but on the module implementation, although the implementation could enforce some kind of convention.

Progressive evolution

Along the examples, we have used progressive evaluation as the methodology to get the applications up and running. Modularization enables white box and black box testing.

Chapter 6

Conclusions

6.1 Introduction

This thesis addressed the problem of reducing the time to market and costs of audio software development by defining a software architecture that enables the construction of complete and rich audio applications by visual means.

Existing platforms provide visual prototyping for the processing aspect using data flow visual languages. When constructing final user products either the processing prototype is embedded into a fully programmed application logic, which implies a lot of development, or the visual prototyping tool itself is used as final user interface, which clutters the user interface with useless, or even inconvenient, functionalities.

Visual user interface builder tools cover visual specification of the interface layout and a minimal reactivity of the interface of an applications. Even by using such tools, the developer must still provide low level programming to connect the user interface to the processing core.

The proposed architecture reuses the concepts from both, visual user interface builders and data flow visual languages for signal processing. Both concepts are broadly used but are hard to be combined to build full applications. The architecture enables the designer to visually bind both sides, enabling her to ignore most implementation details such as communication among real-time threads and platform dependant audio subsystem programming.

Thanks to the architecture a designer could build an application by performing the following steps:

1. Designing a processing network by dropping processing elements into a canvas and connecting their ports to express data and control communication

2. Designing an interface by dropping widgets into an interface and setting up the layout

3. Setting up a binding between visualization widgets and processing data ports and between control widgets and processing core input controls

4. Launch it with the corresponding audio back-end

To evaluate the architecture, an instance of such architecture was built using the CLAM framework and the Qt toolkit. The Qt toolkit was extended by adding new audio related widgets that can be reused for audio data visualization and audio systems control. Such widgets were made available as components for visual composition of interfaces using the Qt toolkit prototyping tool. A visual audio prototyping tool was constructed on the top of the CLAM framework so that it integrates well on the prototyping architecture, and a run-time engine was implemented to join both worlds and providing a minimal application logic.

The development process of existing open source audio projects have been compared with the development of similar functionality by using tools based on such architecture, ascertaining that the architecture contributes to shorten development times and increases the software quality by enabling more iterations over the design.

The benefits of using such architecture has been argued according different criteria that has been proved valid for historical software development tools, such as *learning threshold*, *complexity ceiling*, *path of least resistance*... concluding the following: The architecture lowers the learning threshold, by no requiring programming skills and hiding hard audio related implementation details to the designer. Still the architecture provides a relatively high complexity ceiling to the set of applications that can be built just visually. But it also offers mechanism to raise such ceiling by extending existing components or by programming a more complex application logic.

It has also been shown that a tool based on such architecture offers a *path of least resistance* which leads to good design decisions, for example, separating processing and interface in different threads, setting controlled

real-time and thread-safe means for communicating them, modularizing processing and the visual elements, and reusing components among designs.

6.2 Summary of contributions

This thesis makes significant contributions to the state of the art in audio software engineering. Such contributions are summarized in this section.

A way of systematically analyse the requirements of an audio application. Existing literature on audio software engineering addresses concrete aspects of audio application development. This thesis provides, in chapter **??**, systematic and generic criteria to determine when such issues must be addressed. We have analyzed several facets of audio applications such as data sources and sinks, data-time dependencies, processing modes, and user interface interactions. Such elements conforms the application logic and each one is bound to a set of low level issues.

An architecture that transparently solves low-level issues related to real-time audio applications. The architecture presented in chapter **??** solves transparently most of the issues which are related to the application logic of real-time application. For instance, multi-threading, lock-free thread safe communication with the user interface, system context handling, buffered file access...

An architecture that enables visual building of real-time applications. The architecture presented in chapter **??** also enables the visual building by reusing existing technologies, such as data-flow languages and user interface visual builders, by providing means to dynamically join their outputs: relating them in definition time and bind them in run-time. Application logic can be specified at high level by relating entities of the interface with the ones at the processing core.

