An Object-Oriented Metamodel for Digital Signal Processing

with a focus on Audio and Music

Xavier Amatriain

Departament de Tecnologia

Universitat Pompeu Fabra

Doctorat en Informàtica i Comunicació Digital

2004

This thesis entitled:
An Object-Oriented Metamodel for Digital Signal Processing with a focus on Audio and Music
written by Xavier Amatriain
has been approved for the Departament de Tecnologia

Directed by: Dr. Xavier Serra

Date: October 2004

A thesis submitted to the

Departament de Tecnologia
de la Universitat Pompeu Fabra

in partial fulfillment
of the requirements for the degree of

Doctor per la Universitat Pompeu Fabra

Amatriain, Xavier (Doctorat en Informàtica i Comunicació Digital)
An Object-Oriented Metamodel for Digital Signal Processing with a focus on Audio and Music
Thesis directed by Dr. Xavier Serra
October 2004

Abstract

Classical models for information transmission such as Shannon and Weaver's still tend to be looked at as the only possible scenarios where signal processing applications can be formally modeled. Meanwhile, other disciplines like Computer Science have developed different paradigms that offer the possibility of looking at the same problem from a different perspective.

One of the most favored approaches for software analysis and design is the Object Oriented paradigm, which proposes to model a system in terms of objects and relations between objects. An object is an instance of a real world or abstract concept and it is made up of an identity, a state, and a behavior. An object oriented system is thus described in terms of its internal objects, messages that are passed in between them and the way these objects respond to incoming messages by executing a particular method.

Although object oriented technologies have been applied to signal processing systems, no previous comprehensive approach has been made to translate all the advantages and consequences, both practical and formal, of this paradigm to the signal processing domain.

This dissertation defends the thesis that a generic signal processing system can be thoroughly and effectively described using the object oriented paradigm. For doing so, the Digital Signal Processing Object Oriented Metamodel offers a classification of signal processing objects in terms of their role in a DSP system. Objects are classified into two main categories: objects that process and objects that act as data containers. This OO metamodel turns out to be closely related to Dataflow Process Networks, a graphical model of computation that has already proven useful for modeling signal processing systems. In our study we highlight the similarities of both models to conclude that object-orientation is in fact a superset of process-oriented models and therefore the object-oriented paradigm can be proposed as a general approach for system modeling. Furthermore, it turns out that nowadays the natural target for many signal processing applications is the computer and its software environment and the object-

oriented paradigm becomes a natural conceptual framework where the different development phases fit.

CLAM (C++ Library for Audio and Music) is a framework for developing music and audio applications that has been designed bearing this conceptual model in mind. CLAM is both the origin and the proof of concept of the Metamodel. On one hand its design process and rationale has led to the definition of the metamodel. On the other hand, it demonstrates that the metamodel proposed is more than an abstract wish-list and can be used to model working and efficient applications in the music and audio domain.

The basic Object Oriented metamodel for signal processing systems can be extended to include the idea of Content Based Processing. OO concepts like Inheritance Hierarchies, Polymorphism or Late Binding can be used to model run-time classification of media objects and to deal with the semantic information present in the signal rather than just treating the signal itself. This leads us to the definition of a new metamodel of information transmission that, unlike the traditional ones, does care about meaning.

Finally, the OO paradigm can also be used to model higher-level symbolic domains related to signal processing. For example, music (as a whole) can be effectively modeled using the OO paradigm. An OO model for music is proposed as an instance of the basic signal processing metamodel and the MetriX language is presented as its proof of concept.

Resum

Els models clàssics de transmissió de la informació com el de Shannon i Weaver encara se solen considerar com els únics escenaris possibles en els que aplicacions de processament del senyal es poden modelar formalment. Mentrestant, altres disciplines com la Informàtica han desenvolupat paradigmes diferents que ofereixen la possibilitat de mirar el mateix problema des d'una perspectiva different.

Una de les aproximacions més utilitzades per anàlisi i disseny de programari és el paradigma Orientat a l'Objecte, el qual proposa modelar un sistema en objectes i relacions entre objectes. Un objecte és una instància de un concepte abstracte o del món real que està composat d'una identitat, un estat i un comportament. D'aquesta manera un sistema orientat a l'objecte es descriu en funció dels seus objectes interns, els missatges que es passen entre ells i la forma que aquests objectes responen als missatges entrants executant un mètode concret.

Tot i que les tecnologies orientades a l'objecte s'han aplicat a sistemes de processament del senyal, no hi ha cap intent previ de traslladar tots els avantages i conseqüències, tant pràctiques com formals, d'aquest paradigma al domini del processament del senyal.

Aquest treball defensa la tesi de que un sistema de processament del senyal genèric es pot descriure completament i de forma efectiva utilitzant el paradigma orientat a l'objecte. Per fer-ho, el Metamodel de Processament Digital del Senyal Orientat a l'Objecte ofereix una classificació d'objectes segons el seu rol en un sistema. Els objectes es classifiquen en dues categories principals: objectes que processen i objectes que actuen com a contenidors de dades. Aquest metamodel OO resulta estar molt proper a les Xarxes de Processos amb Fluxe de Dades, un model gràfic de computació que ja ha mostrat la seva utilitat per a modelar sistemes de processament del senyal. En el nostre estudi destaquem les similituds dels dos models per concloure que la orientació a l'objecte és de fet un supra conjunt dels models orientats al procés i que, par tant, el paradigma orientat a l'objecte pot ser proposat com una aproximació genèrica al modelatge de sistemes. A més a més, resulta que avui dia l'entorn destí de moltes

aplicacions de processament del senyal és l'ordinador i el seu programari associat i el paradigma orientat a l'objecte esdevé un entorn conceptual natural on les diverses fases de desenvolupament s'adapten.

CLAM (C++ Library for Audio and Music) és un entorn per a desenvolupar aplicacions d'àudio i música que s'ha dissenyat tenint en ment aquest model conceptual. CLAM és tant l'origen com la prova de concepte del Metamodel. Per una banda el seu procés de disseny ha conduit a la definició del metamodel. Per altra banda, demostra que el metamodel proposat és més que una llista de desitjos abstracta i que pot ser utilitzat per a modelar aplicacions pràctiques i eficients en el domini concret de l'àudio i de la música.

El metamodel bàsic de processament de senyal Orientat a l'Objecte es pot extendre per a incloure la idea de Processament Basat en el Contingut. Conceptes OO com ara Jerarquies d'Herència, Polimorfisme o Enllaç Tardà es poden utilitzar per a modelar classificació en temps d'execució d'objectes media o per gestionar la informació semàntica present en el senyal, en comptes de només tractar el senyal en ell mateix. Això ens porta a la definició d'un nou metamodel de transmissió de la informació que, a diferència dels tradicionals, es preocupa del significat.

Finalment, el paradigma OO també es pot utilitzar per a modelar dominis simbòlics de més alt nivell relacionats amb el processament del senyal. Per exemple la música (en tot el seu abast) es pot modelar de forma efectiva utilitzant el paradigma OO. Es proposa un model OO de la música com una instància del metamodel bàsic de processament del senyal, i el llenguatge MetriX es presenta com la seva prova de concepte.

Resumen

Los modelos clásicos de transmisión de información com el de Shannon y Weaver todavía se suelen considerar como los únicos escenarios posibles en los que aplicaciones de procesado de señal se pueden modelar formalmente. Mientrastanto, otras disciplinas como la Informática han desarrollado paradigmas diferentes que ofrecen la posibilidad de mirar el mismo problema des de una perspectiva diferente.

Una de las aproximaciones más utilizadas para el análisis y diseño de software es el paradigma Orientado a Objetos, el cual propone modelar un sistema en objetos y relaciones entre objectos. Un objeto es una instancia de un concepto abstracto o del mundo real compuesto de una identidad, un estado y un comportamiento. De este modo un sistema orientado a objetos se describe en función de sus objetos internos, los mensajes que se pasan entre ellos y la forma que estos objetos responden a los mensajes entrantes ejecutando un método concreto.

Aunque las tecnologías orientadas a objectos se han aplicado a sistemas de procesado de señal, no hay ningún intento previo de trasladar todas las ventajas y consecuencias, tanto prácticas como formales, de este paradigma al dominio del procesado de señal.

Este trabajo defiende la tesis de que un sistema de procesado de señal genérico se puede describir completamente y de forma efectiva utilizando el paradigma orientado a objetos. Para hacerlo, el Metamodelo de Procesado de Señal Orientado a Objetos ofrece una clasificación de objetos según su rol en un sistema. Los objetos se clasifican en dos categorías principales: objetos que procesan y objetos que actúan como contenedores de datos. Este metamodelo OO resulta estar muy cercano a las Redes De Procesos con Flujos de datos, un modelo gráfico de computación que ya ha mostrado su utilidad para modelar sistema de procesado de señal. En nuestro estudio destacamos las similitudes de los dos modelos para concluir que la orientación a objetos es de hecho un supra conjunto de los modelos orientados al proceso y que, por lo tanto, el paradigma orientado a objetos se puede proponer como

una aproximación genérica al modelado de sistemas. Además, resulta que hoy en día el entorno destino de muchas aplicaciones de procesado de señal es el ordenador y su software asociado y el paradigma orientado a objetos resulta un entorno conceptual natural donde las diversas fases de desarrollo se adaptan.

CLAM (C++ Library for Audio and Music) es un entorno para desarrollar aplicaciones de audio y música que se ha diseñado teniendo en mente este model conceptual. CLAM es tanto el origen como la prueba de concepto del Metamodelo. Por un lado su proceso de diseño ha conducido a la definición del metamodelo. Por otro lado, demuestra que el metamodelo propuesto es más que una lista de deseos abstracta y que puede ser utilizado para modelar aplicaciones prácticas y eficientes en el dominio concreto del audio y la música.

El metamodelo básico de procesado de señal Orientado a Objetos se puede extender para incluir la idea de Procesado Basado en el Contenido. Conceptos OO com las Jerarquías de Herencia, el Polimorfismo o el Enlace Tardío se pueden utilizar para modelar la clasificación en tiempo de ejecución de objetos media o para gestionar la información semántica presente en la señal, en vez de tan sólo tratar la señal en ella misma. Esto nos lleva a la definición de un nuevo metamodelo de transmisión de la información que, a diferencia de los tradicionales, sí que se preocupa del significado.

Finalmente, el paradigma OO también se puede utilizar para modelar nuevos dominios simbbólicos de más alto nivel relacionados con el procesado de señal. Por ejempló, la música (en todo su alcance) se puede modelar de forma efectiva utilizando el paradigma OO. Se propone un modelo OO de la música como instancia del metamodelo básico de procesado de señal, i el lenguaje MetriX se presenta como su prueba de concepto.

Als meus nens Aitor i Adriana, aquells que comparen el procés d'escriure una tesi amb tenir un fill és perquè no saben què és tenir-ne un. I a la Natàlia que ho ha fet possible.

To my children Aitor and Adriana, those who compare writing a thesis with having a child is because they do not know what it is to have one. And to Natalia who has made it possible.

Acknowledgements

This thesis would be unthinkable in an environment different from that of the Barcelona Music Technology Group. And my first acknowledgement goes for the person who made it possible for me to become a member of the MTG and to travel all the way to the end of this PhD while doing many interesting projects: Xavier Serra. Many other people at the MTG have contributed to this thesis in some way or another. I should at least mention Perfecto Herrera for all his fruitful comments and Jordi Bonada for being our personal signal processing consultant and for helping me out since I started my master thesis.

But above all I want to thank all the wonderful developers that have contributed to the CLAM framework in some way, without them this thesis would not have been possible. Among these, particularly important have been Pau Arumí, Maarten de Boer, David Garcia and Enrique Robledo.

Very special thanks go to my wife Natalia and our children Aitor and Adriana. Because of them this thesis did not become an obsession, at least no more than strictly necessary.

Agraïments

Aquesta tesi hauria estat impensable en un entorn diferent del que he gaudit al Grup de Tecnologia Musical de la Universitat Pompeu Fabra. I el meu primer agraïment és per aquell que em va permetre entrar a formar part del MTG i viatjar tot el camí fins al final d'aquests estudis de doctorat mentre també feia molts altres projectes interessants: Xavier Serra. Molta altra gent del MTG han contribuit a aquesta tesi d'una forma o una altra. Al menys hauria de mencionar al Perfecto Herrera per tots els seus comentaris i al Jordi Bonada per ser el nostre consultor personal de processament de senyal i ajudarme sempre que ho he necessitat desque vaig començar el meu projecte final de carrera.

Però sobretot vull agrair a tots els maravellosos desenvolupadors que han contribuit a CLAM d'alguna manera, sense ells aquesta tesi no hauria estat possible. Entre d'ells han estat especialment importants en Pau Arumí, el Maarten de Boer, el David Garcia i l'Enrique Robledo.

Els meus agraïments més especials són per a la meva dona Natàlia i els nostres fills Aitor i Adriana. Gràcies a ells aquesta tesi no s'ha convertit en una obsessió, al menys no més de l'estrictament necessari.

Contents

List of Figures

Figure

List of Tables

Table

Introduction

In this introduction we will define the general scope and goals of this thesis, outlining its most important contents and setting the basis for the conceptual framework that will be later described.

First, it is important to describe its contextual situation. The work here presented is the result of our research in the Music Technology Group of the Institut Universitari de l'Audiovisual at the Universitat Pompeu Fabra in Barcelona. This group focuses its research on the analysis, processing and synthesis of audio in general and musical signals in particular. The research area defined as Musical Technology is very much related, though broader in its scope, to Computer Music. The basic disciplines that are mostly related to it are Computer Science and Signal Processing.

Nevertheless, this Thesis is not about Computer Science nor about Signal Processing techniques. We would ascribe its content to the area of Software Engineering. Although it may not seem obvious in the first place, Software Engineering uses completely different approaches from Computer Science, although it shares some of its basis. Computer Science as a discipline is more related to abstract sciences like Mathematics than to practical ones such as Biology. On the other hand Software Engineering is not a science in its methodology or its approaches, it is an engineering more related in this sense to other engineering areas or architecture. As Herbert A. Simon puts it "Engineering is about synthesis and science is about analysis" [Simon, 1996].

Because of this, the formal approach in this work is somehow different to what one would may expect in the first place. Very few mathematical formulas are included and no experimental data results are thoroughly analyzed. On the other hand formal modelling languages such as UML are used throughout. Software engineering is still a fairly new discipline and as such not many research models are available and some of them are too much influenced by the tradition of Computer Science. With this thesis we hope also to have contributed in this direction.

This thesis has five different parts that may be read and treated differently. Chapter 1 intro-

duces the concepts and terms that will be use throughout the following sections. It should not only be seen as a documentation exercise but also as a presentation of a conceptual framework that is new, maybe not in its parts but as a whole. In chapter 2 we introduce a thorough review of environments for music and audio processing, all of them represent a different approach but are also somehow related to both the conceptual and development frameworks that will be presented in the next chapters. Chapter 3 contains the more practical side of this work as it presents a framework for audio and music applications we have developed and implemented and from which the metamodel presented as the main thesis of this work has been abstracted. Chapter 4 presents the core thesis of this work, introducing an object-oriented metamodel for signal processing derived from the framework presented in the previous chapter. Finally, in chapters 5 and 6, two different and complementary extensions are given to the basic model. The first one deals with the addition of a higher abstraction layer and the idea of using the framework as a base for content processing. The second one enhances the object-oriented metamodel by adapting it to the music domain and gives some examples of implementations.

We will now try to offer a more schematic view of this thesis. This may be used as a guide for the whole work here presented if, reading one of the parts, it is not clear how it is related to the main thesis or hypothesis.

- Main Thesis: The object-oriented paradigm may be used to model any signal processing system in an effective and thorough manner.
  - *Hypothesis 1*: The Object-Oriented model is not just another model, it is the **paradigm** to use when modeling a system.
    - *Hypothesis 1.1*: By using the Object-Oriented paradigm to model the signal processing domain we come up with a graphical Metamodel of Computation that can be instantiated to model any signal processing system. This graphical metamodel is very much related to Process Networks.
  - *Hypothesis 2*: Using this metamodel for signal processing, it is possible to derive a new communication metamodel as an extension of the classical Shannon and Weaver theory.
    - *Hypothesis 2.1*: The object-oriented metamodel for communication promotes the importance of meaning and semantics in the communication chain.
  - *Hypothesis 3*: Frameworks generate metamodels
    - *Hypothesis 3.1*: Although traditionally the first thing to do when implementing a framework is to come up with a domain model, using an incremental process methoda we are

able to see it the other way around: it is the process of implementing a reusable framework that generates the domain metamodel.

* *Hypothesis 4*: The object-oriented paradigm is also useful for modeling more symbolic domains like music.
  - *Hypothesis 4.1*: Music - and by extension any other symbolic domain related to signal processing - is also well suited for using the object-oriented paradigm as long as a system is properly identified, this paradigm acting as a bridge between different abstraction levels. The resulting music model is in fact a particular instance of the basic object-oriented metamodel for signal processing.

All these different thesis are also backed by practical examples that demonstrate their viability. Out of all these examples the CLAM framework, presented in chapter 3 is the most important as it sets the basis for the general metamodel that will be later particularized, extended or instantiated.

As a matter of fact one may ask how hypothesis are tested in a discipline such as Software Engineering. The way hypothesis about models and metamodels related to Software Engineering are validated is very similar to how hypothesis are tested in experimental sciences: by empirical testing. The only difference is that instead of analyzing resulting data to conclude whether it fits the model under study, here we analyze the systems derived from the metamodel to conclude whether they are useful in the domain under study. That is why, of all the metamodels and models here presented include a practical or empirical demonstration of their validity. Namely:

- Hypothesis: DSPOOM (see 4) — Validation: CLAM (see 3)
- Hypothesis: OOCTM (see 5) — Validation: Several related applications (see 5.3.4 and 5.4)
- Hypothesis: Object-Oriented Music Model (see 6) — Validation: MetriX (see 6.4)

§ Initial Glossary

A large part of this work is about concepts and definitions. All of them will be thoroughly defined and commented in the corresponding sections. Nevertheless, and in order to establish a common glossary and start stating a point of view, it is interesting to first give a short definition of some terms.

Object:Physical or abstract entity that can be conceptually isolated due to its high cohesion and low coupling with other entities.

Class: A set of related objects with the same behaviour and internal structure.

System: We will see throughout this thesis how the word system has slightly different meanings and interpretations depending on the context. However, we will accept as a sufficiently generic definition, the one given by the Institute of Electrical and Electronical Engineerings professional group in System Science and Cybernetics when they state that a system is " a large collection of interacting functional units that together achieve a defined purpose" [Rowe, 1965].

Model: A model can be understood as the formal abstract representation of a given system. A single system can be represented through different models, depending on the level of abstraction required and foreseen use.

Metamodel: A model that explains a set of related models.

Paradigm: The most commonly accepted definition of paradigm is that of Thomas Kuhn who describes a paradigm as the set of common beliefs and agreements shared between scientists about how problems should be understood and addressed [Kuhn, 1962].

Framework: A framework (in the Software Engineering domain) is an abstract design of a set of related applications in a particular domain. This abstraction can be then instantiated to build a concrete application.

Metaphor: According to [Beck, 2002], a metaphor is the linking of two sets of concepts so one set of concepts is understood in terms of another.

Signal: A function of one or more independent variables that contains information about the behavior or structure of a given phenomena [Openheim and Willsky, 1997].

All these concepts and the way they relate will be further explained and discussed and will build the conceptual framework upon which this thesis is based. But as a first introduction it may be well worth to add a few considerations that can already help better understand the above given definitions and the way they conceptually relate.

§ Conceptual Roadmap

First, it may be interesting to briefly discuss what is the purpose of coming up with a domain metamodel. As a matter of fact, the main reason and advantage of working on a domain metamodel is the same as the reason for working on a system model: come up with a simplified representation of the domain that helps its understanding and usability. The domain metamodel can then be instantiated to model different systems related to the domain and therefore build a set of related applications.

Note also that when building an application framework we are also in some way trying to come up with a (practical) domain metamodel. We are in fact trying to find a simplified representation of the

domain that can be used to easily construct new applications related with that domain. A framework does not only give a set of tools that can be used as building pieces (black-box framework) but also a conceptual interpretation of the domain (white-box framework) (see 1.3.3). One way to see the process of building a framework is that we must first have to come up with a domain model and then implement it. Nevertheless, if we adopt a more incremental or "agile" point of view, we are able to see it the other way around: when we start building the framework we may have a vague idea of the domain model or metaphor but it is the process of designing the framework that actually ends-up defining the model. All these issues will be contextualized and further developed in chapter 1.

This thesis presents an object-oriented metamodel for signal processing that will be called Digital Signal Processing Object-Oriented Metamodel, DSPOOM for short. DSPOOM is a metamodel that, as it will be proven throughout this work, can be used to effectively translate any digital signal processing system into the software domain. This modeling is accomplished through the use of the object oriented paradigm and, in doing so, we gain more understanding - and thus control - of the different actors and forces present in the original system.

After this introduction, in chapter 1 we will review the basic foundational concepts upon which the thesis is built. Departing from the initial glossary this chapter will focus on establishing a solid ground by clearly defining the main axis: the concept of model, the object oriented paradigm and associated techniques and uses, and a review of different models that have been traditionally been used for modeling signal processing or digital communication systems. In this chapter we will also discuss on issues such as frameworks, models and metamodels. A somehow more extensive review is given of software engineering techniques as the readers are not expected to be experts in the software engineering domain.

Chapter 2 presents thorough review of software environments for audio and music processing. Each of them responds to a different rationale, focus and general model. The analysis of these tools is a vital task for understanding the domain in which this thesis is built. Many of these environments also represent clear predecessors, not only of the framework that will be presented in the following chapter but also of the different conceptual models and metamodels that are later derived.

In next chapter, number 3, the CLAM framework for developing audio and music applications is presented. CLAM is at the same time a proof of concept and the generating seed of the model presented in this Thesis and commented in the next chapter. As a matter of fact, implicit to this thesis is the idea that frameworks generate models, and not the other way around (see 1.3.5).

In chapter 4 the main issues and concepts that form DSPOOM are discussed. The central part of this chapter is devoted to explaining a classification of digital signal processing objects that may be

used to model a system.

In chapter 5 we defend that the model presented, and the framework that realizes it, can be effectively used for working with new paradigms that go beyond the classical signal processing framework. More precisely, DSPOOM is shown to be useful as the basis for a content processing or semantic framework.

In chapter 6 we finish the excursion up the abstraction ladder and end up presenting an object oriented model for music (i.e. symbolic) processing. It is not the first time that music and objects meet, but here we discuss how the model is related to the one presented for the lower signal level and can use many tools included in the CLAM framework.

Finally, in chapter 6.5 we draw some conclusions and envision some future work that could be used to broaden the scope of this Thesis.

§ Contributions

We will now briefly summarize the main contribution of this Thesis.

- An explicit relation between the OO paradigm, system engineering, and graphical Models of Computation.
- A general object-oriented metamodel for digital signal processing (DSPOOM) that combines the benefits of object-orientation and graphical models of computation.
- The CLAM framework as a particularization of the previous metamodel in the audio and music domain.
- An extension of the basic DSPOOM metamodel for including the concept of content processing and translate the benefits of object-orientation also to this domain and giving place to the Object-Oriented Content Transmission Metamodel.
- An object-oriented model of music as an instance of the DSPOOM metamodel in which we deal with symbolic data and the MetriX language as its implementation.

§ Extended Summary

And to finish this introduction and in order to help the reader grasp a previous idea of this Thesis we will now present an extended summary of its different chapters[1].

Chapter 1. Foundational Issues

In chapter 1 we will see the various building blocks upon which the thesis is built. We will now briefly summarize the most important ideas introduced in the different sections.

In section 1.1.1 we introduce the most important concepts related to the object-oriented paradigm. An *object* is a real-world or abstract entity made up of an identity, a state, and a behavior. A *class* is an abstraction of a set of objects that have the same behavior and represent the same kind of instances. The object-oriented paradigm can be deployed in the different phases of a software life-cycle and the *UML* language supports most of the activities contained in them. Apart from the concepts of object and class and the different kinds of relationships that can be established between objects and classes, other concepts such as *encapsulation*, *inheritance hierarchies* or *polymorphism* are important for fully understanding the object-oriented paradigm. The object-oriented paradigm presents many different advantages that can be summarized in: it maps more directly to real world concepts, it enhances *encapsulation*, it improves *information hiding*, it promotes good structuring and it favors re-use. Finally, it is important to note that, although it is often considered otherwise, object-orientation does not imply less efficient code and furthermore its techniques make it even easier to end up having more efficient and robust final results.

In the next section, 1.2, we define the main concepts related to *models* and *systems*. The most commonly accepted definition of a system is that by Hall and Fagen in which they define a system as "a set of objects together with relationships between the objects and between their attributes." On the other hand, a model is an abstract representation of a system with a well-defined purpose, different models may exist for a single system. It is also important to note that the birth of object-orientation is very much related to the study of system simulations by Kristan Nygaard. Finally we define a metamodel as model of models, that is an abstract model that can be used to model a collection of related models.

In section 1.3 we address the issue of software framework development. Although many different definitions can be given for a software framework it is probably that in [Johnson and Foote, 1988] the one that is most commonly accepted. According to this definition,"a framework is a set of classes that embodies an abstract design for solutions to a family of problems". Frameworks offer a way to

[1] The different parts that make up this summary are also included with minor modifications and adding conclusions in the "Summary and Conclusions" section at the end of each chapter.

reuse analysis, design and code. Frameworks can be classified, among other ways, into *white-box* and *black-box*. In white-box frameworks users extend previously existing classes, particularizing for their specific needs. On the other hand, black-box frameworks offer ready-to-use components that can be used as building blocks for an application. Although different approaches may be used for developing a software framework, it is usually recommended to use an application-driven methodology, using a limited amount of already existing applications as the driving force and favoring user-feedback as much as possible. Finally, a well-designed software framework can become a sort of metamodel in itself as it will offer a model of models for a given domain.

Metadata is defined as "data about data". In section 1.4 we introduce the most important concepts and tools related to metadata and our domain of object-orientation and multimedia signal analysis. XML is a general-purpose tagged language that is rapidly becoming the standard for metadata annotation of any sort. Using this same language, MPEG-7 is an ISO proposed standard for multimedia annotation. On the other hand the Object Management Group of the ACM has also proposed the MOF (Meta Object Facility) standard as a metadata management framework for object-oriented systems and technologies.

Graphical Models of Computation are abstract representations of a family of related computer-based systems that use a graph-based representation as the primary way of communicating information about the system. There are many different graphical MoC's, each of them particularly well-suited for some purpose. The most important are outlined in section 1.5. In the context of signal processing applications, Kahn Process Networks and related models such as Dataflow Networks are of particular importance. Although some authors defend that these models should be seen as instances of the Process-Oriented paradigm we defend the thesis that process-orientation or actor-orientation is not more than a particular instance of the object-oriented paradigm.

Chapter 2. Environments for Audio and Music Processing

In this chapter we present a thorough overview of audio and music processing environments. Although all of them have different scopes and motivations, we present a classification in different categories. These categories are summarized in the following list:

(1) *General purpose signal processing and multimedia frameworks*: software frameworks for manipulating signals or multimedia components in a generic way. The most important examples in this category are Ptolemy and ET++.

(2) *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 is the most important framework analyzed in this sub-category.

(b) *Synthesis Oriented*: Audio processing frameworks that focus on generating output audio from input control signals or scores. Here it is important to mention STK.

(c) *General Purpose*: General purpose Audio processing frameworks offer tools both for analysis and synthesis. Out of the ones presented in this sub-category both SndObj and CSL are in a similar position, having in any case some advantages and disadvantages but no being very mature.

(3) *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.

(4) *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.

(5) *Music languages*: In this category we present different languages that can be used to express musical information. We have excluded those having a graphical metaphor, which are already listed in the previous category.

(a) *N-Music languages*: Music-N languages base their proposal on the separation of musical information into static 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 included in this section, because of its acceptance, is CSound.

(b) *Score languages*: These languages are simply ways of expressing information in a musical score, usually based on a textual or readable format.

The basis that we will set in our analysis of the state of the art in our particular domain will be used for both constructing our proposals and also comparing the final results.

Chapter 3. The CLAM Framework

In this chapter we present the CLAM framework. This software framework is a comprehensive environment for developing audio and music applications. It may be also used as a research platform

for the same domain. CLAM can be seen both as the origin and the prove of concept of the conceptual models and metamodels that are included in this thesis.

CLAM is written in C++, it is efficient, object-oriented, and cross-platform. It presents a clean and clear design result of applying thorough software engineering techniques. The framework can be used as a black-box, relying on the offered *repository*, or as a white-box framework, extending its functionality through its *infrastructure*.