Means of reusing visually built components as components in other stereotypes of audio applications. Although visual building is limited to real-time audio applications, section **??** provides some hints on how components visually built can be reused as is on other stereotypes of applications.

Component extensibility. This thesis provides means to extend the available components of a visual prototyping architecture in a very flexible way. The architecture allows (section **??**) to define protocols among interface

and processing components without fixing such protocols on compile time but still allowing protocol checking. The architecture ceiling can be extended in many fronts.

An analysis on how the architecture performs when developing real use cases. The implementation of the architecture and its use to implement several use cases has provided some insights on how the tool performs and how it could be improved.

As result of the work in this thesis, a number of publications have been published in conferences and journals which are listed in appendix **??**. Also the work presented in this thesis has been awarded at the ACM Multimedia Conference 2007 Open Source Competition.

6.3 Limitations

6.3.1 Target applications and processing models

The set of target applications for this first approach of visual prototyping, was intentionally limited to real-time processing applications. That is, applications which process streams of audio related data. They are often referred as audio analysis, synthesis and transformation tools, depending on how many audio input and outputs they take. For instance, a chord extraction application which takes an audio and shows the chords while playing, is an analysis tool. An application which takes an input human voice and changes the speaker gender, is an instance of transformation tool. Also, an application that takes a point on the vowel triangle and makes the corresponding vowel sound is an instance of synthesis tool.

They all share a common trait: data is synchronously flowing and processing elements are taking the decisions based on the past and present data. Some kind of processing is outside of this description. Some algorithms perform what we call 'off-line processing'. This kind of processing is commonly use in Music Information Retrieval when, taking all the data extracted from an audio, the algorithm do an overall reasoning.

Also application with an extended application logic fall out of the scope of the solution provided by the architecture. The architecture just gives the user interaction means to view processing data, to send controls to the processing core and to control the audio sources and sinks. This is insufficient for a large number of audio applications. For instance, audio authoring tools

introduce the concept of timeline and are audio object centered instead processing centered. Audio authoring tools are thought to work with several audio objects and different processing pipelines can be applied. Portions of the architecture could be used but they are not enough to fully build the application.

6.3.2 The ceiling of visual means

We have seen that the described architecture has some clear limits on how much can be done with it, just by visual means. Visual means are not as rich as programming languages but they are easier to learn and use.

Non visual programming is an acceptable learning threshold if we consider that current audio tool development is mostly done by programming. Furthermore, the architecture also lowers the costs of the programming task in the cases when it is needed. So, the ceiling of visual means seems to be something that is not that critical because users are used to work in worse scenarios and the architecture supposes a better one even it could be enhanced. But, visual design introduces a new segment of users which are not skilled on programming. Programming might represent an abrupt slope increase on their learning curve. That's why the visual means ceiling becomes important here.

Moreover, programming is not just a learning threshold, it is also a workload threshold. Programming involves a work context change, and dealing with tedious and time consuming tasks such as deploying a build environment, facing compiling errors, or debugging runtime errors.

There are several aspects where the visual means have a clear ceiling:

- extending the application work flow,
- extending processing or visual components, and
- the visual complexity of wiring networks.

As explained before, the set of target applications has been limited to real-time processing applications. But even when deploying a real-time processing application, designer might want to allow more interaction than just controlling the audio sinks and sources, visualizing data and sending controls. For instance, in a synthesiser application, we could want to choose the instrument we are using, choosing presets, configuring the application,

showing the help... Also, the user interface design limits the visual design options to a single window. As soon as the target requires more application logic, the architecture can not solve that visually and programming is required.

Another ceiling for visual means, is the extensibility of the architecture by providing new interface and processing components. Such extensibility is offered to raise the ceiling of what the architecture is able to build. But again, although the architecture easies the process, programming is still required.