CLAM's repository is made up of a large collection of signal processing algorithms encapsulated as *Processing* classes and a number of data structures included in its *Processing Data* repository. The Processing repository basically includes algorithms for signal analysis, synthesis and transformation. Furthermore it also includes encapsulated platform and system-level tools such as audio and MIDI input/audio both in streaming and file mode. On the other hand the Processing Data repository offers those data types that are needed as inputs or outputs of the processing algorithms. These include classes such as Audio, Spectrum or Fundamental Frequency. It also includes a collection of statistical *Descriptors* that can be obtained from the basic Processing Data objects.

On the other hand CLAM's infrastructure offers ways of extending the already existing repository by deriving new Processing or Processing data classes. In the case of Processing classes this is accomplished by a simple inheritance mechanism in which the user is forced to implement some particular behavior in his/her concrete Processing class. Mechanisms for composing with Processing objects, handling input and output data through *Ports* and control data through *Controls* are also offered. The Processing Data Infrastructure is based on CLAM's Dynamic Types. This is a special C++ class that, using macros and template metaprogramming techniques, offers a very simple way of creating data containers with a homogeneous interface and automatic services such as introspection or passivation facilities. CLAM's infrastructure is completed by a set of tools for platform abstraction, such as audio and MIDI or multithreading handling mechanisms, a cross-platform toolkit-independent visualization module, XML serialization facilities or application skeletons.

CLAM also offers a number of usage examples and ready-to-use applications. These applications include SMSTools, a graphical environment for audio analysis/synthesis/transformation, and Salto, a spectral-sample based sax and trumpet synthesizer. Another important application is the Network Editor, a graphical tool for creating CLAM Networks using a graphical boxes-and-connections metaphor ala Max. This application can be used as a rapid prototyping and research tool. But CLAM has also been used in many other internal projects for instance for developing a voice processing VST plugin, a high-quality time-stretching algorithm or content-based analysis applications.

Chapter 4. The Digital Signal Processing Object-Oriented Metamodel

In this chapter we present the Digital Signal Processing Object-Oriented Metamodel (or DSPOOM for short). This metamodel may be considered the main contribution of this thesis and is basically a result of abstracting the conceptual conclusions found in developing the CLAM framework.

DSPOOM combines the advantages of the object-oriented paradigm with system engineering techniques and particularly with graphical Models of Computation in order to offer a generic metamodel that can be instantiated to model any kind of signal processing related system.

To do so the metamodel presents a classification of signal processing objects into two basic categories: objects that process or *Processing* objects and objects that hold data or *Processing Data* objects. Processing objects represent the object-oriented encapsulation of a process or algorithm. They include support for synchronous data processing and asynchronous event-driven control processing as well as a configuration mechanism and a explicit life cycle. Data input and output to Processing objects is done through *Ports* and control data is handled through the *Control* mechanism. On the other hand Processing Data objects must offer a homogeneous getter/setter interface and support for meta object facilities such as reflection and automatic serialization services.

The metamodel also presents mechanisms for composing statically and dynamically with basic DSPOOM objects. Static compositions are called *Processing Composites* and dynamic compositions are called *Networks*.

Finally the DSPOOM metamodel can also be considered as an object-oriented implementation of a graphical Model of Computation, particularly the *Context-aware Dataflow Networks*.

Chapter 5. The Object-Oriented Content Transmission Metamodel

Here we present an object-oriented metamodel for content processing and transmission called Object-Oriented Content Transmission Metamodel or OOCTM for short. This metamodel may be seen both as an extension and a particularization of the Digital Signal Processing Object-Oriented Metamodel presented in the previous chapter and presents a way of modeling signal processing applications that deal with all aspects of content-based processing such as content analysis or content-based transformations.

The metamodel is based on two conceptual foundations: on one hand we call *content* to any semantic information that is meaningful for the target user; on the other hand, and applying one of the object-oriented paradigm maximas, we state that all semantic information contained in a given signal can be modelled as a collection of related objects.

Following the traditional Shannon&Weaver model for information transmission our metamodel is divided into three main components: a *semantic transmitter*, a *channel*, and a *semantic receiver*. The semantic transmitter is in charge of performing a multilevel analysis on the signal, identifying objects

and finally building a multilevel object-based content description and encoding it in an appropriate format such as XML. The channel transports this metadata description and any added noise will not be considered as such unless the original meaning is modified. Finally the semantic receiver receives the multilevel content description, decodes it and translates it into a synthesizer-readable format. The synthesizer included in the receiver then synthesizes the output signal.

It is important to note that we are not so much interested in the fidelity of the final synthesized signal to the original but rather on whether the original "meaning" is preserved and is useful for the final user.

The Object-Oriented Content Transmission Metamodel can be seen as an extension of the classical Shannon&Weaver model for information transmission. It is very much related to the Structured Audio metamodel and can also be seen as a step beyond parametric encoding. Finally if we add a transformation function to the channel we end-up having a general scheme for content-based transformations.

In the chapter we also give several examples of applications that represent particular instances of the metamodel or subparts. But we also present one sample application that instantiates the whole metamodel in order to transmit and synthesize a previously analyzed and extracted musical melody.

Chapter 6. An Object-Oriented Music Model

Finally in this chapter we present an object-oriented music model that can be interpreted as an instance of the basic Digital Signal Processing Object-Oriented Metamodel dealing in this case with higher-level symbolic musical data.

Following again the object-oriented paradigm we model a music system as a set of interrelated objects. These objects will in general belong to one of the following abstract classes: Instrument, Generator, Note or Score.

An *Instrument* is a generating Processing object that receives input controls and generates an output sound. An Instrument is as a matter of fact a logical grouping of autonomous units named *Generators*. A Generator is the atomic sound producing unit in an instrument and can be independently controlled from the other generators (although it often receives their influence). Examples are the six strings in a guitar or each of the keys in a piano.

A Note is the actual sounding object attached to each generator. A Note can be turned on and off and its properties depend on the internal state of its associated Generator and Instrument.

Finally the internal state of the whole object-oriented music system changes in response to events that are sent to particular Instruments or Generators. A time-ordered collection of such events is known as a *Score*.

The abstract model described is implemented in the MetriX language or in its XML-based version MetriX-ML. MetriX-ML is a Music-N language that therefore offers a way of defining both Instruments and Scores. It is implemented in CLAM and, apart from the concepts previously presented, includes support for defining timbre spaces, break-point-functions and relations between control parameters in an Instrument.

Chapter 1

Foundational Issues

In this chapter a more profound insight will be given on some concepts upon which this Thesis is built. Some of these concepts were already outlined in the Introduction in the form of a glossary. The main goal in this chapter is to justify the title of this dissertation and discuss some important terminological issues that end up becoming hypothesis in themselves.

§1.1 The Object-Oriented Paradigm

Most of this Thesis is founded on the hypothesis that the object-oriented paradigm is well suited for describing signal processing systems. We will now briefly introduce this paradigm by defining its main concepts and procedures and mentioning its advantages.

§1.1.1 Objects

When we look up the word *object* in a dictionary we may find definitions such as "a thing that can be seen or touched"; "a person or thing to which action, thought, etc. is directed"; or in more philosophical terms "anything that can be perceived by the mind".

In Software Engineering or Computer Science, though, the term has acquired a new or more specialized meaning. A software object is still an entity. The difference is that this object "lives" in the computer or software world and not in the real world. But to be useful in some way, the software object must have a more or less direct mapping to a real object in the application domain. A software object may represent a physical object in a particular domain but it may also represent an abstract concept or idea. According to the Object Management Group (OMG) of the ACM an object is "an entity that

has unique identity, a set of operations that can be applied to it, and state that stores the effect of the operations" [OMG, 2003].

Therefore, an *object* is made up of three basic components: an *identity*, a *state*, and a *behavior* [Booch, 1994b]. The identity is the property that enables distinguishing two objects with an identical state and behavior. The identity is unique and may not be shared between different objects, even at different time and does not change during the object lifetime. The object state represents the different possible internal conditions that the object may experience during its lifetime. Finally the object behavior is the way a particular object will respond to received messages.

A *class* is an abstract group of objects that behave the same way. Every existing object is the instance of a class. Not every class though may have instances. A class that cannot be instantiated is called an *abstract class*. A class is basically made up of a unique name, a number of *attributes* and a set of *operations* (the implementation of which is called *method*). The attribute values of a concrete object in a given moment define this object state. Thus attributes in a class define the possible states in which an instance of that class may be. On the other hand, the behavior of an object depends on the class operations plus the particular state of the object as operations may respond differently to input messages depending on the current state. And again according to the OMG, a class is "a classifier that describes a set of objects that share the same specifications of features, constraints, and semantics [OMG, 2003].

An object oriented system may be viewed exclusively as a set of objects that are related and exchange messages, all collaborating to accomplish a common goal that is related to the functional requirements of the system (we will thoroughly define what a *system* is in section 1.2.1). An OO message is made of a *receiver*, a *selector* and an optional *list of arguments*. The receiver is the object that will receive the message. The selector informs the receiving object of which of all the possible messages it can interpret is being sent. And finally the the optional list of arguments holds the values with which the receiver will interpret the message.

The way that objects are related in run-time is defined by the way the classes are associated in the logical view of the system. There are basically four kinds of class relationships: association, aggregation/composition, dependency and inheritance. Each relationship is established between two classes and, apart of the "kind" of relationship, we should describe its navigability (whether objects in both ends of the relation are able to see each other or this navigation is restricted in only one direction) , and cardinality (how many instances of each class can participate in the relationship). Other less important attributes in the relationship such as its name, roles of the participants or visibility may also be included.

An association is the most basic - and weakest - form of relationship. It basically represents that instances of one class know of the whole or part of instances of the other class. That means that instances of one end can access operations or attributes in the other end. All other relations, except for inheritance, can be viewed as particular cases of an association.

A stronger form of association is known as *aggregation.* An aggregation represents a relationship between some "parts" and a "whole". In an aggregation, instances in one end of the relationship "contain" or are "made up" objects of the other end. A particular case of aggregation is a composition. In a composition the life of the parts is connected to the life of its whole. When a part is instantiated, it must be already created as belonging to a whole and when the whole is destroyed, its parts are also destroyed.

Probably the most powerful technique associated to object oriented methods is the inheritance relationship. If a given class A derives from class B, class A is said to be a subclass of B while B is said to be a superclass or a base class for A. A derived class inherits all the behavior implemented in its base class but is also allowed to add or override behavior. A rule of thumb for identifying if a given class can be modeled as a subclass of another one is the 'is a rule': if the sentence "A is a B" makes sense, then probably A is a subclass of B (e.g. "A dog is a mammal").

§1.1.2 The Object Oriented Way

When analyzing a problem from an Object Oriented point of view, the first things that are usually identified are *actors* and *use cases.* An actor is an external (from the system point of view) entity that communicates with the system through a use case. A use case is a generic scenario that illustrates the different services that the system should provide. Use cases are useful in a very preliminary analysis stage but are also interesting to document the system behavior, once the design phase has finished.

Once the basic behavior of the system has been exposed, we are ready to accomplish the most important and maybe difficult task in an Object Oriented process: identify classes. This identification may be performed following one of the following approaches: (1) the application domain is analyzed and classes are identified as "important" entities that belong to the system (as opposed to actors); (2) instead of identifying classes, we concentrate on identifying objects and the way they communicate, classes are later extracted from grouping objects that have an identical behavior.I would recommend a mixed approach. First identify most important entities in the system and relations in between them. This step does not need to get into much detail so attributes and operations do not need to be completely identified. Then, scenarios should be illustrated, analyzing objects and messages in between them. This

step will surely bring to light new classes that had not been previously identified. Class operations will also be obtained by analyzing messages between objects. So, afterwards we can iterate on the first step (class identification) and repeat the iteration as many times as needed.

When identifying classes we are partitioning both the problem domain and the resulting system. An abstract and probably complex problem is methodologically converted into a number of classes. Each class should have a strong internal cohesion and low coupling should exist between the different classes (see [Larman, 2002]). Following this process a new model of the "complex" system is obtained by partitioning the problem into smaller problems that can be treated more easily.

A well defined object oriented model of a system has a more or less direct mapping into the final program code using an OO programming language. Most Computer Assisted Software Engineering (CASE) tools are thus capable of generating code from a well-formed class diagram plus some information about the dynamic behavior of objects (i.e. some sequence, collaboration, state or activity UML diagrams).The *Unified Modeling Language* (UML) is the standard used for depicting all the different steps in the OO lifecycle. UML is a language, not just a notation. As such, it includes vocabulary (i.e. definition of basic concepts), notation (the graphical way to represent model elements), and rules (guidelines on how to use notation).

The Object-Oriented paradigm and related processes define a different way of doing things. This has led to the upcoming of different methodologies that, although not strictly restricted to, are especially fit for the object-oriented paradigm and represent a step forward from the traditional waterfall model. One of the most important object-oriented methodology is the Rational Unified Process (RUP) [Kruchten, 2000] developed by the same main authors than UML as a combination of different already existing methodologies such as Booch or Objectory.

But probably the most important methodological change due to object-orientation is the one introduced by Agile Methodologies [Cockburn, 2002] in general and eXtreme Programming[Beck, 1999] in particular. These methodologies defend a lightweight developing lifecycle in which human values such as communication or courage are favored and formal documentation is avoided. Agile methodologies represent a step further from Incremental methodologies, the planning is performed in small iteration phases and continuously revised. Therefore the line between analysis and design phases is completely blurred. eXtreme Programming in particular puts a high emphasis on the code and promotes the use of practices such as Test-driven Development or Pair Programming.

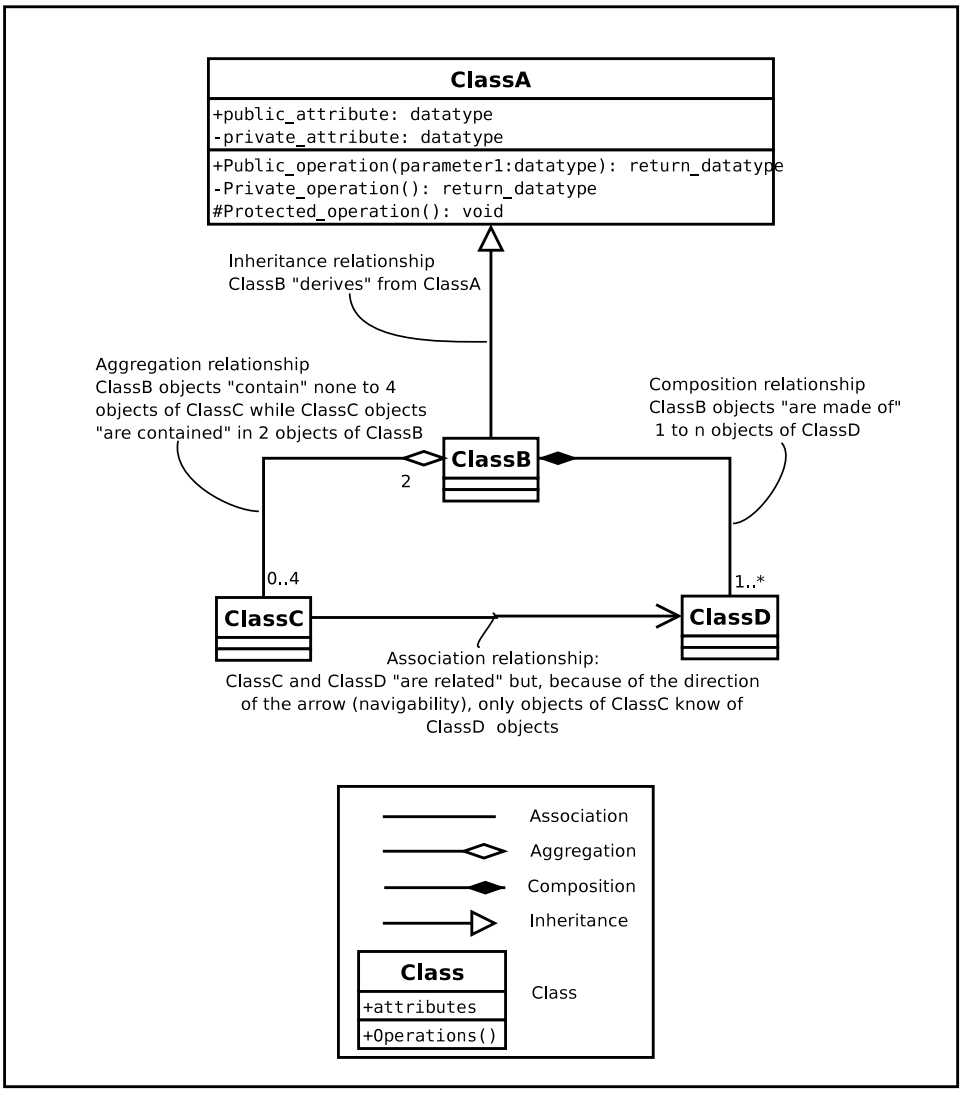

Figure 1.1: Sample UML Class Diagram

§1.1.3 Object Orientation beyond the basics

After having understood the basics that define the Object-Oriented paradigm we will now look more in depth into two of its most important properties: inheritance hierarchies and polymorphism.

§1.1.3.1 Inheritance Hierarchies

As already mentioned in previous sections, inheritance is one of the basic properties of object-oriented languages and is in part responsible for the strength of the object-oriented paradigm. We will now give some more details of its significance and properties, most of which are necessary to understand particular concepts that will be presented in next chapters.

Inheritance implies both *extension* and *contraction*. Because a derived class behavior is, strictly speaking, broader than that of its parents we can say that the child class is an extension of its parent classes. And because the derived class can *override* some of its base class behavior, it is also a contraction.

The main reasons for using inheritance is that it provides both code and concept reuse. Code reuse because the operations implemented in the base class are available in its derived classes without needing to add any extra code. But most importantly, concept reuse because even if methods in the base class are overridden and specialized, the concept that they represent is reused and a better abstraction management is possible. Other benefits of using inheritance are that it enhances robustness, it gives consistency to the interface, it couples well with rapid prototyping techniques, and it promotes information hiding and encapsulation.

But inheritance may also imply some disadvantages. Among the most commonly listed we find that it may compromise execution speed and efficiency (but see next section for a more detailed discussion on this issue), it introduces an abstraction effort overhead in the design phase, it compromises the system flexibility once the hierarchy is established, it tends to de-localize code (responsibility may be distributed in such a way that it may be difficult to identify who does what and where), and it may even introduce some code complexity. Nevertheless, all these inconvenients can be minimized to non-noticeable levels by a an organized and disciplined use of inheritance and the use of supporting technologies such as design patterns[Gamma et al., 1995], unit testing, and CASE tools.

An *abstract* class is defined as a class that cannot be instantiated and is therefore only used in the context of an inheritance hierarchy. An abstract class cannot be instantiated because it has non-defined behavior (abstract or pure virtual methods) that must be defined in any non-abstract derived class. An *interface* class is one that has no defined behavior. Therefore all its methods are abstract and it can be considered as an extreme case of the abstract class. Obviously an interface class cannot be instantiated either and is only used as a way of specifying an interface to which all derived classes should

conform. It is indeed a clear example of *concept reuse* as opposed to *code reuse* in the object-oriented paradigm.

If we look at the origin and properties for the inheritance relationships, we may classify them into different categories:

- *Specialization*: the derived class is a special case of its base class, it specializes its behavior and it is a subtype.
- *Generalization*: the base class is obtained as result of finding common behavior in different classes that become its children; these derived classes override some of the methods in the base class.
- *Specification*: the base class defines some behavior that it is only implemented in the derived class.
- *Extension*: the derived class adds some behavior but does not change the inherited behavior.
- *Combination*: the derived class inherits some behavior from more than one base class (multiple inheritance).
- *Construction*: the derived class uses the behavior implemented in the base class but it is not a subtype.
- *Limitation*: the derived class restricts the use of some of the behavior implemented in the base class.
- *Variance*: the derived and base class are variants one of the other and the relation class/subclass is arbitrary.

Out of all these kinds of inheritance the first two are by far the more common and most of the others can be see as a special case of one of them. The last three are not recommended. On the other hand, while the origin and intent is different in all of the different inheritance kinds, the result of applying them may be indistinguishable. Particularly, by looking at an inheritance hierarchy it is usually impossible to decide whether it has been the result of a generalization or specialization process. In a generalization process the derived classes exist before and the base class is obtained by realizing the commonalities in them. In the specialization process the base class is "broken down" into different derived classes.

Inheritance hierarchies allow to manage a system complexity. We want all branches in an inheritance hierarchy to be *disjoint* and *balanced*. Disjoint means that any given object may not be an instance of two of the subclasses found at the same level of the hierarchy[1]. By balanced we mean that

[1] This rule may be broken if multiple inheritance is used at an intermediate level of the hierarchy but it is beyond the scope of this summary to detail this special case.

two subclasses should represent comparable sized sets of objects, having a very general class and a very particular one at the same level of the hierarchy is not a good idea.

Inheritance can also be classified into *static* or *dynamic*. In a static specialization an object belonging to a particular level of the hierarchy is not intended to change in run-time from one subclass to another. In a dynamic specialization the belonging of the object to a particular subclass depends on its state and therefore can change on run-time. Dynamic inheritance, although theoretically correct, introduces many practical problems and is often substituted, following the *delegation* principle, by an association and a static inheritance.

Delegation can be used in other situations in order to reduce coupling. This mechanism introduces another indirection and therefore the client needs not to know the provider and this provider can change dynamically. Delegation also allows multiple inheritance to be implemented in single inheritance programming languages.

§1.1.3.2 Polymorphism

Polymorphism refers to the ability of executing different operations in response to the same message. Polymorphism can also be static or dynamic. In dynamic polymorphism the response to the message is decided on run-time while in static polymorphism it is decided out of run-time (i.e. on compile-time). The mechanism used to link the message with the method in dynamic polymorphism is known as late or dynamic binding while if the linking is done on compile time it is known as early or static binding.

Polymorphism is implemented using four different mechanisms:

- *Overloading*: an operation name refers to two different method implementations that differs either in its scope or signature. It uses early binding.
- *Generics*: also called *templates*, they define a way to declare generic classes parameterizing some of its types. It uses early binding.
- *Overriding*: a derived class re-defines a method inherited from the base class. It may use late binding.
- *Polymorphic variable*: also called assignment polymorphism, it refers to a variable that can have different types during execution. By performing an *upcast* we convert a variable referring to an object of a derived class to one referring to its base class. It is the most common situation for using a polymorphic variable as we can ensure that a derived object can always be treated as one of its base class. On the other hand if we convert an object of base class to one of its derived

class we are performing a *downcast*. In this case we cannot ensure the correctness of the operation unless the variable had been previously upcasted. Assignment polymorphism is known as *pure polymorphism* if the variable is used as a parameter of an operation. It uses late binding.

§1.1.4 Why Objects anyway?

Up until this point basic concepts and a possible methodology have been defined but it is still not clear why a system designer would want to bother using the Object Oriented approach instead of relying on other paradigms like formal, structured, logical or functional programming. It is specially important to answer this question in the context of this thesis where an Object Oriented model is being proposed as a general framework.

A lot has been written about the benefits of Object Orientation and the reasons why nowadays has become so popular. This is a brief list that tries to illustrate the main advantages usually attributed to OO.

Objects map more directly to real world concepts. As it will be later discussed, although the world is not made of objects, the object-oriented paradigm is the most suitable way to understand systems that reside in it. Objects are suited to model concepts related to whatever domain under analysis and these objects are closer to the actual software solution. Therefore, object- orientation bridges the gap between the problem space and the solution. It also brings common terminology between developers and domain stakeholders.

Objects enhance *encapsulation*. Every class encapsulates a concept. By defining a class we separate between state and behavior: attributes and their values represent the object state while the operations in a class define the behavior of that family of objects. According to the encapsulation principle, attributes should not be accessible from outside the class. Internal state can only be modified by an object responding to a message. Published operations represent the object accessible behavior, but objects can also have non-accessible private behavior.

Objects improve *information hiding*. Encapsulation is in fact the first step towards a more restrictive and stronger form of structuring object-oriented systems: information hiding. In an object-oriented system we may choose not to publish some information for different reasons such as limiting the system complexity and make it more understandable or ensuring system integrity. We may choose to hide private behavior or operations but even more, we may decide to hide all methods or implementation details and just publish an interface for some classes. This interface may be enough for using the whole system functionality while not becoming overwhelmed by its internal details.

Objects promote good structuring. Software systems resulting from object-oriented analysis and design are better structured. Both encapsulation and information hiding, as already commented, enhance good structuring.

Objects favor re-use. All the above properties make object-oriented systems more reusable, not only in what code respects but also conceptually.

It is interesting to note that although all the benefits of object-orientation are better understood when following a complete object-oriented development process, it is also useful to use some of the OO techniques to just some parts of the process. In [Meequel et al., 1997] they explain an approach in which object-oriented analysis techniques are used although the resulting code is not programmed in an object-oriented language due to platform restrictions (embedded software).

§1.1.5 A note on efficiency: OO is not inefficient

Signal processing software applications have traditionally been developed under the structured programming paradigm and more precisely in C language. In many DSP applications, specially in embedded systems where the final code is in assembly language, the program efficiency both in execution speed and size is one of the most important factors. It is still not strange finding that some algorithms are implemented directly in assembler. All the benefits attributed to object-orientation are many times put aside with the intent of building something "fast" and "light".

In this context, it is of course difficult to convince signal processing engineers that object-oriented (and in fact any sort of code structuring that focuses on reusability and understandability) is worthwhile. In the course of the CLAM framework development (see chapter 3) we have had the opportunity to convince many different people that well-structured object-oriented code is not inefficient and can indeed yield more efficient and far more robust applications (see [de Champeaux et al., 1993], for instance).

At this point though the choice of an appropriate object-oriented programming language is very important. Typically languages such as Java, running on a virtual machine, cannot yield efficient code. It is also important to be able to manage low-level issues like memory allocation policies whenever necessary. The choice of C++ as the most appropriate object-oriented language comes naturally, even more when C++ is already a de facto standard for DSP applications and a natural follow-up of the C programming language. It may be argued that C++ is not a "true" object-oriented language, even its creator advertises it as a multiparadigm language [Stroustrup, 1995]. Nevertheless this language can be effectively be used to build a completely object-oriented framework, abandoning the paradigm only for

strictly necessary low-level issues such as hardware access or memory handling but hiding these details in such a way that the user does not even need to be aware they exist.

But the choice of an object-oriented language does not guarantee the quality of the resulting code. Many signal processing applications are in fact written in C++ in its "a better C" meaning, forgetting about all the advantages and tools offered by the object-oriented paradigm. A non-exhaustive list of common FUDs about object oriented efficiency, particularized to the case of the C++ language, are summarized in the following[2]:

(1) *Encapsulation is inefficient*: Some signal processing developers argue that the indirection introduced by adding a Set or Get operation to an attribute yields a less efficient executable. Because of this they consciously break the encapsulation principle by making all attributes public. The disadvantages of doing so are as important as the mixing-up of state and behavior, the existence of a non-homogeneous interface, or the lack of an implementation that can evolve independently from its interface. On the other hand it is definitely not true that the introduction of an operation call introduces inefficiency. In C++ an operation can be "inlined" so the call of an operation does not introduce a memory indirection. Although this is only feasible in methods with very little computation time and space requirements, this is exactly what we face when implementing a Getter or a Setter.

(2) *Modularity is inefficient*: Clearly a memory indirection introduces some overhead that at the end may result in a less efficient final application. But, as already mentioned, this is only so whenever inlining is not feasible. And inlining is not possible when the operation executed in the method is so complex that the time of its execution is several orders of magnitude greater than the time taken for the memory indirection. In this case it is also clear that the overhead of the indirection can be neglected and is by far surpassed by the benefits introduced, which will be later commented.

(3) *Inheritance is inefficient*: When declaring a base class with a virtual operation (needed in any case in order to implement polymorphism) all objects instance of any of its subclasses will have a virtual function pointer table. This produces memory inefficiency as objects will occupy more memory than the strictly necessary for storing their attributes. Furthermore, whenever a virtual operation is invoked on a pointer or reference, dynamic binding is introduced and therefore a new indirection that will add inefficiency. Also, virtual operations cannot be inlined. The solution is to treat inheritance with care and not introduce virtuality on any method that could be inlined (i.e. the cost of an indirection is comparable to the cost of the method itself). On the other hand,

[2]See a more exhaustive report on C++ performance issues at [O'Riordan, 2002].

inheritance, just as modularity, introduces far more benefits than inconveniences.

Although these previous and other related misbelieves can, as already commented, be minimized the main benefit of object-orientation can be better understood in the mid-term. When building a well-structured system we are not only worrying about short-term efficiency issues as the ones commented but we are also setting the grounds for further refactorings in order to improve overall efficiency. In a well-structured system it is much easier to detect efficiency bugs and treat them in a correct way. The clearer, more modular and well-structured the code is the easier it is to optimize.

We will better understand all these issues in the following example.

§1.1.5.1 The Vocal Processor Experience

The Vocal Processor was a research project developed in our group for the Yamaha Company, Japan. The Vocal Processor is a VST plugin for singing voice processing that implements spectral domain techniques. It is designed to run on real-time and, because of its complex algorithms, demands many computer resources.

The initial implementation was done in C++ but the code was highly unstructured and hardly maintainable. For that reason it was decided to port the code to the CLAM framework (see chapter 3). Once the VST plugin was running in its CLAM version, it was discovered that this implementation was almost one hundred percent slower. The first impression was that the Object-Oriented techniques and overall design of the framework were causing this and that it would not be possible to compete with the fine-tuned but highly unstructured original code. The process that followed, and demonstrated that other reasons were behind that bad performance, illustrates the overall message of this section.

None of the efficiency problems found in this application were related to any of the previously mentioned prejudices against object-orientation: encapsulation, modularity, or inheritance. As a matter of fact, having a clear and clean design and code enabled a fast refactoring that ended up in having a fully object-oriented CLAM version of the plugin that was even about thirty percent faster than the original one.

Using specialized profiling tools, the efficiency hotspots were found. These are the main actions that had to be taken in order to improve the first CLAM version:

(1) Algorithm improvement: some algorithms were not well implemented and contained efficiency bugs. These efficiency bugs were usually related to unnecessary memory allocations and independent loops.

(2) VST interface improvement: the incoming data from the VST host was not being correctly handled

and this meant having unnecessary memory allocations of large memory blocks.

(3) Inefficient low level routines: the Microsoft Windows implementation of some low level routines such as float to integer conversion or absolute number were causing an overall slow down of the process. Surprisingly, this was one of the most significant factors. Such functions were being called millions of times during the whole process and the overall effect was really worrying. We finally ended up implementing these routines in assembler code.

(4) Compiler settings. The VST plugin was being compiled for Windows with the Microsoft Visual Studio environment. This compiler has some obscure settings that had to be tuned in order to find the best combination.

(5) Incorrect thread handling.

From this use case it is clear that the object-oriented paradigm did not introduce any efficiency trap. Furthermore, it facilitated the improvement of the original code because of its modularity and clear structure.

§1.2 Models and Systems

The use of the word *metamodel* in this Thesis title has been carefully chosen and it is important to offer a complete definition as the concept itself is very much related to some of thesis defended in this work.

To understand the meaning of metamodel we must first understand what a *model* is. But the concept of model is also closely related to that of *system.* As a matter of fact, the definition of model that will be used throughout this work is the following: "A model is an abstract representation of a given system". It seems clear that we better clearly define what a system is before continuing the discussion on the model itself.

§1.2.1 Systems

The definition of system given the Institute of Electrical and Electronical Engineers professional group in System Science and Cybernetics (from [Rowe, 1965]): "a large collection of interacting functional units that together achieve a defined purpose". Or, in other words, a system is made up of three main components: a goal, a set of things and/or rules, and the way this things and/or rules are

organized or connected in between them. There is a whole corpus of System Engineering but systems are also studied from a psychological or even philosophical point of view.

In [DeGreene, 1970], the author defends that systems are studied to: improve the system or its successor, determine general theories for new system development, and advance science. He also mentions that there are several scientific methods related to systems such as generalization across systems, analysis and synthesis, and modeling and simulation. All of these methods are of course interrelated. When analyzing a system, the basic steps are: (1) recognition that a problem exists and the solution may be related to systems analysis techniques; (2) definition of the problem in a appropriate form; (3) definition of the system itself (iterative process that starts with a gross approximation and results in a conceptual model); (4) definition of performance criteria; (5) definition of alternative configurations and their evaluation; (6) presentation of alternatives and tradeoff results to the user; (7) performance of iterative analysis during development; (9) analysis of operational systems to gather performance data [DeGreene, 1970].

But one of the most commonly accepted definitions for "system" is surprisingly related to the software engineering corpus. In the article entitled "Definition of System" [Hall and Fagen, 1956] the authors define: "A system is a set of objects together with relationships between the objects and between their attributes." This definition, that is largely referenced and commented in the literature about system theory, was made in 1956, much before the term object orientation was even coined. So, according to Hall and Fagen, objects are simply the parts, or components of a system.

Attributes are the properties of objects. Relationships tie the system components together. A system cannot be considered as such if it does not have a purpose in itself. This assertion also implies that a system has properties, functions or purposes distinct from its constituent objects, relationships and attributes. Objects can be physical parts or abstract objects such as variables or equations. The relationships are those that tie the system together. The question of whether a relationship is important or trivial depends on the problem to be solved. And another interesting property of any system is that can be subdivided hierarchically into subsystems, sub-subsystems, components, units, parts and so forth. Any of this levels can be considered to be made up of objects as either a subsystem, component, unit or part is an object in itself.

If we accept Hall and Fagen's definition as approximately valid (note that the definition of system given by the IEEE is not conceptually very different) we can assert that the object-oriented paradigm is the best alternative when modeling a system. A system is in fact made of objects that relate in between them, we just have to identify them.

According to [DeGreene, 1970] different features characterize any system analysis: (1) emphasis

on an uncertain future; (2) tendency to oversimplify; (3) evaluation of alternatives; (4) aid to conceptualization; (5) decline pressures resisting analysis; (6) use mutually reinforcing techniques; and (7) selection of relevant variables. System analysis have a qualitative beginning, quantification becomes possible as greater detail and precision is introduced. As a matter of fact, and according to [Boulding, 1969], one possible approach to general systems theory is through the arrangement of theoretical systems and constructs in a hierarchy of complexity. In any case, and as we will see in next section, this is usually accomplished through the use of appropriate *models*.

The state of a system is the collection of variables necessary to describe a system at a given time. An event is an instantaneous occurrence that may change the state of the system. In general, systems are classified as discrete or continuous (see [Law and Kelton, 2000]). A discrete system is one for which the state variables change instantaneously at separated points of time. In a continuous system these variables change continuously. Some systems are neither completely discrete nor completely continuous: combined discrete-continuous simulation. There are three types of interactions between discrete and continuous variables:

- A discrete event may change the value of a continuous variable
- A discrete event may cause the relationship governing a continuous state variable to change at a particular time.
- A continuous state variable achieving a threshold may cause a discrete event to occur or to be scheduled.

§1.2.2 Models

Once we have a more clear view on what a system is we can come back to the definition of model, which we said is "an abstract representation of a system". The main goal when analyzing a system is to come up with a valid model for a given purpose. If it is possible to alter the actual system and evaluate it under the new conditions, it may be desirable to do so. But this is seldom the case. It is usually necessary to build a model and study it in place of the actual system.

A model consciously focuses on some of the subject matters leaving others out. Therefore a model needs not be complete. But incompleteness and a high degree of abstraction does not mean imprecision.

It is important to note that a given system can be represented by many different models. These models may have different level of abstraction and purpose and the "best" model is only the one that

is the most useful for a particular application or purpose[Hall and Fagen, 1956]. On the other hand a single model may have multiple interpretations, where the *interpretation* is defined as the relation of the model to the thing being modeled [Seidewitz, 2003]. Finally, the relation of all the possible models derived from a particular system is called a *theory*. More precisely, and according to [Anderson, 1983], a theory is a precise deductive system that is more general than a model. As a matter of fact, a model is the application of a theory to a specific phenomenon. We can deduce the quality of a theory by the quality of the models that it generates.

A first classification of models is into static and dynamic models. A static model is a representation of a system at a particular time. A dynamic model represents a system as it evolves over time.

Another way of classifying models is according to the domain in which the model will be deployed. We can then classify models into *physical* models, *mathematical* models and *software* models. Out of these we are interested in the latter but as software and mathematical models share many of their properties we will first summarize the main properties of a mathematical one.

A mathematical model is a set of mathematical expressions that are sufficient to explain the behavior of a given system for a particular purpose. Once a mathematical model is built, if it is simple enough, it may be possible to get an exact "analytical" solution. But this is almost impossible in many cases and then it is necessary to study the model by *simulation* (numerically exercising the model and studying how input variations affect outputs). If a simulation model does not contain any probabilistic (random) components, it is called *deterministic*: once the input and the internal relations are known, the outputs can be determined. Many systems though must be modeled as having one or more random inputs, these are called *stochastic* simulation models [Law and Kelton, 2000].

Software models built in order to have an executable software version of the system under study are called *constructive* or *executable* models [Hylands et al., 2003]. Constructive models define a computational procedure that mimics a set of properties of the system. Software models that are clearly different from the system are often referred to as simulations. But in many systems a model that starts off as a simulation ends up being a software implementation, blurring the distinction between the model and the system itself.

The use of executable models in software engineering has given place to a methodology known as *model-driven development* (MDD) [Ambler, 2003, Meller et al., 2003, Seidewitz, 2003]. In traditional or code-driven development models are treated as simple sketches that are thrown away once the code is done but in model-driven development the models themselves become the primary artifacts in the development in software. In MDD different models are use throughout the developing process and

are divided into two main categories: *conceptual* or *analysis* models that express requirements and *engineering* or *design* models that are ready to be turned into code or an executable. Successive model transformation offer a way of mapping analysis models to design models. Model-driven development is also very much related to the Meta Object Facility or MOF (see section 1.4.3)

Nevertheless it is probably in the Analysis phase of the development cycle when it is most used. We talk about the "domain model" to describe a high-level, abstract model that is used to partition the original domain of the application (e.g. "cattle growing" or "car traffic regulation") usually into objects. When building the domain model we do not intend to have a nearly implementable design, we are only trying to understand the context of the application, partitioning the problem to make it more understandable and easily translatable to the software domain. The domain model focuses on requirements (either functional or non-functional) that a particular domain comprises.

The domain model is many times combined with the "business model". While the domain model focused on requirements, this latter focuses on the internal processes involved in the business we are trying to model. It is somehow in between the domain model and the design model that will latter become the real application. Note though that, in any case, the model is neither the original problem, nor the solution given to the problem in the software application. Both the original problem and the final software implementation have an abstract representation through either a domain model or a design model. As M.A. Jackson points out, from the domain we get a description model true only of the domain, from the machine we get a description true only of the machine; but in the middle we have an intermediate description that is true of both domain and machine maybe under certain restrictions [Jackson, 1995].

Software models are usually expressed using the Unified Modeling Language (UML) but can also be expressed in any other programming language.

To summarize, we can say that a model is a representation of a system that has been built for one of the following purposes:

- Communication of ideas between people and machines
- Completeness checking
- Race condition analysis
- Test case generation
- Transformation into an executable

And observes the following properties:

- Under whatever restrictions specified, the model must yield the same output to an input than the original system.
- It has the shape or appearance of the original (it is an iconic representation).
- But it is always different from the system being modeled (the original) in scale, implementation or behavior under certain conditions.
- It can be manipulated or exercised in such a way that its behavior or properties can be used to predict the behavior or properties of the original (it is a simulation tool).

§1.2.3 Object-orientation, systems and models

And how does the object-oriented paradigm relate to all this? As it has already been pointed out, the concept of system is very much related to objects themselves. It does not surprise, bearing these idea in mind, that object-orientation was born as a "side effect" when designing a programming language for simulation (i.e. modeling a system). Kristan Nygaard, considered the father of object orientation, said when talking about the way that this paradigm was born: "The idea that led to Simula and OOP was to create a language that made it possible for people to comprehend, describe, and communicate about systems; to analyze existing and proposed systems through computer based models"Kristen[Nygaard, 2001].

It is surprising how many authors discuss that the "real" world is not made of objects but an object-oriented model is a good approximation [Graham, 1991]. What is missing in their discussion is what Nygaard and others mainly from the Scandinavian school realized : when building a software application we are in fact implementing a system model (the application itself is a system); the world may not made of objects but when building a software we are not trying to model the world as a whole but a particular system, present in a given domain of the real world. This idea of object-orientation emerging from system modeling will be further developed in the next section.

As already outlined in the previous section, it is an important hypothesis in this work that object-orientation is the best paradigm for describing systems through models. Although this assertion was directly derived from the definition of system itself it is also interesting to note that the birth of the object-oriented paradigm is intimately related to the idea of system modeling. For strengthening this idea, in this section we will take a look at the ideas of the first pioneers that formulated the foundations of object-orientation. To keep our discussion focused, we will concentrate in the ideas of Kristen Nygaard,

creator of the Simula languages, father of object-orientation and founder of the Scandinavian school of object-orientation.

§1.2.3.1 Kristen Nygaard and the Scandinavian School: Object-Orientation as a system simulation tool

Simula I is usually referred to as the first object-oriented language. Nevertheless, in Simula I the words *objects* and *classes* were never used. Instead objects were called *processes* and classes *activities* [Nygaard, 2001]. Simula I aimed at being a system description language and a programming language [Nygaard, 1986]. The original goals of the authors was to create a language that included definitions of basic concepts, allowed a formal description of any system, and in which system descriptions should be easy to read and useful for communication [Dahl and Nygaard, 1966]. The interest of converting Simula into a general purpose language, more than just a simulation language, and the idea of adding inheritance capabilities between *activities* brought up a new version of Simula called Simula 67 [Nygaard and Dahl, 1978].

By reading the (very few) articles that the main authors of the Simula language, Nygaard and Dahl, wrote it is clear that they were conscious of the definition of a system as a set of interacting objects and the role of object-orientation as a modeling or simulation paradigm. What follows can be read as a summary of the Scandinavian school of object-orientation extracted from [Nygaard and Dahl, 1978, Nygaard, 2001, Nygaard, 1986]and [Dahl and Nygaard, 1966]. In this description we will concentrate on the basic ideas that led to the formulation of the object-oriented paradigm but will also include some definitions of concepts that are useful for my dissertation.

According to Nygaard, the idea that led to Simula and object-oriented programming was to create a language that made it possible for people to comprehend, describe, and communicate about systems and to analyze existing and proposed systems through computer based models. The basic concept in Simula is the process, being characterized by a data structure and an operation rule. The individual members of the data structure of a process were called attributes. Simula I was mostly used as a system description language. Most of the time, it was used as a way to gather more information about the system under study and understand it better.

At first, Simula defined a system as containing active components (*stations*) and passive components (*costumers*). A costumer was generated by the service part of a station, transferred to the queue part of another... The whole structure was called a *network* and actions from stations were regarded as happening in discrete moments in time therefore becoming discrete event networks. Then the idea of network was substituted by the more powerful concept of *model*, consisting of processes that were

declared collectively by *activity* declarations.

Nygaard was not very keen on using the term *Computer Science*, he preferred to ad-scribe himself to the discipline of *Informatics*. He had a personal discourse about the meaning of the word and the discipline itself that is interesting for us to understand the whole idea behind object-orientation. Informatics is an applied or empirical science. While mathematics is about relations between specified quantities and operations valid in specified domains, Informatics is about processes and the description of their state transitions. Informatics should be defined like most other sciences (like physics, chemistry, botany, sociology or political science but unlike mathematics) as the study by scientific methods of a domain of phenomena and a perspective selecting a set of characteristics of those phenomena. And the particular perspective selected by Informatics is that of the information aspects or processes. So Informatics is not an abstract science, it is related to real-world phenomena.

If we follow this intellectual discussion and continue defining some of the terms and concepts that already appeared in the previous definition we will end up getting to the concept of model itself. We just mentioned that Informatics is related to real-world phenomena. But what is a phenomenon? A *phenomenon* is defined by Nygaard as a thing that has definite, individual existence in reality or the mind, anything real in itself. A *concept* is a generalized idea of a collection of phenomena, based on knowledge of common properties of instances in the collection. A concept is defined by its extension (the collection of phenomena that the concept covers), intension (the collection of properties that characterize the phenomena represented by the concept) and designation (the collection of names by which the concept is known). A phenomenon M is a *model* of a phenomenon R, according to some perspective P, if a person regards M and R as similar according to P. M is called the model, R the referent and P is the perspective.

In the first definition, we also mentioned the idea of process. We will now dive into it and discover its relation to systems. A *process* is a phenomenon that we choose to regard as a development of a part of the world through transformations during a time interval called its life span. The basic qualities of a process are: its substance, its states, its transitions, and its structure. A state of an information process is expressed by describing: - the moment at which the state is recorded, its substance, measurement of properties and transformations going on, all at that moment. The *structure* of the process is the limitations of its sets of states and thus of its possible sequences of states. A process is regarded as an information process when the qualities considered are: its substance, the physical matter that it transforms, the state of its substance, in terms of measurable properties, the results of measurements represented by values, its transitions, the transformations of its substance and thus of its state.

A *system* is a part of the world that is regarded as a whole, its substance consisting of components, each components state characterized by the states of properties, called attributes, that are selected as being relevant, and by state transitions relating to these attributes and to other components and their attributes.

In object oriented programming an information process is regarded as a system developing through transformations of its state. The substance of the process is organized as the system components, called objects. A measurable property of the substance is a property of an object. Transformations of state are regarded as actions by objects. The substance of the system is organized as objects, building the system's components. All attributes of the substance are properties of objects. Transitions - transformations of state - are regarded as being the result of actions of objects.

> "Q: But what is OO then suited to describe?
> A: Systems.
> Q: What is a system?
> A: A system is something that you have decided to regard as a system.
> Q: What does it imply to regard something as a system?
> A: You regard it as a whole, consisting of components, each component has properties and may interact with other components."

From [Nygaard, 2001]

As a conclusion, the basic ideas behind object orientation that make it different from other perspectives, is that: (1) A program execution is regarded as a physical model system simulating the behavior of either a real or imaginary part of the world; (2) physical modeling is based upon the conception of reality in terms of phenomena and concepts (3) a physical model system is constructed, modeling phenomena by objects and concepts by categories of objects or classes.

At this point it is also important to highlight another important figure: Alan Kay. Apart from also being considered one of the most important people in the birth of object-orientation, Alan Kay is also the father of modern graphical user interfaces and the laptop, the main creator of the Smalltalk programming language and the author of such famous sentences as "the best way to predict future is to invent it". In [Kay, 1993] he summarizes Smalltalk, and therefore object-orientation, in the following principles:

- Everything is an object
- Objects communicate by sending and receiving messages (in terms of objects)
- Objects have their own internal memory (in terms of objects)
- Every object is an instance of a class (which in turn must be an object)
- The class holds the shared behavior for its instances

- To evaluate a program list control is passed to the first object and the remainder is treated as its message.

What is particularly interesting for our discussion is the fact that at the time of formulating the basis of the object-oriented paradigm he was very much interested in system simulation after his work in the Air Force and the National Center for Atmospheric Research. Furthermore, he also acknowledges the deep influence that the Simula language had on his ideas and he explicitly mentions the relation between object-orientation and system modeling when stating that "object-oriented design is a successful attempt to qualitatively improve the efficiency of modeling the ever more complex dynamic systems and user relationships..." [Kay, 1993].

Other renowned authors have also highlighted this relation between object-orientation and simulation. Timothy Budd in [Budd, 1991] describes the existing relationship between object-oriented programming and discrete event simulation. He finishes the discussion stating that "computation is simulation".

§1.2.4 Metamodels

Sometimes when modeling a set of related systems, usually belonging to a given domain, we realize that these models share many constructs. We are then able to generalize across these different models and come up with a model of what the set of related models should conform to. This is what we call a "metamodel", a model of models. As a a matter of fact the term metamodel is still quite controversial and a matter of discussion (see [www-Metamodel,]). The concept has been especially used in relation to UML. The language itself is used to describe the syntactic rules that all models written in UML must adopt thus defining a metamodel: a model of models. Metamodeling and UML are very much related to another OMG standard: the Meta Object Facility or MOF

In many senses a metamodel is similar to a regular domain model. A domain model though usually has a much less ambitious scope as it models the domain of a given application but already bearing this application in mind. For this reason, important domain activities that do not directly relate to the system being modeled are left out of the domain model or just modeled at a very abstract level. On the other hand, when building a metamodel the restriction is not a particular system or application but rather a set of systems or applications that might be modeled according to this metamodel.

In the rest of this section we will reflect how metamodeling is understood and used in different communities related to software design.

In [Mili et al., 1995] they concluded that metamodeling results in "cognitive economy" as it replaces extensions (explicit occurrences of a given concept) by intensions (the definition of the concept itself). It is thus a mechanism very similar to regular classification: the relation between a model and its meta equivalent is that between an instance an a class. Three dimensions of metamodeling were also identified:

(1) Metamodeling as the modeling of a modeling/representation language.

(2) Metamodeling as the multiple instantiation levels of application knowledge.

(3) Metamodeling as the modeling of information about how to use and manipulate application models.

Thus, going "meta" may mean developing a computational model for the family of applications at hand, developing an architecture for simulating and executing such applications, or implementing such an architecture in the form of a development platform/framework.

Going along the first of the three dimensions previously identified we conclude that a kind of metamodel is the language in which a model can be expressed. And this is precisely the interpretation that the Object Management Group (OMG) of the ACM uses when describing UML as a metamodel. Even more, the OMG has defined a Meta Modeling Facility (MMF) in order to create a precise definition of UML in its 2.0 version [Reggio, 2002, OMG, 2003]. The semantics and syntax of UML itself can be defined using a subset of the language. Furthermore, UML and MOF (see section 1.4.3) can be used to describe an object-oriented model that can then be transformed into a different domain (e.g. Petri Nets or Dataflow Networks) by applying formal transformations [Varró and Patarizca, 2002].

Metamodeling is also sometimes understood as the definition of a semantic model for a family of related domain models. In this sense, metamodeling becomes very much related to Ontological Engineering. We define an ontology as being the explicit representation of domain concepts. An ontology is thus a system of concepts that defines the vocabulary of the problem domain and restricts the way that these terms may be combined to model the domain. Under this definition, the process of building an analysis domain model in a standard object-oriented methodology can be understood as metamodeling as long as the resulting model is precise and general enough as to be used to model different systems apart from the one under study. As pointed out in [Devedzic, 2002], the properties of a well designed ontology are the same than those of any software system including classes in an object-oriented design: decomposability, understandability, extensibility, maintainability, modularity and interfaceability. An ontology is a domain metamodel that will be used by the different models related to that particular domain. It conveys to a hierarchical representation where higher layers correspond to domain-independent

or core concepts and lower layers correspond to concepts only meaningful in a particular domain. The Java class library is, for instance, a good example of general usage ontology.

But in system engineering, metamodeling is understood in a quite different sense. A metamodel is here a simple approximation (usually mathematical) of a complex system model [Barton, 1994, Caughlin, 1997]. Here the model input-output behavior is seen as black-box and is approximated by a polynomial metamodel. The first thing to do in the metamodeling process is to clearly identify the purpose of the metamodel. After that, there is an iterative process consisting on [Caughlin, 1997]:

(1) Select an experimental design

(2) Run the simulation

(3) Collect Data

(4) Select a metamodel set

(5) Select identification methodology

(6) Generate the metamodel

The simpler form of a metamodel can reveal the behavior of a more complex model. Beside, the metamodel requires less computer resources and can be run intensively under controlled parameters to reveal what affects the system performance.

But if we accept the broader definition of Metamodel given when starting this section - a metamodel is just a model of a set of related models - we can conclude that most modeling engineering ends up related to metamodeling. When modeling a system we usually want to reuse our abstraction or modeling effort and try to come up with a model that is not only useful for the particular problem at hand but can be re-used in similar situations, that is a metamodel.

In this sense, metamodeling is very much related to the activity of *classifying classes*. When we identify classes from a set of existing object we are applying an abstraction process that is very similar to the one we apply once we have the classes and we try to group them together in different categories or meta-classes. This is usually accomplished by defining inheritance hierarchies.

The first levels of the inheritance hierarchy and the way these, usually abstract, classes relate define a metamodel that can be then instantiated by defining different concrete classes that give place to a new model. This is particularly so when defining a white-box framework (see section 1.3).

As a matter of fact, one of the main goals of this Thesis is to classify classes related to the digital signal processing domain. This classification will result in the metamodel that will be presented in chapter 4.

§1.2.5 Metaphors

Kent Beck is one of the most influential people in modern software engineering. He is the father of the eXtreme Programming methodology, the CRC cards and the first one to envision the use of software patterns (see section 1.3.6). One of the fundamental issues in XP is the importance of using metaphors.

In [Beck, 2002] the author informally defines the idea of a *metaphor* as well as a set of concepts that are related to them. For the purpose of this discussion it it interesting to take a look at some of them:

- *Metonymy* is to refer to a whole by a part
- A *simile* is an explicit comparison
- An *analogy* is an explicit comparison where the parts are connected
- And a *metaphor* is the linking two sets of concepts; The understanding itself of one set of concepts in terms of another

A metaphor is made of a source, a target and a mapping. According to Beck, what makes metaphors useful is "the game of is/is not", which helps thinking about the problem and understanding it. A metaphor is best understood if it is based in a "ground metaphor", that is it has a physical experience basis.

Metaphors are constantly used in software design. Icons, for instance, in a graphical user interface are based on metaphors (the pencil, the paint bucket or the sand clock). But is probably in the XP community where this concept has become especially important. A key point to the XP methodology is what they call *System Metaphor* which is a simple shared story of how the system works[Beck, 1999]. That is a story that everyone - costumers, programmers, a nd managers - can tell about how the system works. A proper system metaphor supports four basic aspects of system building: common vision, shared vocabulary, generativity and architecture. It is also sometimes seen as a valid substitution for what in most methodologies is called the system architecture [West, 2002].

Some authors go even beyond this use of metaphor and claim that in object-oriented design a program is a metaphor of the real world [Noble et al., 2002]. The authors argue that the objects used in the object-oriented program are not the same kind as real-world objects. Objects in the design are not "abstractions" of the real-world objects because real objects are not instances of the software ones, rather the relationship between software objects and external objects can be seen as a "metaphor".

But how does this concept relate to the ones previously introduced? It is our opinion that the use of the metaphor concept in software engineering is misleading. Most of the times when talking about a system metaphor what we are actually doing is to describe a *model*. Let us remember that a model does not need to either be formal or to completely describe a system under study and the correctness of a model is only measured in relation to how well it accomplishes its purpose. The purpose of a conceptual model can be, as already commented in section 1.2.2, to communicate ideas. This is basically the same idea as the *system metaphor*.

The only problem with using the metaphor as a driving force throughout the development is that it promotes the idea that the system is static in two ways. First, a real metaphor is usually difficult to find but even more difficult to change once adopted. Conceptual changes introduced later in the development may contradict one of the underlying ideas in the metaphor. Although the metaphor can then be updated, chances are that it will then become misleading. On the other hand, if we think on models we think on a continuous of different models that evolve from the most abstract and conceptual one we may have in the very initial phases to the executable model or code itself. This idea of the model being able to describe the system at different levels of abstraction is also lost when using the system metaphor paradigm.

§1.3 Frameworks

In next chapter we will present a framework for developing music and audio applications. It is well worth it to first understand all the techniques and concepts related to software framework development. It is also interesting to see how frameworks relate to previously defined concepts like systems or metamodels.

§1.3.1 Definitions

In a general sense we can define a framework as a general pool of constructs for understanding a domain that is not tightly enough organized to constitute a predictive theory. However, it is possible to add additional details so as to come up with a predictive theory from a framework. One judges a framework in terms of the success or fruitfulness of the theories it generates [Anderson, 1983].

The term framework is very much used in software engineering, especially in relation to object-oriented analysis and design. 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++[Weinand et al., 1989] and Interviews. It is interesting to note that most of the seminal frameworks were related to designing user interfaces. In the framework history, it is also important to cite the name of the Taligent company, a joint venture of Apple Computer, IBM and Hewlett-Packard. They developed a set of tools for rapid application development under the name of "Common Point" that consist on more than a hundred OO frameworks.

The most accepted definition for an object oriented framework is: "a framework is a set of classes that embodies an abstract design for solutions to a family of problems" [Johnson and Foote, 1988]. In the framework literature [Beck and Johnson, 1994, Taligent, 1994, Bosch et al., 1999] we can find similar definitions that can be summarized in the following: A framework is the reusable design of a system or a part of a system expressed as a set of abstract classes and the way the instances of those classes collaborate; a framework is a set of prefabricated software building blocks that programmers can use, extend, or customize for specific computing solutions; frameworks are large abstract applications in a particular domain that can be tailored for individual applications; a framework is a reusable software architecture comprising both design and code. As [Johnson, 1997] points out, although different, all definitions are correct because they focus on different views of a framework such as structural or functional.