The later ceiling are the human limits of dealing complex wired networks. When a network gets large, visual design becomes harder. Connections are more difficult to trace and display limitations do not allow the designer to fit a large network in the canvas to analyze it. This can be solved by aggregating the processing elements into a single higher level one. Again, with the current proposal, this can be done just by programming.

6.4 Further research

6.4.1 New application stereotypes

As this thesis has been a first approach to address the visual building of audio applications, the scope was limited to simple use cases. Further research might address other stereotypes of audio applications which may have enough homogeneity to be constructed mostly visually. Chapter **??** gave insights on further generalizations and in section **??**, we suggested new ways of reusing the components the architecture to other kind of processing modules. We have seen two of those stereotypes that could be addressed due to their homogeneity but that went out of the scope of this thesis: audio authoring tools and music information retrieval systems.

6.4.2 Raising the visual prototyping ceiling

Still there is enough room to raise the visual prototyping ceiling. Further research could address new ways of extending the set of components available that minimize the learning threshold of programming. Several proposals might be explored. One could be integrating the component programming cycle into the audio processing prototyping tool. This way we are saving the

designer a lot of work load including a work context change, the deployment of the development environment, and the cycle of compiling and installing before testing. The other proposal could be using scripting languages to develop new components. Scripting languages are known to be easy to learn, and because the lack of compiling cycle users feel more comfortable with them. Both integrated development environment into the processing prototyping tool, and the scripting approach could be combined.

There is also a lot of ways that the processing prototyping tool usability could be enhanced. For example, providing debugging facilities such as step to step execution. Using a console with completion could add speed to the definition of the network as now dragging interface requires mouse precision. Also interfaces to edit the structure of a new processing module with actions such as 'adding ports', adding controls', 'add configuration parameter', 'edit the code', and 'recompile' could help to close the development cycle.

6.4.3 Reducing processing design complexity

Another way to enable the user to provide new components is by composition: aggregating several processing elements into a bigger one to be reused in a different network. Besides being a way of creating new processing, aggregation is a mean to hide processing design complexity addressing the forehand mentioned problem of the wiring complexity. Further research could address this and other ways of reducing it. For instance, a common issue in audio is that several channels are pipelined on equivalent processing chains by multiplying the number of elements and wires. A solution for this problem could reduce the design complexity.

6.4.4 Other multimedia areas

Ideas presented in this research could be also extended to other multimedia areas than just audio. Video and image processing environments could benefit from prototyping tools.

The ability to handle multiple types of data tokens, is perfect to allow high level processing of video. Currently, platforms that provides such data flexibility exist. GStreamer, for instance, provides full routing of complex objects through a processing network and, being a free software platform, it has a wide adoption in the industry. But such platforms are available just

at the programming level. Full visual prototyping and visual binding to a user interface to control and monitor the stream would be valuable.

6.4.5 New devices

Current interface prototyping tools address common arena of desktop computers where the interface paradigm has been stable for so many years. Nowadays, we have a large diversity of computerized devices to interact with. Ubiquitous computing brings new opportunities to architectures similar to the one exposed by embedding rich audio applications in such devices: For instance, adding voice transformation on cellular phones. But those devices imposes new requirements to the interface building and old tools for interface prototyping are becoming obsoleted.

For example, on new devices we can not assume a keyboard and a pointer as input devices, cellular phones just have a numeric keypad and they have input, some devices have no keyboard, and just have a pointer, other provide their custom buttons... Display capabilities may changes and also, we should cope with processing and audio input/output limitations.

Such diversity of devices is a new challenge for interface prototyping, but also for audio processing prototyping. The architecture already provides audio back-end abstraction but there is no way of abstracting the interface for the device input/output capabilities.

Being such devices a commodity, one interesting research area could be to define a device prototyping tool where all, software and hardware interface and the audio processing core can be designed.