§1.3.2 Properties of a well-designed framework

Bearing these definitions in mind, a number of properties arise from a well-designed framework.

A framework defines the behavior of a collection of objects. It provides ways of reusing analysis, design and code and includes a set of software building blocks that programmers can use, extend, or customize for specific computing solutions. These building blocks usually make up a library of subclasses.

A framework, though, is more than a well-written class library, it is an abstract specification of a kind of application. Libraries reuse code but little analysis or design. A class library does not enforce a particular design on an application, it just provides functionality that can help the application to do its job. Nevertheless, some libraries exhibit framework-like behavior and some frameworks can be used as class libraries. It can be seen as a continuum with traditional class libraries at one end and sophisticated frameworks at the other.

With frameworks, developers don't have to start from scratch every time they write a particular application. Frameworks help developers provide solutions for problem domains and better maintain those solutions. They also provide a well-designed and thought out infrastructure so that when new pieces are created and added, they can be integrated with minimal impact on other pieces of the

framework [Nelson, 1994].

No matter how elegant and well-designed a framework is, it won't be used unless the cost of understanding it and then using its abstractions is lower than the perceived cost of writing these functionalities from scratch, without using the framework [Booch, 1994a]. It is important to remember that Software is not reusable until it has been reused [Johnson, 1993].

To be successful, a framework must be complete, flexible, extensible, and understandable [Taligent, 1994]. A framework should also observe ways of minimizing potential client errors and enhance platform portability. It should be illustrated with examples. Examples make frameworks more concrete and make them easier to understand [Johnson, 1992]. As a matter of fact, a framework is just a generalization of preexisting examples in a particular application domain. An example of a framework is a use case.

Frameworks impose a model that the the developer must adapt to. But on the other hand frameworks improve developer productivity [Moser and Nierstrasz, 1996].

The goals of framework design should be to build applications from preexisting components, use a small number of types of components over and over and write as little code as possible (ultimate goal is no-code: build programs by direct manipulation) [Johnson, 1993]. With a framework it shouldn't take a good programmer to build a good program. Frameworks are designed by experts in that domain but can then be used by non-experts.

A framework usually plays the role of the main program in coordinating and sequencing application activity and flow control. In a framework, the main program is reused, and the developer decides what is plugged into it and makes new components that are plugged in. The developers code gets called by the framework code. The framework determine the overall structure and flow of control of the program [Johnson, 1997].

Frameworks are related to designing components. When designing a component we also have to trade simplicity for power. A component with many parameters can be used often but will be hard to learn [Johnson, 1997].

Finally, a framework reuses analysis. It describes the kinds of objects that are important and provides a vocabulary for talking about a problem. An expert in a particular framework sees the world in terms of the framework, and will naturally divide it into the same components. Two expert users of the same framework will find it easier to understand each others designs, since they will come up with similar components and will describe the systems they want to build in similar ways [Johnson, 1997].

§1.3.3 Classification of frameworks

Different views can be used for classifying software frameworks.

Taligent, for instance, elaborated different ways of classifying frameworks [Taligent, 1994]. On one hand, a framework can be classified as an *application framework*, a *domain framework* or a *support framework*. Application frameworks are those that intend to just offer support for developing one kind of software application or part of an application. An example could be a framework for developing graphical user interfaces. Support frameworks offer horizontal, system-level services like file access or support for distributed computing. Finally domain frameworks are those that give support to

Another Taligent classification separated frameworks into *architecture-driven* or *data-driven*. Architecture-driven frameworks rely on inheritance for customization while data-driven frameworks rely on object composition. Architecture-driven frameworks can be difficult to use because they require the client to write a lot of code but data-driven can result very limiting. Typically, a well-designed framework offers and architecture-driven base with a data-driven layer.

The most accepted classification, though, is probably that of Ralph E. Johnson according to which software frameworks can be basically classified into *white-box frameworks* and *black-box frameworks* [Johnson and Foote, 1988, Roberts and Johnson, 1996]. In a white-box framework, the user adds methods to existing super-classes. The implementation of the framework must be understood in order to use it. The framework relies heavily on the use of inheritance, classes are generalized from the individual existing applications. In a black-box framework, components are offered and must be interconnected in order to build an application. The user only needs to understand the interface or protocol of these components. Inheritance can be used to organize the component library but it has the drawback that it cannot be easily changed at run-time. For that reason, composition is used to combine components into applications. White-box frameworks have also been named "calling" while black-box are "called". With this simple black or white classification, it is clear that very few frameworks can be classified into completely white or black-box, most of them are laying in an intermediate grey.

Out of these two basic kinds of frameworks, a number of patterns are identified: "Component Library", "Hot Spots", "Pluggable Objects", "Fine-grained Objects", "Visual Builder", and "Language Tools". In a framework similar classes must be written for each new problem that arises, the Component Library pattern proposes to solve this by starting with a simple library of obvious classes and add the others as they are needed. Hot Spots are parts of code that need to be written over and over again for every new application to the framework. The solution is to separate code that changes from code that doesn't. Encapsulating varying code in objects, we will show the user where the code is expected

to change. Another observation when writing a framework is that most of the subclasses are trivial. To avoid boosting the number of trivial classes the Pluggable Objects pattern proposes to parameterize classes with the messages to send, blocks to access or whatever distinguishes one trivial subclass from the other. On the other hand, the Fine-grained Objects patterns tries to answer the question of how far a framework designer should go in dividing objects into smaller ones. The answer is to continue dividing objects until it makes no sense in the domain of that particular framework. The rationale behind this idea is that domain experts, which are the targeted users for a framework, are better at understanding a complex system in their domain than at understanding complex programming. The last two patterns in the framework catalogue deal with what should be the aimed final objective in framework design. The Visual Builder pattern arises from the observation that in a black-box framework, when connecting objects the connection script is very similar from one application to another. The proposed solution is to create a graphical program that allows to generate an application by graphically connecting objects. Finally, the Language Tools pattern tries to solve the problem of how to inspect and debug complex composite objects created in the visual builder. The answer is to create specialized tools adapted to the domain of the framework.

§1.3.4 The framework development process

There are quite a few documented methodologies for developing a framework. Most of them, though, admit the fact that the objective is to identify abstractions with a bottom-up approach: start by examining existing solutions and be able to generalize from them. When adopting a framework development process it is important to take into account that any framework will usually start as a white-box framework and ideally evolve into a black-box one. It is also important to understand that developing a framework is more difficult than developing and application because users must be able to understand many design decisions. For this reason it is even more important to follow good design practices and principles.

In [Johnson, 1993] the author explains the difference between the "ideal" and the "good" way to develop frameworks and software in general. The ideal consists on three basic phases. First analyze the problem domain, learning abstractions and collecting examples of programs to be built. Then design abstractions that can be specialized to cover all the examples and derive a theory to explain these applications. And finally test the framework by using it to solve the examples. This ideal is impossible to follow thoroughly for two main reasons: it is very difficult and time consuming to do an exhaustive domain analysis, analyzing all the possible existing examples; and old applications work so there is no

financial incentive to convert them to use new software. So a less-than-ideal-but-good way to develop a framework is to first pick two applications in the domain that are supposed to be solved using the framework; make sure that at least some of the developers in the team have developed applications for that particular domain; and divide project into three groups: a framework group, which both gives and takes software, considers how other applications would reuse the framework and develops documentation and training; and two application groups that try to reuse as much software in the framework as possible and complain about how hard it is to reuse.

In [Bosch et al., 1999] the authors give a different (though complementary) overview of the framework development process. First they distinguish two different activities in framework development: *core framework design* and *framework internal increments*. Core framework design comprises both abstract and concrete classes in the domain. The concrete classes are intended to be invisible and transparent to the end user (e.g. a basic storage utility) while the abstract classes are either intended to be invisible or to be used through subclassing. On the other hand framework internal increments build additional classes that form a number of class libraries. They capture common implementations of the core framework design and they can be either subclasses representing common realizations of the concepts captured by the superclasses or a collection of classes representing the specification for a complete instantiation of the framework in a particular context. A final application built from the framework consists of some core framework designs, the framework internal increments and an "application-specific increment".

In this model, the main phases in the development are: (1) a framework development phase that is usually the most effort-consuming phase and is aimed at producing a reusable design in a particular domain; (2) a framework usage or instantiation phase where applications are developed; and (3) a framework evolution and maintenance phase.

A number of activities can be identified in the first development phase, namely domain analysis, architectural design, framework design, framework implementation, framework testing an application testing. When determining the domain scope there is a problem of choosing the correct size: if the framework is too large, many specialists maybe needed in the team and the process may become long and expensive; if the framework is too narrow it may have to be tailored for every new application that comes up. Since the framework may be used in many, sometimes unknown ways, it may simply not be feasible to test all aspects of the framework. Since the framework relies on some parts implemented by the user it may be impossible to test completely before its release.

In the framework usage phase, the main activities involved are: domain analysis, architectural design, framework design, framework implementation, framework testing an application testing. In this

phase an important question is to be faced. If an error in the framework is found when developing an application: who is going to fix the error in the framework? and, should the applications that are working and are apparently not affected by this error upgrade to the latest release of the framework?

It is also interesting to take a look at how a company such as the Swiss Ericsson models the framework development process. Ericsson Software Technology Frameworks have a template methodology that is instantiated with every new framework design and that can be summarized in the following guidelines (see [Landis and Niklasson, 1995]): A list of requirements on at least two applications should be provided together with a list of requirements on the framework. A list of future requirements should also be provided. The project team should include members with knowledge of each application and a member with knowledge on framework design. Information should be gathered from as many sources as possible. Requirements and use cases should be separated into framework-specific and application-specific and they should also be divided into functional and non-functional. High-level abstractions should be identified, preparing for the identification of the framework. But only abstractions that are in the framework domain should be introduced and afterwards they should be named the same in the static model. Existing solutions should be examined and large frameworks should be structured into sub-frameworks. A static model for each application should be developed then introducing abstractions common to several applications into the framework. Using graphical notations the project should be presented clearly to all project members. Subsystems should have high cohesion and little coupling. Finally existing frameworks should be studied trying to reuse as much design as possible.

And according to the previously mentioned Taligent, the process for developing frameworks must observe four important guidelines that can be understood as a summary of the previous list [Adair, 1995]:

- Derive frameworks from existing problems and solutions
- Develop small, focused frameworks.
- Build frameworks using an iterative process driven by client participation and prototyping.
- Treat frameworks as products by providing documentation and support, and by planning for distribution and maintenance.

And related to the framework development process it is finally interesting to note that as Opdyke points out [Opdyke, 1992, Opdyke and Johnson, 1990] one of the main characteristics of a framework is that it is designed to be refined, good frameworks are usually the result of many design iterations and a lot of work involving sometimes structural changes.

§1.3.5 Frameworks generate metamodels

From what has been explained up until now it is clear that when building an application framework we are generalizing across a set of systems that usually belong to a particular domain. We aim at offering the tools and the conceptual infrastructure needed to implement all those systems. A framework is not just about reuse of code but also about conceptual reuse. A well designed framework should present a precise model of computation and conceptual basis.

This "conceptual framework" is in fact a metamodel that can be instantiated as a model to implement a desired application or domain service.

In [Meequel et al., 1997] the authors explain a method for designing a framework. The method is based on a domain analysis. Although they recognize that the framework development process is iterative by nature, their method proposes the following phases for framework design:

(1) domain analysis: analyze products in the domain, commonalities and variations.

(2) architecture: high-level partitioning of the software system into component.

(3) subsystem/component design

It is our belief that this traditional view of "analysis-first" process models is seldom appropriate for framework design (some authors argue that this approach is not appropriate for any kind of software design process). When designing a framework it is obvious that some initial domain analysis must be performed in order to understand basic requirements and different viewpoints represented by different stakeholders. But we cannot aim at understanding and modeling the whole domain from the start, it is important to remember that when building a framework we are not only implementing an infrastructure for existing applications but also thinking about future development in the domain.

We believe in an application-driven approach as that presented by [Johnson, 1993] in which the framework development process is iterative and where the user feedback is necessary on a regular basis. Such a process model has proven useful in our case. Not only is not necessary to have a previous domain analysis model before starting the framework design, it is the framework design that ends up generating an appropriate domain metamodel. Therefore, and already commented in section 1.2.4, frameworks generate metamodels.

§1.3.6 Patterns

A *pattern* is defined as a rule that expresses a relation between a *problem*, a *context* and a *solution* [Gamma et al., 1995]. Software patterns represent simple solutions that have proven successful and that can be applied to any of the activities in the life cycle such as analysis or design. They capture experience in a consistent and methodical way so that it can be easily reused and learned. Patterns do not include actual code. As a matter of fact, a framework is sometimes defined as a set of patterns plus code [Johnson, 1997]. According to [Beck and Johnson, 1994], patterns can be used to derive an architecture from an initial problem where an architecture is understood as the way the parts work together to make the whole [Garlan and Shaw, 1993]. Patterns are not metaphorical but rather metonymycal as, in some sense, you substitute some attribute or cause or effect of the thing for the thing itself [Noble et al., 2002].

The main difference between patterns and frameworks is that patterns represent design reuse while frameworks, apart from design, they also represent analysis and implementation reuse. Nevertheless, because some frameworks such as the Model View Controller have been implemented a number of times they have also become some sort of pattern [Johnson, 1997].

§1.4 Metadata and Meta Objects

Metadata are data about data. We use metadata to describe the content represented by a given data. In order to do so, metadata can describe the actual data or add contextual information. In this Thesis we will introduce different uses and applications of metadata. In chapter 3 we will see how an audio processing framework uses and implements metadata support. In chapter 5 we will also introduce a metamodel for content transmission that is very much related with metadata.

A clear example of the use of metadata is the information attached to an audio file. In a regular audio file we may easily find two kinds of information: *contextual* such as Author, Title, or the Album Title; and *data description* such as File Format, Sampling Rate, Bit rate or Number of Channels.

Out of the many possible languages that can be used for annotating metadata, XML is currently gaining more and more acceptance. It is the language chosen in this thesis for all metadata applications.

§1.4.1 XML and XML Schema

XML [www-XML,] is a text based format to represent hierarchical data. XML uses named tags enclosed between angle brackets to mark the begin and the end of the hierarchical organizers, the

XML elements. Elements contains other elements, attributes and plain content. Let's see a sample XML document:

```
<?XML version='1.0' encoding='ISO-8859-15' ?>
<mainElement>
      <subelement1 attribute1="attribute Content">
            plain content here
            <subsubelement>plain content</subsubelement>
            plain content here
      </subelement1>
      <subelement2 attribute2="attribute Content">
            <subsubelement>plain content</subsubelement>
            <subsubelement>plain content</subsubelement>
            <subsubelement>plain content</subsubelement>
      </subelement2>
      <emptyelement attribute="foo" />
</mainElement>
```

Both attributes and plain content are simple text data. The main different between them is that an attribute is named and plain content is not. Elements also have a name. Names for attributes must be unique inside its hierarchic context, though this restriction doesn't apply to elements' name.

The power of XML is that you can adapt your own tags (elements) and tag attributes (attributes) in order to describe your own data. This is one of the reasons why XML is starting to spread rapidly. The XML specification defines the concepts of well-formedness and validity. We say that an XML document is well-formed if it has a correct nesting of tags. In order for a document to be valid, it must conform to some constraints expressed in its document type definition (DTD) or its associated XML Schema.

A DTD file defines the constraints and structure of a set of XML files but by using a completely different syntax from XML. Here is an example of a basic DTD file:

```
<!DOCTYPE letter[
<!ELEMENT letter (header, text)>
<!ELEMENT header (name,address)>
<!ELEMENT address (street, city)>
<!ELEMENT name #PCDATA>
<!ELEMENT street #PCDATA>
<!ELEMENT city #PCDATA>
<!ELEMENT text #PCDATA>
]>
```

The presented DTD defines: Nesting rules, e.g., a letter is composed of a header and a text. Very simple datatypes, e.g., a name is a PCDATA (parsed character data). Two kinds of items can be defined in a DTD: elements and attributes. Basically elements are tags (name between angle brackets), while attributes are parameters of elements.

On the other hand XML-Schema is a definition language for describing the structure of an XML document using the same XML syntax and it is bound to replace the existing DTD language. It is thus a tagged textual format but it also includes support for most Object Oriented

concepts[www-XMLSchema,]. The purpose of a schema is to define a class of XML documents by using particular constructs to constrain their structure: datatypes, elements and their content, attributes and their values. Schema can be seen as an extended DTD. More important for many purposes is that schemas may be written in XML. Using XML as the document format for schemas allows users to employ standard XML tools instead of specialized applications. See the following example[3]:

```
<xsd:schema xmlns:xsd="http://www.w3.org/1999/XMLSchema">
  <xsd:element name="MetriXInstrumentDefinition" type="MetriXInstrumentDefinitionType"/>
  <xsd:complexType name="MetriXInstrumentDefinitionType">
    <sequence>
      <xsd:element name="Generators" type="GeneratorsType"/>
      <xsd:element name="TimbreSpace" type="TimbreSpaceType"/>
      <xsd:element name="ParamBPFArray" type="BPFArrayType" minOccurs="0"/>
      <xsd:element name="TimeBPFArray" type="BPFArrayType" minOccurs="0"/>
      <xsd:element name="LowLevelParamArray" type="LowLevelParamArrayType"/>
      <xsd:element name="HighLevelParamArray" type="HighLevelParamArrayType" minOccurs="0"/>
    </sequence>
  </xsd:complexType>
</xsd:schema>
```

It is important to note that there are already quite a few multimedia applications of the XML language. See for example [www-XMLMusic,] for a list of music-related XML applications. But probably the most ambitious project for using XML metadata is that of MPEG-7, the multimedia description standard, which will be described in the next section.

§1.4.2 MPEG-7

The Moving Picture Experts Group (MPEG) is a working group of ISO/IEC in charge of the development of international standards for compression, decompression, processing, and coded representation of moving pictures, audio and their combination.

So far MPEG has produced (or is about to produce) the following standards:

- MPEG-1, the standard for storage and retrieval of moving pictures and audio on storage media (approved Nov. 1992), includes in its layer 3 the famous mp3 format.
- MPEG-2, the standard for digital television (approved Nov. 1994), includes most of the formats now used in DVD's.
- MPEG-4, the standard for multimedia applications (first version approved in 1998) includes DiVX, AAC (Advanced Audio Coding) and Structured Audio.
- MPEG-7 the content representation standard for multimedia information search, filtering, management and processing (approved in 2002).

[3]Abridged version from the MetriXML instrument definition presented in section 6.4.4.

- MPEG-21, the multimedia framework (still to be approved).

We will now focus in describing the main features of the MPEG-7 standard not only as an example of metadata usage but also due to its relation with different parts of this thesis and also because of our involvement in its development process [Peeters et al., 2000].

The MPEG-7 standard [Manjunath et al., 2002, Martínez, 2002] also known as "Multimedia Content Description Interface" aims at providing standardized core technologies allowing description of audiovisual data content in multimedia environments. In order to achieve this broad goal, MPEG-7 has standardized:

(1) Descriptors (D) or representations of Features, that define the syntax and the semantics of each feature representation,

(2) Description Schemes (DS), that specify the structure and semantics of the relationships between their components, which may be both Ds and DSs,

(3) A Description Definition Language (DDL), to allow the creation of new DSs and, possibly, Ds and to allows the extension and modification of existing DSs,

(4) System tools, to support multiplexing of description, synchronization issues, transmission mechanisms, file format, etc.

The MPEG-7 standard consists of the following parts, under the general title Information Technology - Multimedia Content Description Interface:

- Part 1: *Systems.* Architecture of the standard, tools that are needed to prepare MPEG-7 Descriptions for efficient transport and storage, and to allow synchronization between content and descriptions. Also tools related to managing and protecting intellectual property.
- Part 2: *Description Definition Language* (DDL). Language for defining new DSs and perhaps eventually also for new Ds, binary representation of DDL expressions.
- Part 3: *Visual.* Visual elements (Ds and DSs).
- Part 4: *Audio.* Audio elements (Ds and DSs).
- Part 5: *Multimedia Description Schemes.* Elements (Ds and DSs) that are generic, i.e. neither purely visual nor purely audio.
- Part 6: *Reference Software.* Software implementation of relevant parts of the MPEG-7 Standard.

- Part 7: *Conformance*. Guidelines and procedures for testing conformance of MPEG-7 implementations.

The following terminology plays a major role in the understanding of the MPEG-7 process:

Data: Data is audio-visual information that will be described using MPEG-7, regardless of storage, coding, display, transmission, medium, or technology.

Feature: A Feature is a distinctive characteristic of the data which signifies something to somebody.

Descriptor: A Descriptor (D) is a representation of a Feature. A Descriptor defines the syntax and the semantics of the Feature representation.

Descriptor Value: A Descriptor Value is an instantiation of a Descriptor for a given data set (or subset thereof).

Description Scheme: A Description Scheme (DS) specifies the structure and semantics of the relationships between its components, which may be both Descriptors and Description Schemes.

Description: A Description consists of a DS (structure) and the set of Descriptor Values (instantiations) that describe the Data.

Coded Description: A Coded Description is a Description that has been encoded to fulfil relevant requirements such as compression efficiency, error resilience, random access, etc.

Description Definition Language: The Description Definition Language (DDL) is a language that allows the creation of new Description Schemes and, possibly, Descriptors. It also allows the extension and modification of existing Description Schemes.

The main tools used to implement MPEG-7 descriptions are the Description Definition Language (DDL), Description Schemes (DSs), and Descriptors (Ds). Descriptors bind a feature to a set of values. Description Schemes are models of the multimedia objects and of the universes that they represent e.g. the data model of the description. They specify the types of the descriptors that can be used in a given description, and the relationships between these descriptors or between other Description Schemes.

In this context the DDL defines the syntactic rules to express descriptions schemes and their interpretation. MPEG-7's DDL is based on an extension of W3s XML-Schema. However, the DDL is not a modeling language such as UML. The DDL is rather to be used to represent the result of modeling, i.e. DS and Ds. Here is an example of an MPEG-7 definition of an image:

```
<Creation>
  <Title type="original">
    <TitleText xml:lang="es">
      Telediario (segunda edición)
    </TitleText>
```

```
    <TitleImage>
      <MediaURL>file://images/teledario_ori.jpg</MediaURL>
    </TitleImage>
  </Title>
  <Title type="alternative">
    <TitleText xml:lang="es">
      Noticias de la tarde
    </TitleText>
    <TitleImage>
      <MediaURL>file://images/teledario_alt.jpg</MediaURL>
    </TitleImage>
  </Title>
  <Title type="alternative">
    <TitleText xml:lang="en">
      Afternoon news
    </TitleText>
    <TitleImage>
      <MediaURL>file://images/teledario_en.jpg</MediaURL>
    </TitleImage>
  </Title>
  <Creator>
    <role>presenter</role>
    <GivenName>Ana</GivenName>
    <FamilyName>Blanco</FamilyName>
  </Creator>
  <CreationDate>
    <D>16</D>
    <M>6</M>
    <Y>1998</Y>
  </CreationDate>
  <CreationLocation>
    <PlaceName xml:lang="es">Piruli</PlaceName>
    <Country>es</Country>
    <AdministrativeUnit>Madrid</AdministrativeUnit>
  </CreationLocation>
</Creation>
```

The Experimental Model (XM) software is the simulation platform for the MPEG-7 descriptors (Ds), description schemes (DSs), coding schemes (CSs), and description definition language (DDL). Besides the normative components, the simulation platform needs also some non-normative components, essentially to execute some procedural code to be executed on the data structures. The data structures and the procedural code together form the applications. The XM applications are divided in two types: the server applications and the client applications.

The server applications are used to extract the descriptor data from the media data. The extracted descriptor data is coded and written to an MPEG-7 bit stream. The client application performs the search in the MPEG-7 coded database, by computing the distance between the query descriptor and all reference descriptor of the database. Therefore, one descriptor, the query descriptor, is extracted in the same way as in the server application except that the coding is not performed. The reference descriptors are all extracted from the MPEG-7 bit stream.

The truth is that in its current state the Experimental Model is far from useful due to its complexity, lack of completeness and mixed approaches. It is our opinion that more efforts should be put into it. As a matter of fact, a standard such as MPEG7 would benefit from being an extensible software framework instead of just a set of tools and definitions.

§1.4.3 The Object Management Group's Meta Object Facility

The Meta Object Facility (MOF) is another standard from the ACM's Object Management Group (OMG) just as UML. MOF is defined like an standard that "provides a metadata management framework, and a set of services to enable the development and interoperability of model and metadata systems" [Adaptive et al., 2003].

Therefore MOF can be viewed both as a metadata interchange framework or as a metamodeling framework. Although MOF has not been formally used in this Thesis it is interesting to note the many similarities exist between this standard and the solutions provided in CLAM (see chapter 3) or the conceptual metamodel presented in the Object-Oriented Content Transmission Metamodel (see chapter 5).

MOF is platform and implementation independent and can be mapped to different languages such as XML or Java.

MOF defines separate concerns or capabilities for reuse by other models and metamodels. These concerns are organized into packages and the main packages addressing modeling and metadata management concerns are: Reflection, Identity and Extension.

The **Reflection** package extends a model with the ability to be self-describing, that is it provides all the necessary infrastructure to use an object without prior knowledge of its specific features. In MOF an object's class (i.e. its meta object) reveals the nature, kind and features of the object.

The Reflection package includes the *Object* class, which is the the superclass of all model elements. This abstract class has the following main operations:

- *getMetaClass()*: returns the class that describes this object
- *container()*: returns the parent container of the object if any
- *equals(element)*: returns true if the element and the 'this' reference the same instance.
- *get(property)*: returns the value of the given property
- *set(property, element)*: sets the element to the given property of the object.
- *isSet(property)*: true if value is different from the default value (if multiplicity is >1 is true if there is at least 1 element).
- *unset(property)*: sets the value to default

The other important class in the Reflection package is the *Factory* class, which is in charge of creating MOF instances and offers the following interface:

- *createFromString(datatype, string)*: creates an Element from the value of the String with a format defined by the XML Schema Simple Type corresponding to that data type.
- *convertToString(datatype,element)*: creates a string representation of the Element
- *create(metaClass)*: creates an object with all properties unset that is an instance of the metaclass.

The **Identity** package provides an extension for uniquely identifying metamodel objects without relying on model data subject to change. The main concept in this package is the *identifier*. The identifier of an object is a formal representation of the object identity (see section 1.1.1), it distinguishes a given object from all the rest. Identifiers allow the serialization of references to external objects, can be used to coordinate data updates where there has been replication, and can provide clear identification in communication. Identifiers also facilitate Model-Driven Development by providing an identifier immutable to model transformations (see section 1.2.2 for a brief summary on Model-Driven Development).

The Identity package provides the concept of *extents*. An Extent is a context in which an object can be identified. An object may be the member of zero or more extents. An Extent is not an object, it is part of a MOF capability. It has the following operations:

- useContainment(): returns true if it includes all objects contained by members of the objects.
- objects(): returns a ReflectiveSequence of the objects directly referenced by this extent.

Identity extends the Basic::Package with a URI that can be used for externally identifying it. It also extends the Basic::Property with the ability to designate a property as an identifier for the containing object.

An *URIExtent* is an Extent that provides URI identity. It has the following operations:

- contextURI(): returns a string specifying the URI established for the context of the extent.
- uri(object): returns the uri of a given object in the extent.
- object(string): returns the object identified by a given uri in an extent.

Finally, the **Extension** package offers a simple way of extending model elements with name/value pairs. MOF offers the ability to define metamodel elements like classes with properties and operations. But sometimes it is necessary to extend model elements with additional information such as information missing from the model or data required for a particular application.

The Extension package includes the *Tag* class. A Tag is a single piece of information that can be associated with model elements. It has the following properties:

- name: string
- value: string (MOF places no meaning on the value)

§1.5 Graphical Models of Computation

Models of Computation (MoC's) are abstract representations of a family of related computer-based systems. Although the word "model" will be kept for consistency with existing literature, it is clear that following the definitions given in the previous sections, the word "metamodel" would be much more appropriate. A MoC offers a particular abstract vision with a particular purpose that can be instantiated to model many different systems. Using the proper model of computation improves the development process and yields a better understanding of the system under design and its properties. Selecting the appropriate model of computation depends on the purpose but the choice is also generally conditioned by the application domain: DSP applications, for instance, will generally benefit from Dataflow models while control-intensive application mostly use Finite State Machine or similar models.

General design paradigms such as object-oriented, functional or logic may be interpreted as being models of computations. Nevertheless, such paradigms are still too abstract and generic to be really useful as a model of computation. As a matter of fact, a single paradigm may be used to model different models of computations and a single model of computation may be well consist on applying different general purpose paradigms. After having decided to use the object-oriented paradigm, much work in this thesis will be on finding the most appropriate Model of Computation it is therefore important to first understand what are the ones most related to our domain of interest.

Most useful models of computation belong to the category of "graphical MoC's". By *graphical* we are expressing the fact that according to the model, the system being modeled can be explicitly specified by a graph, being a graph a general mathematical construct formed by "arcs" and "nodes". Many MoC's are obtained by assigning a concrete semantic to arcs and nodes and by restricting the general structure of the graph (see figure 1.2). A non-exhaustive list of graphical MoC's includes: Queueing Models, Finite State Machines, State Charts, Petri Nets, Process Networks and Dataflow Networks[4].

Although the rest of this section will give a more detailed view of the models of interest, we will now briefly give a description of each of these models in order to have a first general overview:

[4]Other graphical MoC's including Component Interaction (CI), Communicating Sequential Processes (CSP), Continuous Time (CT), Discrete Events (DE), Distributed Discrete Events (DDE), Discrete Time (DT), Synchronous Reactive (SR), and Timed Multitasking (TM) can be found in the context of the Ptolemy project (see [Hylands et al., 2003]).

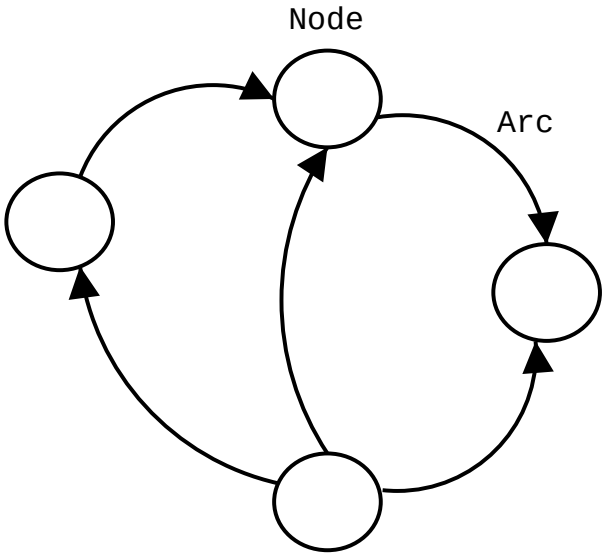

Figure 1.2: Graph

- Queueing Models is a graphical model where nodes represent complex operators such as Poisson queues or decision nodes and arcs represent events/tokens/requests. They are used for performance estimation, like determining the total latency of a network of queueing nodes.

- Finite State Machines are especially useful for specifying sequential control in control-intensive tasks and protocols. An FSM graph is made up of *inputs*, *outputs*, *states*, *initial state*, *next state* and *outs*. The model is not Turing complete and it renders models that are more suitable for formal analysis.

- State Charts is a graphical model consisting on *states*, *events*, *conditions* and *actions*. Events and conditions cause transitions and there are AND and OR compositions of states.

- Petri Nets is a graphical model for describing and studying systems with concurrent, asynchronous, distributed, non-deterministic, and parallel characteristics. It consists of *places*, *transitions* and *arcs* that connect them. A Petri net is executed by the firing rules that transmit the marks or tokens from one place to another and such firing is enabled just when each input place has a token inside.

- Process Networks or Kahn Process Networks is a concurrent model of computation that is a superset of data flow models. As a directed graph, each arc represents a FIFO queue for communication and each node represents an independent, concurrent process.

- Dataflow Networks is a special case of Process Networks where nodes are actors that respond to *firing rules*.

We will now describe the last three of these models for their relation to the CLAM network model.

§1.5.1 A brief catalogue of Graphical MOC's

§1.5.1.1 Petri Nets

Petri Nets is a graphical model of computation introduced by C.A. Petri in his PhD "Communication with Automata"[Petri, 1962] . Petri Nets are used for describing and studying information processing systems that are characterized as being concurrent, asynchronous, distributed, parallel, nondeterministic and/or stochastic. It is an asynchronous model that describes graphically and explicitly: sequencing/casualty, conflict/non-deterministic choice, and concurrency and has been applied

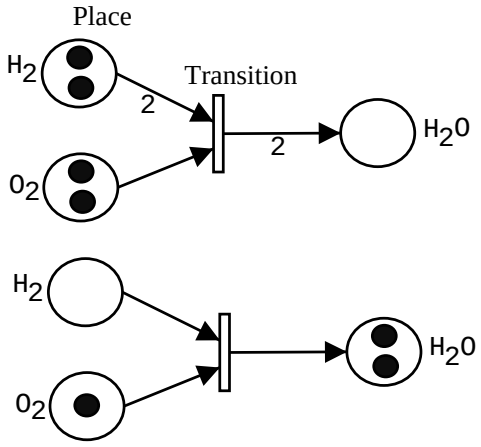

Figure 1.3: Petri net representation of water composition

to different areas such as distributed computing, manufacturing, control, communication networks or transportation.

A Petri Net is a particular case of directed graph with an initial state called the *initial marking*. There are two kinds of nodes: *places* and *transitions*. Arcs are either from a place to a transition or a transition to a place. Therefore Petri Nets are formed by a six tuple N=(P, T, A, w, x0) where P is a finite set of places and T is a finite set of transitions, A is a set of arcs, w is a weight function and x0 is an initial marking vector.

In modeling, places represent conditions and transitions represent events. A marking (state) assigns to each place a nonnegative integer. If a marking assigns a place p with a nonnegative integer k we say that "p is marked with k tokens". A transition (event) has a certain number of input an output places representing the preconditions and postconditions of the event. The presence of a token in a place is interpreted as holding the truth condition related to that place. In other words, k tokens are put in a place to indicate that k data items or resources are available. Typical modeling correspondences of input places - transitions - output places are:

- preconditions-event-postconditions
- input data-computation step-output data
- input signal-signal process-output signal
- resources needed-task or job-resources released
- buffers-processor-buffers

Figure 1.3 illustrates a simple Petri Net. Note that places are represented by circles, transitions are bars

or boxes, arcs are arrows labeled with weights and tokens are black dots. A marking (state) assigns to each place a nonnegative integer. If a marking assigns a place p with a nonnegative integer k we say that "p is marked with k tokens" and we place k black dots inside the circle.

As pointed out by [Murata, 1989], "there is only one rule to learn about Petri net theory: the rule for transition enabling and firing". A state or marking in a Petri Net is changed according to the following transition (firing) rule:

- A transition t is enabled if each input place p is marked with at least w(p,t) tokens, where w(p,t) is the weight of the arc from p to t.
- An enabled transition may or may not fire, depending on whether the event actually takes place.
- A firing of an enabled transition t removes w(p,t) tokens from each input place p of t and adds w(p,t) tokens to each output place p of t.

A transition with no input places is called a "source" transition and one without output places is called a "sink" transition. Source transitions are always enabled while the firing of a sink transition does not produce any tokens. A "self-loop" occurs when a place p is ate same time input and output to a transition t. A Petri Net without any self loop is called "Pure Petri Net". A Petri Net is said to be ordinary if the weight of all its arcs is 1. A finite capacity Petri Net is that in which there is a maximum of tokens defined for each place. For such Petri Nets there is an additional firing rule that says that after firing a transition, the number of tokens in its output places must not exceed their maximum.

Petri Nets is a very general model. Finite State Machines, Process Networks and Dataflow Networks are all subclasses of Petri Nets. As a mathematical model, it is possible to set up state equations, algebraic equations and other models governing the behavior of the system. Due to its generality and permissiveness, the model can be applied to any area or system that can be described graphically. But the more general the model is, the more complex it is. A major weakness of Petri Nets is its complexity. The problem may become unsolvable even for modest sized systems. That's why special modifications or restrictions need to be added suited to the particular application or domain.

Another of the problems of basic Petri Nets, its lack of composition and scalability, is addressed by a modification of the model called Higher-level Petri Nets[Janneck and Esser, 2002]. In a higher-level Petri Net, a Petri Net can appear wherever a data object can appear: as a token, as a parameter, or as the value of a computation. Tokens may carry arbitrary data objects as well as functions and petri nets. The main addition to the basic Petri Net model is the "Component". A component has parameters, can be instantiated and connected to the environment through ports. Input ports may be connected to places and output ports to transitions. To be able to connect components, places in a Petri Net are also

granted ports and they become "container places". When a transition produces a token, that token is not put onto the container place itself but rather on its input port.

The basic problem with Petri Nets, though, is that they are so simple that even a small system needs many states and transitions. This problem is known as the "state explosion". Because of this, different approaches to high-level and higher-order Petri Nets modelling have been attempted although with irregular results (see [Janneck and Esser, 2002]).

But one of those approaches that have been quite successful is of particular interest to our study: *Object-Oriented Petri Nets*. As a matter of fact, as Bastide points out in [Bastide, 1995] the relation between Object-Orientation and Petri Nets can be observed in two different ways: "Petri Nets inside Objects" and "Objects inside Petri Nets". In the first case Petri Nets may be used to model the inner state of an object where transitions in the net model the execution of a method in the object. The second case is much more common, as a matter of fact Object-Oriented concepts such as abstraction, encapsulation and inheritance have been used since the 1980's to build "layered" Petri Nets as opposite to "flat" ones. The basic idea in this approach is to increase the information available in the tokens considering that they are instances of a class. But a transition does not have to create or destroy objects, it can just move objects from one place to another. In OO Petri Nets the net models the control structure while the tokens model the data structure of the system. Finally, it is interesting to note that many different languages have been created to integrate OO concepts into Petri Net modeling, see for instance OOPN in [Niu et al., 2004].

§1.5.1.2 Kahn Process Networks

Process Networks is a Model of Computation (MoC) that was originally developed for modeling distributed systems but has proven its convenience for modeling signal processing systems. Process Networks are also called Kahn Process Networks after G. Kahn who first introduced them in his thesis (see [Kahn, 1974]). It is a natural model for describing signal processing systems where infinite streams of data are incrementally transformed by processes executing in sequence or parallel. Nevertheless, and as pointed out by [Lee and Park, 1995], this model of computation does not require multitasking or parallelism and usually neither infinite queues; it is in fact usually more efficient than comparable methods in functional languages. Process Networks have found many applications in modeling embedded systems as it is typical for embedded systems to be designed to operate infinitely with limited resources.

Commercial systems like SPW from Alta Group of Cadence, COSSAP from Synopsys, the DSP Station from Mentor Graphics, Hypersignal from Hyperception or Simulink by Mathworks and research software like Khoros from the University of New Mexico and Ptolemy from the Univ. of California at

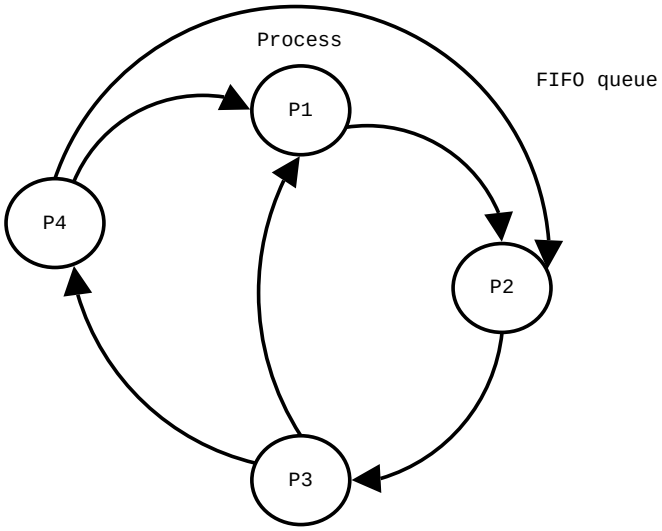

Figure 1.4: A Kahn Process Network

Berkeley are all based on variants of the Process Network model. Departing from the original Process Networks by Kahn, a number of more specific models have been derived. In this section we will give an introduction to the basic Kahn's Process Networks and will leave the more specific models for next sections.

Process Networks are directed graphs where nodes represent *Processes* and arcs are infinite FIFO queues that connect these processes (see figure 1.4). Writing to a channel is non blocking (it always succeeds immediately) but reading is blocking. If a process tries to read from an empty input it is suspended until it has enough input data and the execution context is switched to another process or level. A process may not "test" channel for presence of data. At any given point a process is either "enabled" or "blocked" waiting for data on one of its channels. It can not be waiting for data on one or another input channel.

In Kahn Process Networks processes produce tokens (data elements) that are sent along a communication channel and consumed by the destination process. Communication channels are the only way processes may exchange information. KPN systems are determinate: the history of tokens produced/consumed does not depend on execution order. A *stream* is a finite or infinite sequence of data elements or tokens X=[x1,x2...] where xi is a particular token. A process is in this sense a functional mapping from input to output streams.

As pointed out by [Parks, 1995] a parallelism can be made with Turing Machines: a Process Network can be seen as a set of Turing machines connected with one-way tapes, each machine operating on its own tape. Because of the "halting problem" we cannot tell in a finite time whether an arbitrary Turing machine program will halt, and the same is true for Process Networks. Two properties of Process Networks are related to this problem: *termination* and *boundness*. These properties are undecidable in finite time for the general case but, under some restrictions, we can study and classify PN before execution.

According to [Webb et al., 1999] interesting properties of process networks that make them a suitable model for computation are:

- Each process is a sequential program that consumes tokens from its inputs queues and produces tokens to the output queues.
- Each queue has one source and one destination.
- The network has no global data
- Each process is blocked when it tries to read from a a channel with insufficient data. The process resumes when there is enough data again.

- Although writing is generally non-blocking, blocking can be used to be able to use bounded queues.
- Concurrency can be implemented safely.
- Scheduling is transparent to the user
- Hierarchy or scalability.

§1.5.1.3 Dataflow Networks

Dataflow Networks is a graphical MoC very closely related to Process Networks. In this model arcs also represent FIFO queues. But now the nodes of the graph, instead of representing processes, represent *actors*. Instead of responding to the simple the blocking-read semantics of Process Networks, actors use *firing rules* that specify how many tokens must be available on every input for the actor to fire (see figure 1.5). When an actor fires, it consumes a finite number of tokens and produces also a finite number of output tokens. A process can be formed by repeated firings of a dataflow actor.

An actor may have more than one firing rule. The evaluation of the firing rules is sequential in the sense that rules are sequentially evaluated until at least one of them is satisfied. Thus an actor can only fire if one or more than one of its firing rules are satisfied. In general, though, synchronous dataflow actors have a single firing rule of the same kind: a number of tokens that must be available at each of the inputs. For example, an adder with two inputs has a single firing rule saying that each input must at least have one token.

As pointed out by [Parks, 1995] breaking down processes into smaller units such as dataflow actors firings, makes efficient implementations possible. Restricting the type of dataflow actors to those that have a predictable consumption and production pattern makes it possible to perform static, off-line analysis to bound the memory.

In Dataflow Networks instead of suspending a process on blocking read or non-blocking write, processes are freely interleaved by a scheduler that determines the sequence of actor firings. The biggest advantage is that the cost of process suspension and resumption is avoided[Lee and Park, 1995].

In many signal processing applications the firing sequence can be determined statically at compile time. The class of dataflow process networks where this is possible are called "synchronous dataflow networks" and will be commented in next section.

Dataflow graphs have data-driven semantics. The availability of operands enables the operator and hence sequencing constraints follow only from data availability. This feature has its limitations. The principal strength of dataflow networks is that they do not over-specify an algorithm by imposing unnecessary sequencing constraints between operators [Buck and Lee, 1994].

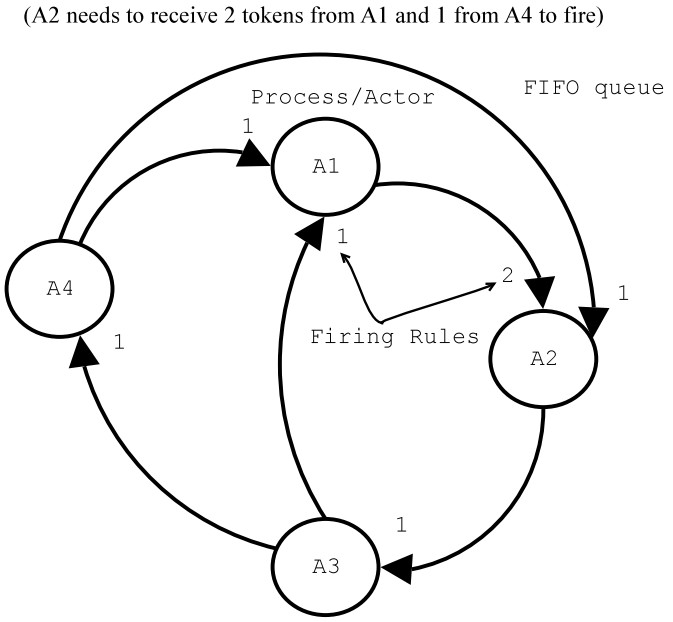

Figure 1.5: Dataflow Process Network

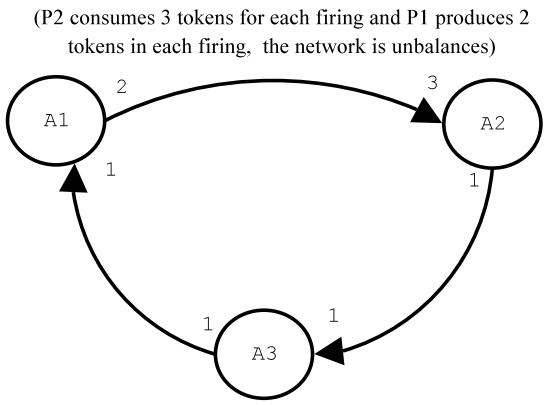

Figure 1.6: A Synchronous Dataflow Process Network

§1.5.1.4 Synchronous Dataflow Networks

Synchronous Dataflow Networks (SDF) is a special case of Dataflow Networks in which the number of tokens consumed and produced by an actor is known before the execution begins. The same behavior repeats in a particular actor every time it is fired. Arcs can have initial tokens. Every initial token represents an offset between the token produced and the token consumed at the other end. It is a unit delay and is represented by a diamond in the middle of the arc. Figure 1.6 illustrates a Synchronous Dataflow Network.

Schedule can be performed statically. As the execution of the graph is going to be repeated the compiler should just construct one "complete cycle" of the periodic schedule. A "complete cycle" is defined as the sequence of actor firings that returns the graph to its original state. From the static information of the network we can construct a "topology matrix" that contains relations between produced/consumed tokens in every arc. The element ij is defined as the number of tokens produced on the ith arc by the jth actor. Although it is only a partial info because there is no information on the number of initial tokens on each arc we can use the matrix to build the static schedule. For doing so we must find the smallest integer vector that satisfies the equation matrix*vector=0. It must be noted though that in complex networks these equations may not have a solution.

§1.5.1.5 Boolean Dataflow Networks

Although SDF is adequate for representing large parts of systems it is rarely enough for representing an entire program. A more general model is needed to represent data-dependent iteration, conditionals and recursion. We can generalize synchronous dataflow to allow conditional, data-dependent execution and still use the balance equations. Boolean Dataflow Networks (BDF) is an extension of Synchronous Dataflow that allows conditional token consumption and production.

By adding two simple control actors like switch and select we can build conditional constructs like if-then-else and do-while loops. The switch actor gets a control token and then copies a token from the input to the appropriate output, determined by the boolean value of the control token. The select actor gets a control token and then copies a token from the appropriate input, determined by the boolean value of the control token, to the output. These actors are not SDF because the number of produced/consumed tokens is not fixed and depends on an input boolean control [Buck and Lee, 1994].

§1.5.1.6 Dynamic Dataflow Networks

Dynamic Dataflow Networks are a Boolean Dataflow Networks with one additional variation: the control actors mentioned in the BDF model can now read multiple token values and the data actors can be fired conditionally based on the control actors read. Although dynamic scheduling might also be used for any of the previous models, it is a must for this model as production/consumption rates may vary during execution.

Dynamic scheduling can be classified as *data-driven* (*eager* execution), *demand-driven* (*lazy* execution) or a combination of the two. In eager execution a process is activated as soon as it has enough data as required by any of its firing rules. In lazy execution a process is activated only if the consumer process does not have enough data tokens. When using bounded scheduling (see [Parks, 1995]) three rules must be applied: (a) a process is suspended when trying to read from an empty input, (b) a process is suspended when trying to write onto a full queue and (c) on artificial deadlock, increase the capacity of the smallest full queue until its producer can fire.

§1.5.1.7 Computation Graphs

Computation graphs are a model of parallel computation similar to Process Networks. It is represented by a finite graph with a set of nodes, each associated with a function, a set of arcs , where a branch is a queue of data directed from one node to another. Four non-negative integers (A, U, W and T) are associated with each arc. 'A' is the number of tokens initially present in the arc, U is the number of tokens produced by the function associated with the node, W is the number of tokens consumed by the function associated with the node, T is a threshold that specifies the number of tokens that must be present in the arc before the function can be fired (obviously, T>=W).

Questions of termination and boundness are solvable for Computation Graphs, which turn out to be a restricted version of PN. It is interesting to note that Synchronous Dataflow Networks is a special case of Computation Graphs where T=W for all arcs.

§1.5.1.8 Context-Aware Process Networks

A special kind of Process Network introduced as an extension to the basic model but that is interesting for our purposes is that of Context-aware Process Networks [van Dijk et al., 2002]. This new model emerges from the addition of asynchronous coordination to basic Kahn Process Networks so process can immediately respond to changes in their context. This situation is very common in embedded systems.

In Context-aware Process Networks, stream oriented communication of data is done through regular FIFO channels but context information is sent through unidirectional register links (REG). These links have destructive and replicative behavior: writing to a full register overwrites the previous value and reading from a register returns the last value regardless if it has been read before or not. Thus, register links are an event-driven asynchronous mechanism. As a consequence, the behavior of a CAPN depends on the applied schedule or context.

A simple example of a system that can be effectively modeled by a context- aware network is a transmitter/receiver scheme in which the receiver needs to send information about its consumption rate to the transmitter so transmission speed can be optimized. The basic transmitter/receiver scheme can be implemented with a Kahn Process Network but in order to implement feedback coordination we need to use the register link provided by context-aware process networks.

Context-aware systems are indeterminate by nature. Unless the indeterminate behavior can be isolated, a composition of indeterminate components becomes a non-deterministic system, which is possible but not practical. Nevertheless as mentioned in [van Dijk et al., 2002] some techniques can be used in order to oraclise indetermination.

§1.5.2 A New Paradigm?

Due to their many practical applications, strong formalisms and distinct vocabulary some authors (see [Strom, 1986, McReynolds et al., 1999], for example) defend that Process Networks and related models can be seen as a sign of a new and underlying paradigm: the Process paradigm. Although the process paradigm is usually seen as a vehicle for the implementation of applications involving parallel or distributed computing it has also been formulated with a broader scope.

In the process paradigm, each process is a center of activity that holds internal state. The program executing that process is the only entity that can modify that state. Other processes may access that process indirectly through a message channel. A message channel is formed by connecting plugs named "input ports" to sockets "output ports"

All variables (and ports) are typed. A process communicates with another issuing a call on two operands: an output port and a parameter list or call-message. The call message is transmitted to a queue associated with the input port connected with the output port. The caller then waits for the call-message to be returned. The process owning the input port issues a receive operation to dequeue the message, it interprets the message and performs some action that may involve modifying the local state and the call message and finally issues a return operation to return the call-message.

Each process is a serial computation which can only schedule one activity at a time, and will only respond to a message after explicitly dequeuing it - this is in contrast to the object-oriented paradigm, in which a method is immediately invoked when a message is sent. There are no shared variables, global state or global time. Processes do not have type, only their ports. Each process may support different interfaces and give different services to different "users". In the process paradigm, a type determines only the interface, not the internal structure. Processes are active and explicitly schedule the receipt of messages.

According to [Strom, 1986] the more important practical difference is that OO puts emphasis in global structure and the process paradigm in that all data is local. In the process-oriented world there is no superuser, that is no-one has to be in charge of managing the overall system.

Furthermore, and related especially to dataflow models, some authors talk about *actor-oriented* programming as another new and different paradigm (see [Hewit, 1977, Agha, 1986, Liu et al., 2004] or [Hylands et al., 2003]).

In the actor-oriented paradigm, components called actors execute and communicate between them in a model. Actors, like objects, have a well-defined interface that abstracts internal state and behavior and restricts how an actor interacts with its environment. This interface includes ports that represent points of communication and parameters that are used to configure the operation of the actor. Often but not always parameters are part of a priori configuration and do not change upon execution.

In actor-oriented design communication channels are very important. Instead of transferring control by method call like in the OO paradigm, actors interact by sending messages through channels. Actors interact only with channels, not with other actors.

An actor is an encapsulation of parameterized actions performed on input data to produce output data. Actors may be stateless or stateful. Input and output data is communicated through well defined ports. Ports and parameters are the interface of an actor. A port does not follow the call-return semantics of OO.

It is our opinion that neither process orientation nor actor orientation represent a paradigm beyond object orientation but rather a particular subset or instance. As already highlighted in section

1.2.3 object orientation was born from some ideas that were closely related to process orientation and this is documented in the seminal works by Nygaard. Nevertheless, object-orientation grew way beyond these initial ideas becoming valid for describing any kind of software system.

§1.5.3 Patterns of Graphical MOC's

There have been several attempts to write patterns about graphical Models of Computation [Buschman et al., 1996, Shaw, 1996, Meunier, 1995, Edwards, 1995] including some specialized for particular domains [Posnak et al., 1996].

But probably the most complete catalogue is that of Dragos-Anton Manulescu who gives a quite complete overview of software patterns applied to the data flow model [Manolescu, 1997]. In this article the author formally defines the following patterns: Data flow architecture, Payloads, Module data protocol, and Out-of-band/in-band partitions. It must be noted that these patterns are not the result of any theoretical approach to dataflow architectures but rather the result of an exhaustive analysis of existing solutions. Therefore they represent a practical translation of the model requirements into the software domain.

In the next sections we will now briefly summarize each of these patterns.

§1.5.3.1 Data flow architecture

A variety of applications apply a series of transformations to a data stream. The architectures emphasize data flow and control flow is not represented explicitly. They consist of a set of *modules* that interconnect forming a new module or *network*. The modules are self-contained entities that perform generic operations that can be used in a variety of contexts. A module is a computational unit while a network is an operational unit. The application functionality is determined by: types of modules and interconnections between modules. The application could also be required to adapt dynamically to new requirements.

In this context, sometimes a high-performance toolkit applicable to a wide range of problems is required. The application may need to adapt dynamically or at run-time. In complex applications it is not possible to construct a set of components that cover all potential combinations. The loose coupling associated with the black-box paradigm usually has performance penalties: generic context-free efficient algorithms are difficult to obtain. Software modules could have different incompatible interfaces, share state, or need global variables.

The Solution is to highlight the data flow such that the application's architecture can be seen as a network of modules. Inter-module communication is done by passing messages (sometimes called

tokens) through unidirectional input and output ports (replacing direct calls). Depending on the number and types of ports, modules can be classified into *sources* (only have output ports and interface with an input device), *sinks* (only have input ports and interface with output devices), and *filters* (have both input and output ports).

Because any of the component depends only on the upstream modules it is possible to change output connections at run-time. For two modules to be connected the output port of the upstream module and the input port of the downstream module must be plug-compatible. Having more than one data type means that some modules perform specialized processing. Filters that do not have internal state could be replaced while the system is running. The network usually triggers recomputations whenever a filter output changes.

In a network, adjacent performance-critical modules could be regarded as a larger filter and replaced with an optimized version, using the *Adaptive Pipeline* pattern [Posnak et al., 1996] which trades flexibility for performance. Modules that use static composition cannot be dynamically configured.

§1.5.3.2 Payloads

In dataflow-oriented software systems separate components need to exchange information either by sending messages (payloads) through a communication channel or with direct calls. If it is restricted to message passing, payloads will encapsulate all kinds of information but components need a way to distinguish the type as well as other message attributes such as asynchronousity, priority... Some overhead is associated with every message transfer. Depending on the the kind of communication, the mechanism must be optimized.

Payloads give a solution to this problem. Payloads are self-identifying, dynamically typed objects such that the type of information can be easily identified. Payloads have two components: a descriptor component and a data component. In the case where different components are on different machines, payloads need to offer serialization in order to be transmitted over the channel.

Payload copying should be avoided as much as possible using references whenever possible. If the fan out is larger than one, the payload has to be cloned. In order to reduce copies even in that case, the cloned copies can be references of the same entities and only perform the actual copy if a downstream receiver has to modify its input. If it is not possible to avoid copying there are two possibilities: shallow copy (copy just the descriptor and share the data component) and deep copy (copy the data component as well maybe implementing copy-on-write).