6.5 Final summary

This research have demonstrated how the development process of audio applications can be fastened by providing visual means to construct them. We have focused on a reduced but very common subset of audio applications offering an architecture that joins two existing technologies: data-flow visual languages and graphical interface builders. Exercising such architecture addressing different problems has also provided useful insight for future research on extending visual prototyping beyond its current limits.

Appendix A

Related publications by the author

In this annex, we provide a list of publications which are relevant to this thesis in which its author has participated. Abstracts and electronic versions of most of these publications, as well as a list of other publications from the author non related to this dissertation are available from http://www.iua.upf.es/mtg

A.1 Presentations on conferences

Authors: Garcia, D. Arumí, P. Amatriain, X.
Year: 2007.
Title: 'Visual prototyping of audio applications.'
Conference: Proceedings of Linux Audio Conference 2007; Berlin.
Related to: chapter **??**

Authors: Garcia, D. Arumí, P. Amatriain, X.
Year: 2006.
Title: 'Extracció d'acords amb l'Anotador de Música de CLAM'
Conference: Proceedings of V Jornades de Programari Lliure; Barcelona.
Related to: chapter **??**

Authors: Amatriain, X. Arumí, P. Garcia, D.
Year: 2006.
Title: 'CLAM: A Framework for Efficient and Rapid Development of Cross-

platform Audio Applications'
Conference: Proceedings of ACM Multimedia 2006; Santa Barbara, CA.
Related to: chapter **??**.

Authors: Arumí, P. Garcia, D. Amatriain, X.
Year: 2006.
Title: 'A Data Flow Pattern Language for Audio and Music Computing'
Conference: Proceedings of Pattern Languages of Programs 2006; Portland, Oregon
Related to: section **??**.

Authors: Herrera, P. Celma, O. Massaguer, J. Cano, P. Gómez, E. Gouyon, F. Koppenberger, M. Garcia, D. G. Mahedero, J. Wack, N.
Year: 2005.
Title: 'Mucosa: a music content semantic annotator'
Conference: Proceedings of 6th International Conference on Music Information Retrieval; London, UK
Related to: chapters **??** and **??**.

Authors: Amatriain, X. Massaguer, J. Garcia, D. Mosquera, I.
Year: 2005.
Title: 'The CLAM Annotator: A Cross-platform Audio Descriptors Editing Tool'
Conference: Poster presented at 6th International Conference on Music Information Retrieval; London, UK
Related to: chapters **??**, **??** and **??**.

Authors: Celma, O. Gómez, E. Janer, J. Gouyon, F. Herrera, P. Garcia, D.
Year: 2004.
Title: 'Tools for Content-Based Retrieval and Transformation of Audio Using MPEG-7: The SPOffline and the MDTools'
Conference: Proceedings of 25th International AES Conference; London, UK
Related to: chapter **??**.

Authors: Arumí, P. Garcia, D. Amatriain, X.
Year: 2003.
Title: 'CLAM, Una llibreria lliure per Audio i Música'
Conference: Proceedings of II Jornades de Software Lliure; Barcelona, Spain
Related to: chapters **??** and **??**.

Authors: Amatriain, X. de Boer, M. Robledo, E. Garcia, D.
Year: 2002.
Title: 'CLAM: An OO Framework for Developing Audio and Music Applications'
Conference: Proceedings of 17th Annual ACM Conference on Object-Oriented Programming, Systems, Languages and Applications; Seattle (USA)
Related to: chapters **??** and **??**.

Authors: Garcia, D. Amatriain, X.
Year: 2001.
Title: 'XML as a means of control for audio processing, synthesis and analysis'
Conference: Proceedings of MOSART Workshop on Current Research Directions in Computer Music; Barcelona, Spain
Related to: chapter **??**.

A.2 Theses

Authors: Garcia, D.
Year: 2002.
Title: 'Suport de XML/MPEG-7 per una llibreria de processat d'àudio i música.'
Type: Master Thesis
Institution: Enginyeria La Salle. Barcelona
Related to: chapters **??** and **??**.