The greatest disadvantage of the payload pattern compared to direct call is its inefficiency, associated with the message passing mechanism. One way to minimize it is by grouping different

messages and sending them in a single package.

A consequence of this pattern is that new message types can be added without having to modify existing entities. If a component receives an unknown token, it just passes it downstream.

§1.5.3.3 Module data protocol

Collaborating modules pass data-blocks (payloads) but depending on the application, the requirements for these payloads could be very different: some may need asynchronous user events, some may have different priority levels, some may contain large amounts of data. On the other hand, sometimes the receiving module operates at a slower rate than the transmitter, to avoid data loss the receiver must be able to determine the flow control.

Besides, we must take into account a number of possible problems. Large payloads make buffering very difficult. Payloads with time-sensitive data have to be transferred in such a way that no deadlines are violated. Asynchronous or priorized events are sent from one module to another- Shared resources for inter-module communication might not be available or the synchronization overhead not acceptable. And flow control has to be determined by receiving module.

There are three basic ways to assign flow control among modules that exchange Payloads:

- Pull (functional): The downstream module requests information from the upstream module with a method call that returns the values as result. This mechanism can be implemented via a sequential protocol, may be multithreaded and may process in-place. The receiving module determines flow control. It is applicable in systems where the sender operates faster than the receiver. This mechanism cannot deal with asynchronous or priorized events.

- *Push* (event driven): The upstream module issues a message whenever new values are available. The mechanism can be implemented: as procedure calls containing new data as arguments; as non-returning point to point messages or broadcast; as priorized interrupts; or as continuation-style program jumps. Usually the sending module does not know whether the receiver is ready or not. To prevent data loss the receiver can have a queue. If there are asynchronous or high-priority events, the queue must let them pass, else a simple FIFO queue can do.

- *Indirect* (shared resources): Requires a shared repository accessible to both modules. When the sender is ready to pass a payload to the receiver, it writes in the shared repository. When ready to process, the receiver takes a payload from the repository. The sender and the receiver can process at different rates. If not all the payloads are required by the receiver, the upstream module can overwrite data.

It must be noted though that having more than one input port complicates flow control and requires additional policies.

§1.5.3.4 Out-of-band and in-band partitions

An interactive application has a dual functionality: first it interfaces with the user handling event-driven programming associated with the user interface and the response times have to be in the order of hundreds of milliseconds; second it handles the data processing according to the domain requirements

User actions are non-deterministic so user interface code has to cover many possibilities. Data processing has strict requirements and the sequence of operations (algorithm) is known before hand. Human users require response in the order of hundreds of milliseconds but applications emphasize performance that is irrelevant for the user interface. Generally, a large fraction of the running time is spent waiting for user input. The user interface code and data processing code are part of the same application and they collaborate with each other.

The solution is to organize the application into two different partitions:

- Out-of-band partition: typically responsible for user interaction.
- In-band partition: it contains the code that performs data processing. This partition does not take into account any aspects of user interaction

§1.6 Summary and Conclusions

In this chapter we have seen the various building blocks upon which the thesis is built. We will now put them all into context, relating them and pointing out the consequences of our domain analysis. We will first briefly summarize the most important ideas introduced in the different sections.

In section 1.1 we have introduced the most important concepts related to the object-oriented paradigm. An *object* is a real-world or abstract entity made up of an identity, a state, and a behavior. A *class* is an abstraction of a set of objects that have the same behavior and represent the same kind of instances. The object-oriented paradigm can be deployed in the different phases of a software life-cycle and the *UML* language supports most of the activities contained in them. Apart from the concepts of object and class and the different kinds of relationships that can be established between objects and classes, other concepts such as *encapsulation*, *inheritance hierarchies* or *polymorphism* are important for fully understanding the object-oriented paradigm. The object-oriented paradigm presents many different

advantages that can be summarized in the following: it maps more directly to real world concepts, it enhances *encapsulation*, it improves *information hiding*, it promotes good structuring and it favors reuse. Finally, it is important to note that, although it is often considered otherwise, object-orientation does not imply less efficient code and furthermore its techniques make it even easier to end up having more efficient and robust final results.

In the next section, 1.2, we have defined the main concepts related to *models* and *systems*. The most commonly accepted definition of a system is that by Hall and Fagen in which they define a system as "a set of objects together with relationships between the objects and between their attributes." On the other hand, a model is an abstract representation of a system with a well-defined purpose, different models may exist for a single system. It is also important to note that the birth of object-orientation is very much related to the study of system simulations by Kristan Nygaard. Finally we define a metamodel as a "model of models", that is an abstract model that can be used to model a collection of related models.

In section 1.3 we have addressed the issue of software framework development. Although many different definitions can be given for a software framework it is that of Ralph E. Johnson in [Johnson and Foote, 1988] the one that is usually cited. According to this definition,"a framework is a set of classes that embodies an abstract design for solutions to a family of problems". Frameworks offer a way to reuse analysis, design and code. Frameworks can be classified, among other ways, into *white-box* and *black-box*. In white-box frameworks users extend previously existing classes, particularizing for their specific needs. On the other hand, black-box frameworks offer ready-to-use components that can be used as building blocks for an application. Although different approaches may be used for developing a software framework, it is usually recommended to use an application-driven methodology, using a limited amount of already existing applications as the driving force and favoring user-feedback as much as possible. Finally, a well-designed software framework can become a sort of metamodel in itself as it will offer a model of models for a given domain.

Metadata is defined as "data about data". In section 1.4 we introduce the most important concepts and tools related to metadata and our domain of object-orientation and multimedia signal analysis. XML is a general-purpose tagged language that is rapidly becoming the standard for metadata annotation of any sort. Using this same language, MPEG-7 is an ISO proposed standard for multimedia annotation. On the other hand the Object Management Group of the ACM has also proposed the MOF (Meta Object Facility) standard as a metadata management framework for object-oriented systems and technologies.

Graphical Models of Computation are abstract representations of a family of related computer-

based systems that use a graph-based representation as the primary way of communicating information about the system. There are many different graphical MoC's, each of them particularly well-suited for some purpose. The most important are outlined in section 1.5. In the context of signal processing applications, Kahn Process Networks and related models such as Dataflow Networks are of particular importance. Although some authors defend that these models should be seen as instances of the Process-Oriented paradigm we defend the thesis that process-orientation or actor-orientation is not more than a particular instance of the object-oriented paradigm.

As a conclusion we must point out that this present chapter has presented the most important foundational issues upon which we build our thesis. We have related concepts and ideas coming from disciplines such as software engineering, signal processing and system analysis. Although some of the issues here presented may seem trivial to a reader familiar with a particular technology we believe that the way that subjects are presented and related is not trivial in any way. Furthermore, it is important that the reader of the next chapter has a clear view of how and why different basic concepts are used.

It is in this sense important to note that although object-orientation is the obvious driving force that ties together the different hypothesis and parts of the work there other concepts presented in this chapter are not less important. We use object-orientation for describing systems through particular models and therefore it is important to have a clear view of what both systems and models mean and how they are used. It is also in this sense that graphical Models of Computation and software patterns associated to them play an important role in our work.

Software frameworks also play a central part in this work. The whole metamodel for digital signal processing, central to this Thesis and presented in chapter 4, was abstracted from (or rather in) the design of the CLAM framework for audio and music. It is also a thesis of this work that software frameworks, when well-designed and sufficiently generically constructed, generate domain metamodels, where a metamodel can be defined as a model of models or a classification of classes. Finally metadata is an important concept also related to metamodels. In different parts of this thesis, and particularly in chapter 5, we will see ways of using metadata.

Chapter 2

Environments for Audio and Music Processing

The metamodel presented in this Thesis is the result of our work in developing the CLAM framework, which will be presented in next chapter. As already mentioned, developing a framework or any other kind of environment needs an activity of, either implicitly or explicitly, building a model for a particular application or a domain metamodel that will then be instantiated through different models.

As it will be detailed in next chapter, CLAM was not developed as an intellectual exercise in order to find a good metamodel. Rather it evolved iteratively to user requirements until the metamodel emerged. Because of this development process not much attention was payed to already existing environments, only a first brief study was enough to determine that no existing environment offered what was about to be developed.

Nevertheless, once the metamodel is stable enough, it is important to carefully review all the different existing environments in order to extract commonalities and highlight whether CLAM and its metamodel represent an innovative approach. In this chapter we will review a great number of audio and music processing environments, many of them with different initial requirements and even different focus. Due to space limitation it is not possible to give a thorough review of each environment, instead we intend to give a brief conceptual introduction and provide references for further information.

§2.1 Introduction. A Classification of Audio and Music Processing Environments

As already mentioned the great number of existing environments for audio and music processing depart from different requirements and many focus on different aspects of audio and music. Before describing these different environments we will give a brief, and therefore incomplete, classification.

Until now, we have intentionally been using the word "environment" instead of "framework". The reason is that we will only use the word framework when talking about an application or domain framework in the sense described in section 1.3. The environments included in this present chapter range from full-fledged application frameworks such as Open Sound World (see 2.3.2.2) or SndObj (see 2.3.3.2) to simple visual applications with some extension capabilities such as WaveWarp (2.5.3). Some environments (see Supercollider in section 2.6.1.1) even go a step further and offer a complete programming language. This difference offers us a first classifying criteria, with extremely flexible application frameworks or programming languages on one end and simple applications on the other. Optimizing flexibility often compromises other features, namely usability or understandability, so a compromise needs to be taken.

Another interesting question that is particularly important for music and audio environments is whether a particular environment is focused on analysis, synthesis or both. Audio and music analysis has quite different requirements from synthesis and it is often difficult to match both. (As a simple example, in audio synthesis real-time performance is a big issue while in analysis application, the most important issue is usually how to organize and effectively store the analysis results).

In figure 2.1 we illustrate a temptative graphical classification map of the different environments reviewed in this chapter. We distinguish environments that have already been discontinued in dotted lines. On the other hand, the most important environments are highlighted with a larger font size. Note that although special care has been put on the position of each environment in some particular cases the fact that one environment is above or below another one, especially in overcrowded clusters, is meaningless and responds to purely aesthetic reasons.

Many other features can be used to classify these environments. After presenting the CLAM framework in next chapter we will concentrate on some of them in order to highlight its similarities and differences. Nevertheless it is already interesting to list them in order to focus our attention on those particularly interesting features. When looking at one of this environments, apart from the three conceptual issues previously commented we will also be interested in the following issues:

- Is it cross-platform?
- Is it Free Software?
- What are its limitations regarding the kind of signals that can be processed?
- Is it efficient?
- Is it maintained, updated and documented?

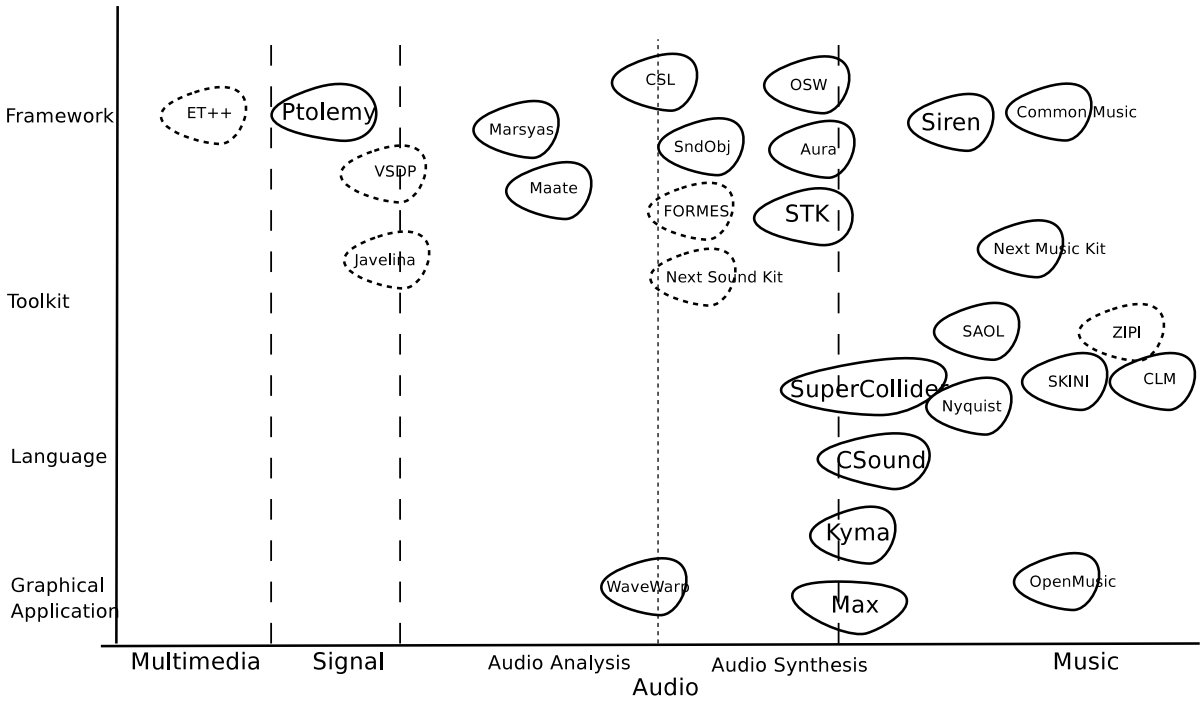

Figure 2.1: Classification of Audio and Music Environments

- Does it include both an algorithm repository as well as general infrastructure?
- What programming language is it written in?
- Is the model presented object-oriented?

Bearing all this issues in mind we will now give an overview of existing music and audio processing environments[1]. We have classified them into four main categories: general purpose signal processing and multimedia environments, audio processing frameworks, music processing frameworks, audio and music graphical programming applications and music languages. It is important to note that although we have forced every environment to be classified in one of the categories, it is seldom the case that they perfectly fit into it, having some features that probably belong to a different one. Therefore we should think of this classification as a fuzzy set with blurred border lines.

§2.2 General Purpose Signal Processing and Multimedia Environments

This first category could in fact be separated into two different ones: General Purpose Signal Processing Environments and General Purpose Multimedia Environments.

In the first subcategory we have environments that are designed to acknowledge any kind of input signal and offer convenient tools and constructions for processing them. In the second subcategory we have environments that instead of performing actual signal processing treat with multimedia objects such as video, images, or sound, offering tools to process them and to compose mixed multimedia scenes.

In any case we are not talking about environments specifically designed for audio or music. And although the environments presented in this section are able to treat with these signals and objects the reason for having included them here is a different one: the approach they represent and techniques they include are very relevant for the more focused environments and particularly for our CLAM framework to be presented in next chapter.

[1] Other environments such as AudioMulch [Bencina, 1998, www-AudioMulch,], JSyn[Burk, 1998], Cmix[Lanski, 1990, Riddell and Bencina, 1996, Helmuth, 1990], Sonic Flow[Seppänen and Kananoja, 1998a, Seppänen and Kananoja, 1998b], or SPKit[Lassfolk, 1995] have been reviewed and could also be included in the previous list. Nevertheless they have been discarded either because their particular approach is very similar to one already included or because they do not offer enough reliable information.

§2.2.1 Ptolemy

The Ptolemy Project [Hylands et al., 2003, www-Ptolemy,] is an informal group of researchers that is part of the Chess (Center for Hybrid and Embedded Software Systems) at the University of California, Berkeley. According to its authors, Ptolemy is above all "a laboratory for experimenting with design techniques".

The main focus of the Ptolemy Project is on embedded systems (embedded software is software that resides in devices that are not first-and-foremost computers), particularly those that mix technologies including, for example, analog and digital electronics, hardware and software, and electronics and mechanical devices. But the group is also interested on systems that are complex in the sense that they mix different operations such as networking, signal processing, or user interfaces.

Ptolemy is open source but its BSD license allows commercial software to be created with it, therefore maximizing its impact.

The Ptolemy group has produced three different frameworks: Gabriel, Ptolemy Classic and Ptolemy II.

Gabriel was developed between 1986 and 1991. It was written in Lisp and focused on signal processing. It included code generators for DSP's that produced efficient assembly code, especially for Motorola processors, and also hardware/software simulators which ran a parallel processor simulation.

Ptolemy Classic was developed between 1990 and 1997 but it has been still in use thereafter. It is written in C++. It was the first modeling environment to support multiple MoCs, hierarchically combined. The SDF (see 1.5.1.4) MoC implementation was ported from Gabriel and Boolean Dataflow (BDF), Dynamic Dataflow (DDF), multidimensional synchronous dataflow (MDSDF) and Process Networks (PN) were also added. DSP code generators were ported and C and VHDL code generators were developed. They developed a discrete-event domain and demonstrated joint modeling of communication networks and signal processing. And they also developed a hardware simulation domain called Thor. Portions of Ptolemy Classic were commercialized in different products.

The Ptolemy II version was started in 1996. The main reason for starting it was to exploit the capabilities of Java. It introduced the notion of domain polymorphism (a component could be designed to work on different domains) and modal models (where FSM are combined hierarchically with other MoC's). Ptolemy II added a sophisticated type system where components can be designed to operate on multiple data types and an expression language. Ptolemy II uses XML for data persistence. Some (but not all) the SDF capabilities of Ptolemy Classic were ported. The Ptolemy project contributed to a user-interface toolkit called Diva and used it to design a user-interface called Vergil. They built models

that could be used as applets from a web browser. Different experimental domains for real-time and distributed computing were also implemented. Instead of components as generators Ptolemy II uses a component-specialization framework built on top of a Java compiler toolkit called Soot. From now on when talking about "Ptolemy" we will be referring to the Ptolemy II version.

Ptolemy has a visual (block diagram) programming interface and a textual interface that offer two different ways of defining and modifying networks. Furthermore, the primitive actors or processes can be extended using the host language (Java).

The Model of Computation is not completely integrated into the framework and as a matter of fact, one of the goals of Ptolemy is to offer a platform for testing different MoCs[Lee and Park, 1995]. Nevertheless, some models of computation are already available and don't have to be provided by the user. Ptolemy offers support for Synchronous Dataflow Networks (SDF), Dynamic Dataflow Networks (DDF)*, Boolean Dataflow Networks (BDF)* and a more generic Process Network model (PN)[2]. Ptolemy also offers support for some other MoC's that are less interesting from our point of view. These include Component Interaction (CI), Communicating Sequential Processes (CSP) ,Continuous Time (CT), Discrete Events (DE), Distributed Discrete Events (DDE), Discrete Time (DT), Finite State Machines (FSM), Process Networks (PN), Synchronous Dataflow (SDF), Synchronous Reactive (SR), and Timed Multitasking (TM).

The amount of MoC's makes it difficult to choose one although most developers are usually just faced with a small subset of them. Nevertheless to design interesting systems heterogeneous models need to be used. According to the authors a grand unified approach could try to seek a model that serves all purposes. This could be accomplished by creating a mix of all the previous ones but such a mixture would be extremely complex and difficult to use.

Ptolemy supports both interpreted and compiled execution modes. Nevertheless, the code generation mechanism does not implement optimizations as these require more knowledge about the primitives than simply gluing together code fragments from each process.

Subgraphs can be encapsulated into a single node but it is always important to make sure that the resulting hierarchical node preserves properties of the conforming actors (see [Lee and Park, 1995] for examples of how this is not always guaranteed).

In Ptolemy, like in many similar software environments there are three phases to the execution of a program: setup, run and wrap-up.

In the first phase the hierarchical graph is traversed in order to initialize delays and state variables, evaluate parameters and the part of the schedule that can be statically evaluated, and perform

[2] At the time of this writing the ones marked with * have still not been integrated into Ptolemy II.

other setup operations. Parameters evaluated during this phase represent the part of the operation that does not operate on streams, defining a clear syntactic difference between *parameter arguments* and *stream arguments*. When compiled mode is used, code generation is done after the parameters have been evaluated, allowing for highly-optimized code. In Ptolemy, an actor or process that only has parameters as inputs is called a *source* as it has no dynamic inputs.

In the Run phase, the execution is carried on following either a precomputed schedule or computing the schedule on the fly.

Finally, in the Wrap-up phase if the run is finite (it is often not the case) the memory is freed and final results are presented to the user.

Ptolemy uses polymorphism for process data transmission. Input and output ports are strongly typed but they only expect data *tokens*. Any data type, even an array, can be encapsulated in a token.

Ptolemy is highly modular and has a careful package structure: *core packages*: support data model or abstract syntax and offer abstract semantics; *UI packages*: support for XML file format and visual interface; *library packages*: provide actor libraries that can operate on a variety of domains; and *domain packages*: provide domains each of which implements a MOC and some of which provide their own domain-specific actor libraries.

§2.2.2 Javelina

Javelina [Hebel, 1991] was a software environment for the development of discrete-time signal processing systems written in Smalltalk-80. It included notation editors and tools for digital filter design, mathematical optimization of state variable systems and the generation of optimized machine language code for a variety of digital signal processors.

It was developed by Kurt Hebel who latter became one of the creators of Kyma (see section 2.5.1). Although it is currently discontinued it still represents a different and interesting approach to signal processing environments.

The Javelina system is divided in three levels. At the highest level discrete-time systems are designed graphically with the Z Plane Editor or other mathematical function editor. At the middle level the mathematical description of the discrete-time system is manipulated. Algebraic transformations are used to improve characteristics such as output noise, memory use or processing time. The resulting mathematical function is then compiled into an intermediate register transfer language (RTL) for the low level. Finally this RTL is optimized and machine code is generated.

§2.2.3 VDSP

Virtual Signal Processing (VDSP) [Mellinger et al., 1991] was an environment for digital signal processing in terms of a *virtual processors* and its associated *virtual data*. This environment for vectorized signal processing was implemented in Smalltalk-80 and it was hardware independent. As Javelina, VDSP is currently discontinued.

In VDSP a virtual processor is an object that provides a means for allocating and deallocating data objects, a standard set of vectors, scalar and vector-scalar operations, input/output to external devices capable of handling data formats commonly used in signal processing, and it provides communication between real processors and the user's computing environment.

Virtual data encapsulates the data and offered the following functionality: they provide a handle that is used by the virtual processor, they provide representations of useful data types, and they respond to messages requesting operations on their data by invoking the corresponding operation on a virtual processor. For instance, multiplying two arrays of integers using a virtual processor means: (1) send a message telling the virtual processor to allocate two vector objects whose contents are obtained from the two Arrays; (2) tell one vector to multiply itself by the other; (3) tell the result vector to make a new Smalltalk Array whose contents are the data of the result vector.

VDSP is not an optimizing compiler but rather a portable high-level interface to common signal-processing functionality. It is not a stream computation system, at user request it performs one operation on one block of data at a time. It has a primitive kernel made up of operations such as addition or multiplication.

§2.2.4 MET++

MET++ (Multimedia ET++) is general compositional environment for multimedia with interactive editing facilities in which multimedia presentations are regarded as a hierarchical composition of time objects. MET ++ was developed at the University of Zurich by Philip Ackermann [Ackermann, 1994a]. It was published and distributed as Free Software in 1996 but no updates have been added since then as the author joined a startup company shortly after and discontinued its support[3]. Nevertheless, the framework has been used for several developments and it is a reference for multimedia software design.

[3]Thanks to the fact that MET++ was distributed as Free Software the source code is still available for download. This means that although support and development are currently discontinued, anyone can take it and develop it further. For this reason, and unlike other discontinued environments, we have chosen not to write this description in past tense.

Much of its impact is due to the fact that the media framework is based on the ET++ framework written in C++ also developed at the Univ. of Zurich. ET++[Weinand et al., 1989] is an OO framework that integrates user interface building blocks, basic data structures, support for object input/output, printing, and high-level application framework components. ET++ is a classical reference in OO literature for being a test-bed for software engineering practices such as design patterns and refactoring. To the already existing support for 2D objects present in ET++, MET++ added support for 3D graphical objects, camera objects, and audio and music object.

MET++ is an object-oriented application framework that addresses the main difficulties that can be encountered when designing applications for multimedia authoring or editing, namely: integration of several media types with different real-time constraints at low-level device interfaces; support for media compositions to define high-level inter-media synchronization and real-time control; intuitive user interface interactions with direct manipulations of graphical representations and high semantic feedback; and flexibility and portability for different hardware configurations and platforms.

Although the main development platform of MET++ was a Silicon Graphics Indigo workstation, the framework is cross-platform and hardware dependencies are hidden in a portability layer that provides abstract interfaces to operating system, windows system, as well as audio and MIDI input/output.

Perhaps the most important aspect in the framework is the way it handles time synchronization [Ackermann, 1994b]. The temporal information is represented through a hierarchical composition that includes information about part-of relations, grouping, and temporal constraints and allows automatic time calculation. The components in the structure are modelled as time events that have their own starting point, duration and virtual line. The base class of the time dependent objects is the `TEvent`. The high-level synchronization accuracy is based on milliseconds time resolution. In a multimedia presentation temporal relationships are configured by composing several universal time objects such as `TSequence`, `TimeLine`, `TSynchro`, `TShift` and `TLoop` with special media objects. Any time dependent object inherited from TEvent can be inserted into the composition.

The time dynamic behavior of a time wrapper is supported through time functions expressing actions such as fading, scaling or positioning. A time function can manipulate the time itself so that every object can have its local time. In real-time presentations there is usually a `Conductor` object that periodically sends `Perform` messages to the time dependent objects in the hierarchy. Media objects compute only the necessary data for the next interval. Continuous media data is computed incrementally ahead into a buffer. Interactions such as fast-forward or slow-motion are realized by changing the parameters of the `Perform` message: start time, presentation duration and real-time duration (the

behavior of the media objects depend on the way they respond to the relation between these two).

The `Intensity` class is used for representing values for amplitude, gain, volume... It can operate and convert itself to byte, integer, floating-point, decibel and MIDI value. The `Pitch` class is able to handle information about pitch key and frequency. It uses the Tonal System to map symbolic keys to Hz. The `Beat` class models beat and bar properties and delegates quantization and the mapping from symbolic beat measure to physical seconds to the `MusicalContext` class. The `PitchScale` class can represent symbolic scales (chromatic, dorian...) or physical intervals for tuning purposes.

Audio resources are modelled with the abstract `AudioUnit` class as modules of a source-filter-sink architecture. The output of an `AudioUnit` can be sent to another `AudioUnit` and define an audio signal flow graph. Audio files can be played in real-time from disk or converted to a Sound object.

The interpretation of a musical performance is modelled in the time-dependent `MusicalContext` class. It holds information about the tonal system, tonality, signature, measure, and tempo. The `MusicPlayer` interprets its notes in the context and delegates the note playing to the abstract `MusicInstrument` class. The concrete classes `MIDIInstrument`, `CSoundInstrument`, and `Synthesizer` (allocates Oscillators) map the note representation to device specific synthesis parameters.

§2.3 Audio Processing Frameworks

The environments presented in this category are frameworks in the sense defined in section 1.3. They provide access to the sources code and allow extending it in a white box manner but they also offer components that can be used in black box mode.

We also subdivide this category into three different subcategories. Analysis oriented audio processing frameworks are those that focus on offering tools for signal analysis, understanding and classification and hardly offer any tools for processing or synthesizing sounds. Synthesis oriented frameworks are, on the opposite, those that offer tools for audio processing and synthesis but not for analysis. Finally, the broadest subcategory, that of general purpose audio processing frameworks, includes frameworks that provide tools for both analyzing and synthesizing sounds.

§2.3.1 Analysis Oriented

Analysis oriented frameworks are audio processing frameworks with a clear focus on extracting features from an incoming audio signal. Although some other frameworks that will be seen in other

categories may also include audio analysis tools, two particular frameworks, Marsyas and Maate, have an exclusive focus on such applications.

§2.3.1.1 Marsyas

Marsyas [Tzanetakis and Cook, 1999, Tzanetakis and Cook, 2002, Tzanetakis and Cook, 2000, Tzanetakis, 2002] or MusicAl Research SYStem for Analysis and Synthesis is a framework for experimenting, evaluating and integrating techniques for audio content analysis. Although the name includes the word Synthesis, Marsyas' focus is clearly on sound analysis tools and information retrieval techniques. The framework allows to integrate these tools using a semi-automatic approach and a graphical interface. On the other hand Marsyas is released under the GPL license and is therefore Free Software.

In order to come up with a valid model for Marsyas, different algorithms and techniques were studied and common behavior and features were abstracted. OO programming techniques were used to implement abstract classes that provide a common API for the building blocks of the system and inheritance is used to factor out common operations.

The environment is able to combine traditional bottom-up processing (from signal to metadata) as well as top-down (according to the author prediction-driven, for instance, has proven to be interesting). Although the objects form a natural bottom-up hierarchy, top-down flow of information can be expressed in the framework (e.g. a silence feature can be used by an iterator for music/speech to avoid calculating features on silent frames).

The framework design is based on a client-server architecture. The server is written in C++ and contains all the signal processing and pattern recognition algorithms, optimized for performance. The client is written in Java, contains only the graphical interface and communicates with the server using sockets. Both the server and the client run on Solaris, SGI, Linux and Windows.

The main classes of the system can roughly be divided into process-like and data-structure-like.

The Process-like classes can be divided in the following categories:

- *Transformations* are low-level signal processing units used by the system. They take as input a frame of sound samples and output a transformation of that frame (e.g. power spectral density, caepstrum, windowing...)

- *Features* process a frame of sound samples and output a vector which unlike transformations is reduced significantly in dimensionality. More than one "physical" feature can be combined in a single vector.

- *Memories* are circular buffers that hold previously calculated features for a limited time. They

are used to compute means and variances of features over large windows.

- *Iterators* break up a sound stream into frames. For each frame they use memories and features to compute a feature vector. The time-series of feature vectors is called the feature map. Typically there is a different iterator for each classification scheme (e.g. silence/non silence iterator uses only energy as a feature and no memories, the music/speech iterator uses 9 features and 2 memories (of different sizes).

- *Classifiers* take as input a feature vector and output its estimated class. They are trained using labeled feature maps.

- *Segmentators* take as input feature maps and output a signal with peaks corresponding to segmentation boundaries.

Data structure classes can in turn be categorized in:

- *Vectors* are the basic data components of the system. They are float arrays tagged with sizes. Operator overloading is used for vector operations to avoid writing many nested loops for signal processing code. The operators are inlined and optimized and the resulting code is easy to read without compromising performance.

- *Sound data* contain samples of audio as vectors with header information such as sample rate or channels.

- *Feature maps* are time-series of feature vectors. They can be class labeled for evaluation and training.

- *Time regions* are time intervals tagged with annotation information.

- *Time lines* are lists of time regions.

- *Time trees* are arbitrary trees of time regions. They represent a hierarchical decomposition of audio into successively smaller segments.

All objects contain methods to read/write them to file and transport them using the socket interface.

Implemented features in the framework include spectral centroid, spectral moments, spectral flux, pitch, harmonicity, mel-frequency cepstral coefficients (MFCC), linear prediction (LPC) reflection coefficients, zero crossings, RMS, and spectral rolloff. For all of them means, variances and higher-order statistics can be computed using memories. New features can be easily added by just writing the code for computing the feature over a frame of samples.

Two classifiers have been implemented: the Gaussian (MAP) classifier and the K-Nearest Neighbor (KNN).

Different applications such as music/speech discriminator have been implemented in order to test the architecture.

The user interface looks like a typical tape-recorder wave editor but in addition it allows skipping by either user-defined fixed duration blocks or time lines containing regions of different duration.

At the moment of this writing Marsyas is going an overall rewrite towards a 0.2 version of the framework.

§2.3.1.2 Maaate

Maaate is an audio analysis toolkit that supports the extraction of structure and content of MPEG-encoded audio files as well as raw files. Maaate was implemented by Silvia Pfeiffer and other researchers of the Commonwealth Scientific and Industrial Research Organization (CSIRO), Mathematical and Information Sciences (CMIS) in Australia. It is offered as Free Software under the GPL license for GNU/Linux and Windows. Maaate is implemented in C++ but a C wrapper interface is also offered. The framework is functional but not updated regularly, with the author's focus turning in other directions.

There are several reasons for which the authors justify the use of MPEG encoded files. First, MPEG is a de facto standard not only for internet audio for also for audio encoding in video formats. Second, the encoding process usually removes non-perceivable sounds that are not interesting either for content description applications. And finally, MPEG encoded files already have frequency domain information that can be used to extract some descriptors more easily and efficiently as the library allows to access directly the fields of the MPEG encoded file without having to decode it.

Apart for the general infrastructure, Maaate also includes modules with analysis algorithms such as loudness approximation, segmentation, silence or background noise level.

Maaate's architecture allows to easily add new algorithms. It is designed in tiers in order to separate functionalities and offer a cleaner API.

Tier 1 is in charge of parsing MPEG streams and offers access to the encoded fields. The most important class in this tier is the MPEGfile class.

Tier 2 offers two generic data containers (`SegmentData` and `SegmentTable`) that can be used from the analysis modules. `SegmentData`: contains analysis data for one segment (continuous time range) of the audio file. A `SegmentTable` contains a collection of segment of an audio file. These segments are ordered by their start time. Tier 2 also offers a module interface to plugin analysis routines that

are stored in dynamic libraries. By using the offered plugin interface modules can be developed and compiled separately from Maaate and the toolkit can be extended without ever having to recompile it. The boundary between Maaate and the analysis modules is explicit so the author can clearly keep his/her authorship.

A module is simply a collection of related functions that offer a particular functionality. Modules that analyze content usually collect information on several MPEG audio frames and calculate information from these. Tier 2 was constructed for such modules but other kinds are also possible. Modules can be broadly classified into feature extraction, feature analysis and content analysis. Feature extraction modules make use of tier 1 access functions and store the result in tier 2 containers. Feature analysis modules use the already extracted features for further (usually statistical) analysis such as clustering or segmentation. Finally content analysis modules calculate higher level information such as silence/music/speech discrimination using feature extraction and analysis modules and storing the result in convenience containers.

A module is in fact an instance of the Module class, which also offers convenience functions to access the information, handle parameters, check constraints, and call the module functions. The *apply* method contains the implementation of all the analysis functionality, taking a list of parameters as input and producing another list of parameters as output.

Any module must offer an appropriate constructor and a function to add itself to the list of available modules. The other important operations present in the Module class and all its subclasses are the following:

- `init` (required): sets up basic module information (name, author...) and the input/output parameter specification.
- `default` (required): sets default values for input parameters and returns the parameter list.
- `suggest` (optional, recommended): takes an input parameter list, suggests parameter values based on information from other parameters and changes constraints on input parameters.
- `reset` (optional): resets a module by reseting internal processing information or parameter values.
- `apply` (required): takes an input parameter list, performs the analysis function and returns the calculated output parameters.
- `destroy` (optional): cleans up allocated memory and deletes parameter specification.

A parameter is an instance of the `ModuleParam` class. The `init` function sets up the list of input/output parameter specification and the application calls the `default` function to set default values. From there

on, the input parameters may be modified by the application. It can also call the `suggest` function at any time for filling-in necessary parameter values and constraints and performing sanity checks. Finally, the application may then call the `apply` function, which first checks for parameters to be within the constraints.

There is a short list of allowed simple or complex data types for parameters. Basic types may be: boolean, integer, real and string (all of them defined in Maaate with particular conditions). Complex types may be: a pointer to an opened MPEG file, a pointer to a segment data structure, or a pointer to a segment table. All of them have their particular `MAAATE_` define.

There are three types of constraints: no constraints, a list of single values that allow the parameter to take as value any on the list, and a list of ranges according to which parameters may take any value in any range on the list. A constraint is an instance of the `ModuleParameterConstraint` class that can handle either single values or a single range. For a single value there is usually a list of constraints realized by instantiating the `MaateConstraint` class, a class that provides functions for such things as adding constraints or checking whether values satisfy them.

On the other hand, a parameter specification is an instance of the `ModuleParamSpec` class, which contains the specification of a single parameter. The specification consists of a name, a description, a data type, default values and constraints.

The `Plugins` class provides functionality to load and unload single modules and whole libraries, and administrates the list of available modules. For building a shared library a function loadModules must be supplied that instantiates the modules and returns their list to Maaate. For dynamically loading modules into an application an instance of the `Plugins` class must be declared and the particular library must be loaded using the class' `AddLibrary` function. Then the different loaded modules can be accessed with the `GetModule(string)` function.

Bewdy is a sample graphical application that demonstrates Maaate by using a few of its modules. It runs on GNOME and needs the OSS sound system.

§2.3.2 Synthesis Oriented

Just as in the previous section we commented audio processing frameworks with a clear focus on analysis in this one we will include three frameworks that switch their focus to synthesis. Apart from the obvious inclusion of different kinds of synthesis algorithms new requirements come into play, namely those related to real-time and user interaction.

§2.3.2.1 STK

The Synthesis Toolkit in C++ (STK) [Cook, 1996, Cook and Scavone, 1999] is a set of open source audio signal processing and algorithmic synthesis classes written in C++. It was developed to facilitate rapid development of music synthesis and audio processing software, with an emphasis on cross-platform functionality, real-time control, ease of use, and educational example code. The fundamental design goals of STK were defined as follows: (1) Cross-platform functionality; (2) Ease of use; (3) User extensibility; (4) Real-time synthesis and control; (5) Open source C and C++.

STK has been distributed freely since 1996 and included in various collections. Perry Cook started developing STK under NeXTStep at CCRMA in the early 90's. When he moved to Princeton in 1996 he ported everything to C++ on SGI hardware, added real-time and enhanced the synthesis algorithms. With the help of Bill Putman he made a port to Windows95. Gary Scavone began using STK in 1997 and completed a full port to Linux in 1998. He finished the fully compatible Windows port (using Direct Sound API) in June 1998. A great deal of improvements and extensions have been done since then[Cook and Scavone, 2003]. It has also been ported to Max/MSP on Mac by Dan Trueman and Luke Dubois, and distributed as PeRColate. STK is updated on a regular basis.

STK is a framework, not a particular application. Some applications are distributed as an example of usage but even these will probably have to be personalized. Examples don't have a fancy GUI wrapper as, according to the author, it goes against the spirit of STK to spend many hours developing GUI's that will then not be completely cross-platform.

STK works with real-time support (MIDI and audio) on SGI (Irix), Linux, Macintosh OS X, and Windows. STK is free for non-commercial use. It offers some simple Tcl/Tk GUI that offer the same interface as the MIDI input. It can generate SND, WAV, AIFF and MAT file outputs.

Almost all STK is regular C/C++ code that can be compiled on any platform. OS dependencies are kept within a small number of classes. In order to make the GUI cross-platform Tcl/Tk is used. It has no other hardware requirements than a regular sound card. STK is object-oriented and the code is clear. Some optimization issues are sometimes addressed but in general optimization is sacrificed for the sake of clarity.

All STK classes inherit from `Object` class. This class does not offer any functionality but it is a convenient mechanism for defining global program and operating system parameters. For instance, `MY_FLOAT` can be defined as either float or double in Object.h. Audio sample based classes implement the `tick()` method. This method returns a `MY_FLOAT` or a `MY_MULTY` (a pointer to `MY_FLOAT` for multichannel audio). Inside this method, all computations take place. The `lastOut()` method returns the last result of a computation allowing a single source to feed multiple consuming objects without having to copy

the result in an external variable.

STK implements only single-sample `tick()` functions minimizing memory usage, allowing to build short recursive loops and guaranteeing minimum latency. No specific support for vectorized classes is planned but they are designed to allow easy conversion.

At its core, STK uses the unit generator paradigm from Music N (see section 2.6.1). All unit generators derive from the Instrument base class. Instrument classes include envelopes, filters, noise generators, nonlinearities, and data input/output handlers. `WvIn` and `WvOut` and associated classes allow to handle .wav, .snd, .mat (Matlab) and .raw files as well as real-time audio input and output.

There are many different synthesis algorithms: oscillator-based additive, subtractive, FM, modal, sampling, physical models of string and wind instruments and physically inspired particle models. Several models are provided for the voice and more are planned for the future. It also includes several simple delay based effects for reverberation, chorus, flanger and pitch shifting.

STK control sources connect to synthesis programs via pipes and sockets allowing for networked connections and decoupling audio synthesis from control generation. An input handler MD2SKINI converts input MIDI controls to the SKINI score format (see section 2.6.2.2).

§2.3.2.2 Open Sound World (OSW)

Open Sound World (OSW) [Chaudhary et al., 1999, www-OSW,] is a scalable, extensible programming environment that allows to process sound in response to expressive real-time control. OSW combines the familiar visual patching paradigm with solid programming-language features such as a strong type system and hierarchical name spaces and an intuitive model for specifying new components. OSW is also highly dynamic and allows users to both edit *transforms* and manipulate performance controls simultaneously, run audio signals at several rates simultaneously and change patches or the basic configuration even while the audio is running.

OSW allows development of audio applications using patching, C++, high-level specifications and scripting. New components can be expressed using familiar mathematical constructs without a deep knowledge of C++ programming and the processed data can have any valid C++ type. OSW uses a reactive real-time scheduler that safely and efficiently handles multiple processors, time sources, and synchronous dataflows.

OSW is a dataflow programming language: users connect primitive components to form a network. Each component accepts data, processes and sends it back to the network. OSW is also an OO language because components are instances of classes that specify their behavior. OSW employs a visual programming environment that allows users to instantiate and connect graphical representations

of components.

Primitive components in OSW are called *transforms*. They accept data through their *inlets*, and produce results in their *outlets*. For instance, in OSW an oscillator is a transform with two inlets (timeIn and frequency) and one outlet (samplesOut). Transforms can be very simple or extremely complex. Because computations are more efficient between a single transform, complex ones are often favored

Transforms can be connected to form larger networks called *patches* which are themselves also transforms. Connections are strongly typed and each outlet is connected at most at one inlet. In order to connect one outlet to different inlets a special FanOut transform must be used. The same way, one inlet must only be connected to one outlet or else use the FanIn transform.

The work in transforms is done inside *activation expressions*. An activation expression is a piece of C++ code that is executed when certain inlet or state variable is modified. If the expression depends on more than one input variable, all of them must be modified in order to trigger the expression and then the activation can be executed immediately or be delayed for a previously specified amount of time. The result of an activation expression is usually assigned to one or more outlets. Whenever an outlet is assigned a new value, this value is sent to the connected inlet; if the receiving transform is *active* it will respond with an activation expression, if it is *passive* the value is assigned to the inlet but no further processing occurs.

OSW provides a set of primitive data types (integers, floating point...) as well as some useful datatypes for music and signal processing such as samples (as floating point or integer numbers), frequency domain spectra, notes, MIDI events, and SDIF. It is relatively easy to add new data types as C++ classes. OSW uses a hierarchical namespace that can be used to reference instances or variables (e.g. "/sinewaveplayer1/sinewave/frequency"). In a similar way transform classes are grouped into packages PackageName::Transform.

The Get and Set transform can be used to query and modify variables by their pathname. They are similar to the "goto" in structured programming. Its expressive power is that you can break the dataflow model and access variables in a transform that is not connected. Abusing of such feature though may make the resulting program hard to read and debug.

OSW provides abstractions for the input/output devices as well as transforms for communicating with them. OSW supports audio hardware, MIDI I/O, Ethernet and serial ports. It can also be extended to support additional devices such as graphics tablets.

Some transforms include outlets of type *any* that can be connected to any kind of inlet. Every time a new data is sent from the outlet to the connected inlet, it must be checked if it is a compatible

type. The use of *any* is inefficient and seldom used (usually only for `Get` and `Set` on global or *free* variables). On the other hand some transforms include *dynamically typed inlets* that are assigned a type at connection time. These transforms though do not introduce any run-time efficiency penalty and they are used for polymorphic operations such as arithmetic operators.

In dataflow languages for signal processing several copies of the same connected transforms are often done (e.g. for multichannel processing). OSW offers a different abstraction (apart from the obvious of creating a separate patch): transform arrays. An `Array` transform takes a transform class name, an integer, an instance name and arguments for the transform class and makes a single object containing n copies of the transform.

Bundles are an abstraction of buses in audio engineering, they are used to transmit n data objects over a single connection (especially useful in `Array` connections). Special transforms are used in order to convert to an form bundles.

Transforms are implemented in C++ while the graphical interface is implemented in Tcl/Tk and Tcl scripting is also used for defining patches. OSW offers an extensible object-oriented model that allows users to develop at different levels including visual patching, high-level C++ and Tcl scripting. Users can provide C++ code or Tcl script to override the default behavior of the transform in the graphical environment.

OSW includes a tool called the "Externalizer" that automatically converts a high-level specification into efficient C++ code. The code is then compiled and can be loaded into the OSW environment. The Externalizer presents a transform as a collection of inlets, outlets, state variables and activation expressions that the user can modify. The externalizer also allows to define new data types to be used for transform variables (a conversion to string must be included).

Expert programmers can bypass the Externalizer and write the transforms in C++ directly deriving from one the base transform classes: Transform or TimeDomainTransform. All derived classes include members for inlets, outlets, state variables and activations, a member function for each activation and a constructor.

OSW allows users to write familiar mathematical expressions instead of hand optimized C++ code, this is accomplished through the use of operator overloading and the osw::vector class.

OSW is designed for implementing *reactive real-time* audio and music applications. Reactive real-time involves maintaining output quality while minimizing *latency*, delay between input and output of the system, and *jitter* (change of latency over time). In OSW there is also a notion of *real time* and *virtual time*. Real time is a quantity that increases at a fixed rate as measured by a clock while virtual time can be scaled or translated (e.g. tempo changing, fast-forward...).

An audio output device is a transform that has only state variables and no inputs or outputs and is not part of any patch but can be accessed using Get and Set. The output device controls when the transforms produce the samples by controlling the device clock. When the clock is updated it triggers the queue of activations, evaluating those scheduled to occur in that time.

Synchronous transforms are those that produce samples as a function of virtual time. Activation expressions are guaranteed to occur exactly once each period of the clock to which the transform is synchronized. The order in which they are executed within a period is also fixed. This is true of more general *synchronous dataflow graphs*. Because OSW allows multiple audio devices and clock sources, several synchronous dataflow graphs that run at different sample rates and buffer sizes are supported.

Unlike transforms in the synchronous chains, the relative execution times of asynchronous transforms (events coming from GUI or MIDI device) cannot be predicted. Some inlets and state variables that are sensible to asynchronous updates must be protected. (e.g. If the coefficients of the filters are updated asynchronously the filter may become unstable) This protection is included in the more general parallel scheduler as asynchronous events are a special case of parallelism.

OSW is designed to take advantage of multi-processor capabilities. Given enough processors, each transform could be executed on a different processor, executing whenever its inlets or state variables change, but processor utilization would be poor.

A *chain* is a set of connected transforms in which no outlet or inlet in the chain is connected to a transform not in the chain. A chain has no branches. It is said to be a *maximal chain* if no other chain can contain it (i.e. if we add another transform it will no longer be a chain). Maximal chains must be scheduled sequentially while separate maximal chains can run in parallel. They are considered the formal unit of parallel computation in OSW. Deferred deallocation can be performed between execution calls to maximal chains or in a low-priority separate thread.

OSW provides methods for locking and unlocking variables. When a value is written onto a locked variable they are placed in a buffer and assigned to the variable only when this is unlocked (this technique is known as double buffering).

OSW supports networking. Open Sound Control (OSC) is a protocol for high-level control of sound synthesis and other applications. It divides the world into clients that generate control messages and servers that produce sound. OSW is a natural server for OSC.

OSW also supports the SDIF format and OSW patches are tcl scripts that can be downloaded and executed in a browser with a plug-in.

§2.3.2.3 Aura

Aura [Dannenberg and Brandt, 1996b, Dannenberg, 2004, Dannenberg and Brandt, 1996a] is a framework for software processing of audio and music signals mainly developed by Roger B. Dannenberg as a generalization of the CMU MIDI Toolkit. Although Aura has been under development for some years the author considers that it is still not mature and stable enough to offer it publicly mainly because of its lack of a packaging infrastructure and appropriate documentation. Nevertheless, the author plans on offering it as Free Software.

The first versions of Aura ran on MS Windows but the latest ones work on GNU/Linux using Port Audio, Port MIDI and wxWidgets for the graphical interface. Although the use of system-dependent features is sometimes necessary for real-time performance, Aura aims at being as much portable as possible, encapsulating all system-dependent code for things such as graphics, MIDI or audio. On the other hand, Aura tries to reuse as much code as possible from previous efforts by the author and intends to be itself reusable.

Aura offers a way to create, connect and control signal processing modules. It was designed with audio processing in mind and offers support to multiple threads and low-latency real-time audio computation.

The framework was in its first versions divided into an architecture for audio signal processing, which was called Aura, and a framework for building event-driven real-time software called W (see [Dannenberg and Rubine, 1995] for a description of the framework in its inital versions). W was designed as a framework for general real-time computing. W was a successor of V, itself a successor of Garnet, a constraint system written in Lisp. In a musical context, W was used for handling user interaction and MIDI data. The combination of W and Aura allowed operations such as controlling synthesis software from a MIDI keyboard. But in its latest versions (see [Dannenberg, 2004]) the name Aura is used to refer to the whole framework including the features and functionalities inherited from W.

The Aura model involves objects that send messages to other objects. Objects have typed attributes (integer, double-precision float...). Some objects are made by the users while others are part of the standard Aura environment (debugging tools, I/O ports for MIDI or audio, and interface components).

The basic concepts in Aura are the *unit generator* (or *ugen*) and the *instrument*. A unit generator is the encapsulation of a DSP algorithm in an object that has inputs, outputs and some sort of internal state. The concept is borrowed from the Music N paradigm (see 2.6.1) and it is shared by most other environments reviewed in this chapter. An instrument is a static composition of unit generators that is designed on pre-compile time.

According to the author, all of the previously existing solutions suffer from focusing on a single paradigm: graphical edition or textual-based programming and control. Graphical systems are easy to use but make it difficult to reconfigure at run-time. Textual systems offer more flexibility but lose some intuitive feel and debugging support. Aura offers both graphical edition and textual programming in an attempt to offer the best from both worlds.

But more than textual vs. graphical the author is interested in static vs. dynamic systems. Static connections can be more efficient but are far less flexible. Aura 2 supports from fully static to fully dynamic systems. Static graphs are created with the visual editor forming instruments members of the Aura Instr class. The instruments then can be dynamically allocated and connected at run time.

Although the first versions of Aura intended to offer a computing graph as dynamic as possible, experience indicated that most designs were mainly static. By using static instruments the main advantages are: more efficiency avoiding the overhead of dynamic patching; better debugging; inlining and other tricks can help improve performance; and users can reason better about their code when it is static.

In dynamic graphs, when graphs are updated Aura recalculates the execution order. It also uses reference counting to delete objects that are no longer referenced. Aura also uses global names for instruments and a remote procedure-call system. Instruments can be controlled from process running asynchronously.

The Aura visual editor is used to integrate unit generators into instruments and it is integrated with text-based programming. When the user creates a new instrument, the editor can automatically update the user's makefile and write scripting language functions to create an instance of the instrument, integrating the new instrument automatically into the user's program.

After the user creates a valid instrument, the editor can generate its C++ implementation. A `set` method is generated for each signal input to an instrument. The user can then make a remote procedure call to this method using C++ or Serpent (Aura's real-time scripting language). This can be done dynamically. A topological sort is performed on the instrument graph so that signal flows from input to output in a single pass. Buffers are allocated for intermediate results and they are reused whenever possible to minimize storage. Although it is assumed that an optimized buffer allocation policy is necessary to have efficient DSP applications, the author made exhaustive testing to find out that this only slightly matters for large collections of unit generators.

Aura supports different kinds of signals and the editor helps by selecting compatible types and unit generators. The types are:

- Audio-rate signals are streams of floating-point numbers processed one block at a time (typically

32 samples by block).

- Block-rate signals are computed synchronously with audio but have only one sample per block.
- Constant-rate unit generators remain at the same value until changed by a message.
- Interpolated inputs accept block-rate signals and convert them internally to audio-rate.

In the Editor the user selects the unit generator by a generic name and the editor selects the appropriate one depending on the input/output types. Different code and name is needed for each combination of input rates. This could lead to an explosion of different implementations for the same ugen but according to the author, no ugen has needed more than a dozen.

Furthermore the Aura graphical Editor is capable of automatically generating a visual interface for any new instrument in order to test and debug it.

Messages are time-stamped with the time stamp indicating when the message should be received. An object can send a message to a particular target or simply send it to all objects that have been connected to its outputs. These connections are created on run-time.

Aura supports a model of computation based on fixed-priority scheduling and the notion that all computation within an object should run at the same priority. Each object is assigned a priority level called a *zone*. Computation of different objects within a zone is non-preemptive and runs on a single thread. E.g. an application can be divided in three zones, one for GUI objects, one for MIDI objects and the other for audio objects. It is up to the application designer to insure that the total computation in a zone (and all higher priority zones) does not exceed the shortest latency required in that zone but Aura can assist in measuring computation times and detecting exceptions. Messages are delivered synchronously within a zone. Messages between zones are sent asynchronously. All interzone messages are enqueued by the sender and later delivered to the receiver.

A message stamped for the future is usually stored until its time comes. But a *precomputation* zone may run ahead of real time, reading incoming messages before it otherwise would, it can then compute and send messages to a real-time zone where they are automatically buffered. (E.g. a precomputation zone can be used for reading audio or video files).

For efficiency, data is usually calculated in blocks of 32 samples. According to the author Aura gains roughly a factor-of-two performance increase over previous systems. In particular CSound (see 2.6.1.2) does not use interpolation to smooth control signals so users must compensate by computing with short inefficient sample blocks. And the ISPW (see 2.5.2) does not offer low sample rates for control signals so it computes twice as many samples as Aura to achieve the same result. But, as the author acknowledges, these other systems are more mature an offer a more complete library.

§2.3.3 General Purpose

This final subcategory of audio processing frameworks includes frameworks that do not have a clear focus neither on audio analysis or synthesis. The first three include functionalities both for extracting features from a sound and for synthesizing it. The third one offers system level general tools.

§2.3.3.1 The Create Signal Library (CSL)

The Create Signal Library (CSL) [Pope and Ramakrishnan, 2003], pronounced "Sizzle", is a general purpose C++ library for digital audio processing mainly designed by Stephen Travis Pope, also author of the Mode and Siren frameworks (see section 2.4.2). The first implementation dates back to 1998 when it was called the CREATE Oscillator or CO. But the current implementation was started by students in 2002. CSL has been used to build stand-alone applications, interactive installations, MIDI instruments, and light-weight plug-ins.

CSL programs are written in standard C++ and linked against the library. CSL has no graphical interface but GUIs are expected to be built for manipulating patches. CSL is not a music representation language rather it is a low-level signal processing and synthesis engine.

CSL works on Linux, Unix (Solaris, Iris, OpenBSD), and MacOSX. Windows is supported but some features (such as abstraction for network and threads) are missing .

CSL applications can be controlled via Siren (see 2.4.2) or MIDI messages. CSL has no scheduler, it simply responds to incoming control messages as fast as it can. CSL has no notion of time but unit generators may have state.

The goals of the CSL project can be summarized in the framework being *a scalable*, *portable*, and *flexible* network-driven sound synthesis package . By scalable, the author means "orchestrascalable": large groups of instruments with complex synthesis models. This scalability will be accomplished by running clusters of CSL-based synthesis and processing server programs on many computers connected by a fast LAN. Portable means that the software must not depend on any hardware platform or operating system. It is written in standard C++ and uses hardware abstraction classes for I/O ports, network interfaces and thread APIs. Flexible means that the library should support several techniques of sound synthesis or processing and also be useful for embedding in other applications.

In a CSL program there are C++ objects called "unit generators". They can be connected using C++ variables representing their inputs and outputs. In order to connect an object A to the input of object B a simple method must be called: B.root(A). The scheduling is done by pull. The output device asks for samples and this request is propagated.

The CSL library consists of several components: (1) The object framework for the synthe-

sis/processing engine; (2) The unit generator class library; (3) The start-up, configuration and system save/restore facilities; (4) The OSC control interfaces; (5) The database interface for sound samples and spectra; (6) The CRAM interface for managing multiple CSL instances over a network.

The OO domain model consists of abstractions for objects that create or process blocks of samples (Buffer, FrameStream, SampleStream, Processor...); objects representing control variables (StaticVariable, DymanicVariable...); objects that connect to I/O driver (`IO` and its subclasses); and objects that help manage CSL patches and instrument libraries.

The heart of CSL is its unit generator and signal processing class library: the subclasses of FrameStream. There are several "control sources" such as wavetable oscillators, noise sources, chaotic generators, FFT/IFFT. Signal processors such as filters and panners take synthesis graphs as inputs. They are subclasses of both FrameStream and the mix-in Processor class. CSL includes canonical form FIR filters, panners, mixers, convolution and flexible delay lines. Simple operators are handled by the `AddOp` and `MulOp` unit generator.

A CSL program is a graph of DSP units, generally a number of *patches* (subgraphs) connected to a mixer. This graph has a single *root* node, usually the output unit generator or a mixer taking several subgraphs at its inputs. Different sound file formats can be loaded into a graph. The evaluation of a graph is triggered by the pull of an `IO` object (an instance of the `IO` class) that is usually connected to a direct output API such as PortAudio [Bencina and Burk, 2001], to a socket-base network protocol or a sound file.

The blocks of samples can be sent through sockets using a protocol based on UDP in which data packets have a header that incorporates an instance ID and sequence number. The mixer and the spatializer are CSL programs that perform no synthesis but instead read sample blocks from other CSL instances over a network and process them.

CSL can be used in distributed systems. The framework is callable from Open Sound Control [Wright, 1998a] or Corba. Its output samples can be sent directly to an output device or to a network socket. Process in different machines support inter-machine sample streaming and are integrated in the CREATE Real-Time Application Manager [Pope et al., 2001].

In the basic CSL framework there is no essential difference between constant values, control signals and audio signals. Samples are usually 32 bits floats though this can be changed to integer or 64 bits with a single definition. All processing is done in blocks typically between 32 and 1024 sample frame in size. Envelopes are breakpoint functions of time. There are helper classes that provide constructors for standard envelope types: triangle, AR, ADSR, various windows, etc...

The main declarations are in the FrameStream.h file which defines the following classes: `Buffer`,

the basic n-channel sample buffer class; `FrameStream`, the central abstraction in CSL; `SampleStream`, an l-channel frame stream; `Processor`, a mix-in for framestreams that process an input frame stream; `Writeable`, a mix-in for framestreams that one can write into; `Phased`, a mix-in for framestreams with phase accumulators; `Positionable`, a mix-in for framestreams that one can position; `IO`, an input/output stream or driver abstraction.

Instances of the `Buffer` class represent multi-channel sample buffers. They have memory pointers to sample storage as well as a set of flags about the storage state (allocated, zero, populated...). `FrameStreams` are objects that can generate buffers of frames where a frame is a collection of samples that are meant to be manipulated simultaneously. `SampleStream` is a `FrameStream` of special importance. It is a one channel `FrameStream` that copies the single channel to all its outputs. The class `Gestalt` has static methods for the sample rate, default buffer size, safe memory allocation. `ThreadedFrameStream` uses a background thread to compute samples. It caches buffers from its producing subgraphs and feeds them to its consumer thread on demand. It controls the scheduling of the producer. This introduces latency but not jitter. When `FrameStreams` and `Processors` need different buffer sizes, a `BlockResizer` object can be placed between two elements of the DSP graph.

An Instrument has a DSP graph, a set of reflective accessors and a list of envelopes. The DSP graph is the instrument's "patch", the accessors define the controls and the envelope list holds the envelopes that need to be triggered to start a new note. Using the instrument/accessor framework one can set a CSL program to respond to commands coming from a variety of sources such as OSC, MIDI, CORBA or score file readers.

There are several versions of the CSL `main()` although in some uses CSL is not even involved in the `main()`. Since CSL is simply a C++ class, it can be used in different ways: incorporate it as a component of another application, use CSL to build plug-ins, build an application with a graphical user interface than controls CSL synthesis and processing.

Different applications have been developed using CSL:

Sensing/Speaking Space is an interactive audio/video installation. A computer vision system analyzes the movement of spectators and sends OSC messages to a sound synthesis server. The first version of the server was written in Supercollider (see 2.6.1.1) but suffered from low reliability, excessive memory usage (1 GB) and poor debuggability. The final version written in CSL was very reliable during a week and sounded just like the first version. In both versions the code is about 1200 lines, including helper classes and a GUI with sliders to mix different layers.

Ouroboros is an application for processing, sampling, and looping audio input and sound files. In this case, CSL is not used for processing, the program hosts AudioUnits, the standard plug-in

format for MacOSX, and lets the users create graphs of AudioUnits to process sound. CSL is used for simplifying the reading and writing of audio files and for capturing and looping the sound. *OndeCorner* is an AudioUnit plug-in written in CSL. It transforms audio to the wavelet domain and lets users modify coefficients with a variety of processes. Apart from being an example of plug-in writing in CSL it also shows how to integrate DSP code from different sources.

The *Reverb plug-in* was developed on a graduate course on spatial sound, when students used CSL to implement reverberation algorithms. It was later used for a convolution-based reverberator and HRTF-based spatializer using the FFTW library.

The *Expert Mastering Assistant* is the largest project using CSL. It is an expert system that uses fine-grained multi-level music analysis to suggest parameters for signal processing to be applied during music mastering. It uses a combination of CSL, AudioUnits and third-party DSP code.

CSL is still in its first stages of development and the authors recognize not feeling particularly comfortable with the C++ language. Nevertheless this first approach is already more important than it may seem. The main author is highly experienced, has designed other related environments (see 2.4.2, for instance) and may be considered an authority in the field. CSL is a clear recognition of two facts: (1) C++ is better suited than Smalltalk for building efficient audio frameworks, and (2) languages like Supercollider (see 2.6.1.1) end-up not being convenient for building some efficient applications (the author presents the framework as a substitute of Supercollider for some particular tasks).

§2.3.3.2 SndObj

SndObj [Lazzarini, 2000b] is an OO sound synthesis and processing programming library released under the GPL designed for the development of music applications as well as research and implementation of DSP algorithms. SndObj is written in C++ and it can be deployed in music software applications as a toolkit or as a framework for developing and implemented new sound processing algorithms. In the first releases SndObj processed on a sample by sample basis but as this proved not efficient, from version 2.0 on it processes vectors (see the following references for an overview of the library's evolution: [Lazzarini, 1998, Lazzarini and Accorsi, 1998, Lazzarini, 2000b, Lazzarini, 2000a, Lazzarini, 2001, www-SndObj,]).

SndObj has three important characteristics: (1) Encapsulation: it encapsulates all the processes involved with production, control, manipulation, storage and performance of audio data. (2) Modularity: processing objects can be freely associated as modules in an analogue synthesizer or unit generators in a computer music system. (3) Portability: the core is portable to any platform with a POSIX compliant C++ compiler. Some classes as real-time IO are platform specific. It has been developed on different

platforms: Sun Sparc under Solaris, IBM RISC 2000 under AIX, SGI O2 under IRIX, and Intel PC under Linux and Windows (using Cygwin and gnu g++). The latest beta version has been released for Windows, Linux and IRIX.

There are four base classes in the framework: `SndObj` for signal processing related objects, `SndIO` for signal input/output objects, Table for mathematical functions and `SndThread` for thread management.

Objects of the `SndObj` class share some properties such as the sampling rate, output vector size, an output buffer, a `SndObj` input object and an on/off switch. They also share methods for the basic operations such as addition, subtraction and multiplication as well as methods for setting and retrieving their basic attributes, this includes retrieving samples from the output buffer. The `SndObj` classes also include a main processing method, `DoProcess()` which is overridable. This is where each derived class implements a particular algorithm. The `SndObj` object will typically access the output signal of an input object and perform the processing that will eventually fill the output buffer. The default `DoProcess()` implemented in the base class just copies from the input object to the output buffer. The `SndObj` base class has a single input but derived classes can have any number (even none as is the case of some generators).

Objects of the `SndIO` class tree are designed to deal with input and output of audio. They implement five basic tasks: standard IO, soundfile IO, digital-to-analog and analog-to-digital IO, buffer memory (RAM) IO and MIDI input. The base `SndIO` class implements a very simple standard IO. The most important methods are the `Read` and `Write` operations. They operate on vectors just like the `SndObj` classes. `SndIO` classes can receive input from `SndObj` objects and can send signal to special `SndObj` classes such as `MidiIn` and `Bend`. Some `SndIO` classes are platform dependent and even have different interface depending on the platform. `SndRTIO` and `SndMidi/SndMidiIn` are only implemented in three platforms (Linux/OSS, SGI and Windows)

Finally, the `SndThread` class encapsulates the main process loop as a separate POSIX thread. Apart from that it offers very small functionality as it is still ongoing work.

The library is distributed with a number of examples that present in a simple way the use of SndObj. Cvoc, for instance, is a simple phase vocoder based on Butterworth filters. There are also some examples of MIDI usage such as Pluck, a Karplus-Strong based plucked-string synthesizer. SndObj can be integrated into GUI frameworks such as MFC and V and there some examples of such functionality.

§2.3.3.3 FORMES

FORMES [Rodet and Cointe, 1984, Rodet and Cointe, 1991] was an interactive system devel-

oped in VLisp first intended for Musical Composition and Synthesis. But due to its architecture and flexibility it found applications in areas such as speech synthesis or graphics animation. FORMES is currently discontinued but its design has influenced the frameworks that have been designed thereafter.

The main goals of FORMES were: generality; universality; independence from a particular synthesis technique; compatibility, models compatible in any context; simplicity of program text; ease of use; modularity; and hierarchical construction.

In FORMES time-dependent objects are called *processes*, built from sub-objects called *offspring*. A FORMES process groups *rules* (procedure bodies), a *monitor* (a kind of scheduler), and *environment* (local variables) and *offspring* (or children). A process has the ability to "sleep", "wakeup", "wait" and "synchronize" when asked.

The role of each process is to ensure the calculation of a particular musical characteristic. This calculation takes place during a precise duration called span, from a begin time (btime) and an end time (etime). These times, though do not need to be explicitly specified. Computation is accomplished through rules in the process environment.

A process is defined by instantiating an original process called generator. New processes can be defined from this original one by deriving from it and adding features. A process may be built from subparts, themselves built from subparts an so on. Therefore a process can have children and a parent. The structure in which processes are organized can be represented as a tree.

FORMES programs and commands can be prepared in files and then loaded or typed on-line. A FORMES program is a structure of processes and the execution involves the repetition of the following two steps: (1) update the list of rules that make the calculation tree and (2) execute or evaluate the rules.

Users communicate with processes by passing or sending messages whenever they want to query a process on its nature, state, or capabilities or they want it to do something. The user of FORMES can compose by connecting already existing objects or by defining new processes. All this is done in Lisp.

A *monitor* defines a temporal control structure. The monitor of a process is the scheduler of its children. The monitor has three tasks: (1) Determine the start time (btime) for each child and start it; (2) Update the calculation tree when the state of the process is modified; (3) If possible, determine the duration of the process

§2.3.3.4 The NeXT Sound Kit

The NeXT computer offered a complete system for manipulating sound and music divided in two "kits" [Jaffe and Boynton, 1991]. The Sound Kit provided object-oriented access to the basic

sound capabilities of the NeXT computer, allowing sound recording, playback, display and editing. The Music Kit provided classes for composing, storing, performing and synthesizing music. It allowed the communication with external synthesizers or the creation of internal software synthesizers.

Both kits were implemented in objective-C. They were mostly independent but could also be used together. The Sound Kit, for instance, could be used to record sound that could then be used in a Music Kit performance.

Although NeXT computers were discontinued after the company was bought by Apple the current Macintosh operating system, OsX, borrows many ideas from NeXTStep and it includes some toolkits such as Core Audio that can be considered as successors of NeXT's initiatives. On the other hand, the Music Kit is still maintained. We will now explain the Sound Kit and leave the Music Kit for the next section on music-oriented environments (see 2.4.1).

The most important class in the Sound Kit is the `Sound` class. It offers an Objective-C wrapper around the data structure that contains raw sound data. `Sound` objects can be instantiated from a sound file, or from the pasteboard, or can be created empty for recording.

The Sound Kit makes extensive use of the virtual memory and interprocess message passing provided by the Mach operating system allowing for efficient manipulation of large sounds. Sound data is rarely moved, is rather mapped into virtual memory. Copying operations employ "copy on write". Reading from a sound file is instantaneous, the data is not brought in from disk until they are required by the application. For storing a sound the Sound Kit uses a file format provided by NeXT. This format lets applications share data.

Playback and recording are performed asynchronously by background threads. The `Sound` class can contain DSP code to be synthesized instead of data, its use is transparent.

Finally, the `SoundView` class provides a mechanism for displaying data in a single Sound object.

§2.4 Music Processing Frameworks

In the previous section we presented a number of frameworks that aimed at offering tools for analyzing, synthesizing and processing audio signals. In this section we will introduce some frameworks that, instead of focusing on the signal, offer tools for processing music at the symbolic level. Note that we have consciously left out some interesting frameworks such as Humdrum [Huron, 1995], Melisma [Temperley, 2004] or POCO [Honing, 1990]. These frameworks focus on the analysis of symbolic music information but never address the signal level. They represent too different of a focus to actually be considered relevant to our study.

§2.4.1 The NeXT MusicKit

The MusicKit is an object-oriented software system for building music, sound, signal processing, and MIDI applications. It has been used in such diverse commercial applications as music sequencers, computer games, and document processors. The MusicKit was the first to unify the MIDI and Music V paradigms.

The NeXt Music Kit was the musical framework of the Next environment and it was the musical counterpart of the Sound Kit just presented in section 2.3.3.4. But unlike the Sound Kit the Music Kit is still maintained and available for different platforms (see [www-MusicKit,]).

The NeXT MusicKit was first demonstrated at the 1988 NeXT product introduction and was bundled in NeXT software releases 1.0 and 2.0. Beginning with NeXT's 3.0 release, the MusicKit was no longer part of the standard NeXT software release but was supported and distributed as Version 4.0 by the Center for Computer Research in Music and Acoustics (CCRMA) of Stanford University. Versions 5.0 to 5.4.1 were then supported by tomandandy music, porting to several more popular operating systems. Currently source code is Freely available for everything, with the exception of the NeXT hardware implementation of the low-level sound and DSP drivers (see [www-MusicKit,]).

Some of the most important features in MusicKit are, according to its original author (see [www-JaffeMusicKit,]):

- Useful for composers writing real-time computer music applications.
- Also useful for programmers writing cross-platform audio/music applications.
- Extensible, high-level object-oriented framework that is a super-set of Music V and MIDI paradigms.
- Written in Objective C and C, using Apple's OpenStep/Cocoa API, the FoundationKit.
- Representation system capable of depicting phrase-level structure such as legato transitions.
- General time management/scheduling mechanism, supporting synchronization to MIDI time code.
- Efficient real-time synthesis and sound processing, including option for quadraphonic sound.
- Complete support for multiple MIDI inputs and outputs.
- Fully-dynamic DSP resource allocation system with dynamic linking and loading, on multiple DSPs.
- Digital sound I/O from the DSP port with support for serial portdevices by all popular vendors.

- Non-real time mode, where the DSP returns data to the application or writes a sound file.
- Suite of applications, including Ensemble an interactive algorithmic composition and performance environment (including a built-in sampler), and ScorePlayer a Scorefile and standard MIDI file player.
- Library of instruments, including FM, wavetable, physical modeling and waveshaping synthesis.
- Library of unit generators for synthesis and sound processing.
- Documentation, programming examples, utilities, including a soundfile mixer, sample rate converter, etc.
- ScoreFile, a textual scripting language for music.
- Connectable audio processing modules or plugins including standard audio effects such as reverb.
- MP3 and Ogg/Vorbis streaming of audio output to web servers using the libshout library.

The Music Kit has tools that address three areas: music representation, performance and synthesis. The goal is to combine the interactive gestural control of MIDI [MMA, 1998] with the precise timbral control of Music V (see 2.6.1). The Music Kit fully accepts MIDI data in any form but is not limited by its specification (for example it has much more resolution in frequency and amplitude).

In its first versions the Music Kit generated sounds by sending synthesis instructions to the NeXT DSP. In its current form the hardware synthesis has been substituted by software based algorithms. But because of its architecture the Music Kit can implement virtually any synthesis strategy.

In the NeXT Music Kit music is represented in a three-level hierarchy of `Score`, `Part` and `Note` objects. A `Score` represents a musical composition, a `Part` corresponds to a particular means of realization. `Parts` are time-sorted collections of `Notes`, each of which contains data that described a musical event. There are methods for rapid insertion, deletion, and lookup of `Notes`.

A `Note` consists of a list of attribute-value pairs called `parameters`, a `NoteType`, a `NoteTag` and a `TimeTag`.

A `Parameter` supplies a value for a particular attribute of a note such as the frequency or amplitude. A parameter value may be simple (integer, real or string) or it may be another object. The `Note` provides methods for setting the value of a parameter as an *Envelope* or a *Wavetable* object. The way a parameter is interpreted depended on the `Instrument` that realized the `Note`. The `Instrument` class defines the protocol for all objects that realized `Notes`. In some way, parameters are similar to

object-oriented messages, the meaning depends on the way the method is implemented in the receiving object.

The noteType and noteTag are used together to help interpret a Note's parameters. There are five noteTypes: NoteDur represents a note with a duration, NoteOn establishes the beginning of a note, NoteOff establishes the end, NoteUpdate represents the middle of the note and Mute is general-purpose. A noteTag is an arbitrary integer used to identify different Notes as parts of a musical phrase or note. (A legato can be created by sending a series of NoteOns, all with the same noteTag). This way the Music Kit solves many of MIDI's problems.

A Note's timeTag, expressed in beats from the beginning of the performance, specifies when the Note is to be performed.

A entire score can be stored in a score file. Score files are in ASCII format and can contain any information that is in a Note. Apart, the Music Kit provides a language called ScoreFile that can be used to add simple programming constructs such as variables, assignments or arithmetic expressions. A score may also be stored in a midifile and utilities are provided for converting to and from standard MIDI file format.

During a Music Kit performance, Note objects are dispatched in time-sorted order to objects that realizes them in some matter. This process involves instances of the classes Performer, Instrument and Conductor. A Performer acquires Notes either from a file, a Score or generating them itself and sent them to one or more instrument. An Instrument receives Notes sent to it by one or more Performers and realizes them in some distinct manner. The Conductor acts as a scheduler ensuring that Notes are transmitted from Performers to Instruments in time-sorted order at the right time. Both Performer and Instrument are abstract superclasses and the Music Kit offers subclasses such as SynthInstrument, MidiOut or ScoreRecorder.

In order to generate sound from musical data the Music Kit uses three main classes: SynthElement, SynthPatch and SynthInstrument. SynthElements are the basic building blocks and they correspond either to code, through the UnitGenerator subclass, or to data, through the SynthData subclass. A SynthPatch is the configuration of SynthElements that define a synthesis strategy and it is analogous to a voice or instrument setting in a regular synthesizer. Finally the SynthInstrument is a subclass of Instrument that realizes Notes by assigning them to particular SynthPatches.

§2.4.2 MODE and Siren

The MODE [Pope, 1991c, Pope, 1994, Pope, 1991b] was a collection of OO classes for general

sound, event, event list and score processing as well as a music oriented user interface tool kit, embedded in the Smalltalk-80 Programming System. The MODE was substituted by Siren [Pope, 2001, Pope, 1998a, Pope, 2003, www-Siren,] in 1998.

Both frameworks are the result of the author's, Stephen Travis Pope, continuous iterations in order to find a tool for his compositions as well as a platform for practically demonstrating his research on object-oriented programming and software engineering[Pope, 1991d]. As such, they implement a quite particular vision of musical composition that is tightly integrated with an object-oriented model[Pope, 1991b, Pope, 1997]. Both frameworks have been developed taking into account very little inputs from users and as the author observes "(...) if Siren works well for other composers, it is because of its idiosyncratic approach, rather than its attempted generality"[Pope, 2001].

Before taking a look at the MODE and Siren, let us summarize the different packages and versions the author has worked on.

The MODE was already the result of several iterations of Smalltalk-80 based toolkits for musical score and sound processing and performance. It was itself a reimplementation of the author's earlier package, the HyperScore ToolKit [Pope, 1987], son of DoubleTalk, son of ARA. ARA was a Lisp system. Double-Talk was a Petri net editing system in Smalltalk-80.

Siren is a software framework that includes a set of flexible and reusable components that are designed for extension and customization but also carries with it a "way of thinking" about music and composition. It has been developed in the Smalltalk language and it is intended to be used by Smalltalk-80 programmers. Previous systems strove for extreme flexibility at the expense of additional complexity but Siren makes decisions differently.

The system is designed to accept pluggable front ends and back ends. It is efficient for real-time composition and portable as it runs in several OS. Application areas are: sound and score editors, real-time algorithmic composition, and music performance front ends. The purpose of the framework is to provide comprehensive note, score and sound processing tools for the rapid prototyping of music applications. The framework includes an abstract music representation language, an interface for real-time I/O, a user interface framework, and connection to object databases. It is also integrated with a scalable distributed processing framework.

Siren is a software framework for sound and music composition and production made of about 350 Smalltalk classes. It is platform independent and runs on Macintosh, Windows, and Unix-based computers. The Smalltalk code is available for free.

The motivation behind the MODE and now Siren was to build a powerful, flexible, and portable computer-based composer's tool and instrument. Siren is designed to support composition, off-line

realization, and live performance. Other desired applications are music databases, music analysis and music scholarship and pedagogy.

On the other hand, the technical goal is to present good OO design principles and elegant state-of-the-art software engineering practice. It needs to be easily extensible, to provide abstract models of high-level musical constructs and flexible management of large datasets.

The main components or packages in Siren are:

- The Smoke music representation language (classes for music magnitudes, events, event list, generators, functions and sounds).
- voices, schedules and I/O drivers (real-time and file-based I/O for sound, OSC, and MIDI).
- user interface components for musical applications (tools and music/sound widget).
- several built-in applications (editors and browsers for Siren objects).

It is possible to use inheritance for building specialized versions of existing components.

Smoke [Pope, 1992] is the "kernel" of Siren. It is a set of classes organized in meta-categories such as Magnitudes, Events and EventLists, Schedulers, or Interfaces. Smoke is described in terms of two description languages: a compact binary interchange format and a mapping onto concrete data structures.

According to the author Smoke can be summarized as follows: Music can be represented as a series of *events*. Events are simply property lists or dictionaries that can have named properties with arbitrary values. These properties may be music-specific objects and for that reason models of many common musical magnitudes are provided.

Music Magnitudes are extensible abstract representations for the properties of musical events such as pitch, duration and loudness. Each Music Magnitude can have different representations (e.g. pitch can be represented in integer, float, string or fraction). Their primary behavior is that they can translate freely between their representations. `MusicMagnitude` objects are characterized by their *identity*, *class*, *species* and *values* (e.g. the pitch object representing the note C3 is a member of the class `SymbolicPitch`, of the species Pitch and has a value of c3 (note that class+species allows for multiple inheritance).

The basic abstract model classes are `Pitch`, `Loudness` and `Duration`. They are abstract and have no subclasses, they are used by species for families of classes.

The basic event classes, `Event` and `EventList` both of which derive from `AbstractEvent`, are used for describing musical structures. In Smoke, an event is simply an object that has a duration

and possibly arbitrary other properties. The `AbstractEvent` in Smoke is modeled as a property-list dictionary with a duration. There is no prescribed grain size or level for events.

`EventList` hold collections of events sorted by start time. Event lists are events in themselves and can therefore be nested into trees in a hierarchical structure. An event can be in more than one list at different relative start times and with different properties mapped into it. `Events` don't know their start-time, which is always relative to some outer scope.

`Events` and `EventLists` are "performed" by the action of a scheduler that passes them to an interpretation object or *Voice*. Voices map event properties onto IO parameters.

`NoteEvent` classes are like generic `Events` that represent musical notes with the default parameters pitch, amplitude and voice. Links between events and event lists can have some symbolic description (e.g. `isVariationOf`, `isTonalAnswerTo`...)

Sampled sounds can be properties of events. The `Sound` class allows reading and writing a number of file formats and maintains a list of named cue points in the sound.

Siren has classes for representing "middle-level" structures e.g. cluster, chord, ostinato or rubato. Music formats can be characterized in a very compact way. Two abstract classes are defined: `EventGenerator` and `EventModifier`. Composers can enrich the generator hierarchy for a specific composition.

`EventGenerators` can either return an `EventList` or behave like processes and be told to play and stop. The three abstract `EventGenerators` are `Cluster`, `Cloud` and `Ostinato`. `Cluster` classes describe a one-dimensional collection of pitches or rhythms (their events occur simultaneously or are repetitions of the same event). Concrete types of `Clusters` are chords and arpeggi. `Cloud` classes are random generators that produce notes from a given range. Most process-oriented generators take the form of `Ostinati`, which create repeating versions or variations of the input material or parameters.

The `EventModifier` class models objects that have a function object and a property name so they can apply the function to the given property of an `Event` or an `EventList`. `EventModifiers` can be lazy or eager. Eager `EventModifiers` apply the function as soon as they are given an `EventList` while lazy wait until scheduling time.

Using the messages to the previous basic classes one can make scripts (programs) of messages to `Events`, `EventLists` and `Functions` to describe simple musical processes. Regular messages from the Smalltalk-80 environment can be used to inspect objects.

Siren has a special structure that allows the same score to be played independently of the final synthesis method. Properties of events are encoded in an abstract symbolic way that is then expanded into device-specific or output format-specific parameters.

Siren also has a complete graphical environment that can be used to develop graphical applications for music processing.

One of the basic problems for making cross-platform music tools was the lack of good portable APIs for sound and MIDI I/O. This has been helped recently by cross-platform libraries such PortAudio [Bencina and Burk, 2001], PortMIDI [www-PortMIDI,] or LibSndFile [www-libsndfile,]. All of these libraries are implemented in C/C++ and it was difficult to integrate them into Smalltalk. But VisualWorks has a powerful for interfacing Smalltalk code to C libraries.

For network and file-oriented IO, it uses Open Sound Control (OSC)[Wright, 1998a]. Siren has also been used as a front-end to CSL (see 2.3.3.1) through OSC messages.

The author of Siren has also implement the Create Real-Time Application Manager (CRAM) [Pope et al., 2001] for large-scale distributed processing. Siren and CSL are designed to be used in distributed systems controlled by CORBA and with messages sent through OSC.

Also recently new class libraries have been added to support using large speech databases with phoneme segmentation and detailed feature extraction. The analysis core of the Siren speech database is the Segmenter, which uses a combination of time-domain and spectral-domain features to break continuous speech into phonemes.

Although as already commented most of the applications developed with the Siren/MODE framework are musical composition environments, the frameworks is sufficiently flexible so as to be used in different situations. Paleo, for instance, is a suite of sound and music analysis tools integrated with an OO persistence mechanism in Siren. Paleo uses dynamic feature vectors and on-demand indexing. Annotational information derived from analysis can be added to the database at any time. Paleo performs analysis of MIDI files and allows for complex queries.

Most of the advantages and disadvantages of Siren are related to its language of choice, Smalltalk. These are, according to its author the most important advantages of Smalltalk and therefore Siren:

Smalltalk is a simple programming language; the class library is quite compact and extense especially when compared to C++; Smalltalk has an extensive development environment with code browsers or in-place debugger; finally, it is important to point out that the language, libraries and IDE have been quite stable for the last 20 years.

As disadvantages he cites the following: Smalltalk now is not a mainstream language; the VisualWorks/Smalltalk implementation is large (2000 classes + 300 Siren classes), this is a very large system to learn; like in Java, Smalltalk programs are generally compiled to a virtual machine which may be interpreted, translated or cross-compiled at run-time, this provides cross-platform portability of

object code but at the cost of some run-time performance; garbage collection also makes development easier but also adds overhead; finally the Siren package itself is complex and implements a very particular design approaches. It does not include MIDI sequencing or common music notation editors due to lack of interest by the authors. They do with Siren what they cannot do with a combination of SuperCollider, Peak, Finale and ProTools.

New applications in different areas are planned for Siren [Pope, 2003] such as controlling graphical animation from Siren. But probably the most surprising news is that, after 20 years of Smalltalk development, the author is thinking on changing to a different language. At the moment of this writing, they are experimenting with Ruby, Self, and Supercollider. Porting Siren to another language means keeping Smoke's class library specification but abandoning its syntax that is too Smalltalk-oriented. Smoke event list are then written in the language as the implementation being used.

§2.4.3 Common Music

Common Music [Taube, 1990, Taube, 1998] is an object-oriented musical composition interface that bases its results in the definition of classes and objects that interact between them at the program level. CM is Free and it is updated regularly.

Common Music treats the composition process as an experimental process aimed at describing the sound and its higher-level structure. It has a number of composition tools as well as a public interface. It can be used in conjunction with many different existing protocols and languages such as MIDI, CSound, Common Lisp Music or Music Kit.

Common Music is implemented in Common Lisp and CLOS and runs on different platforms including PC, Macintosh, SGI, NeXT and SUN. Source code is publicly available.

Common Music structure is based on the separation of three different levels in the music composition process. On one level, the composer concentrates on developing musical ideas. On another level, the composer worries about how these ideas can be translated to the real world. An on the last level, the musician must understand how the composer's ideas have to be conceived. This separation allows the re-use of common higher level structures.

Common Music provides three different interaction modes that can act in parallel. The most basic way to interact with the system is through the program Lisp source code. This mode of interaction offers much flexibility to control algorithms but ignores many of the system utilities. The second mode implies working through the command interpreter, which translates text messages into system actions. The command interpreter in Common Music is called Stella. The main advantage of this mode is that

it allows for flexible edition but, on the downside, the creation of complex commands results into very complex messages. And finally, the third interaction mode is through the use of a graphical interface called Capella that is only available for the Macintosh platform.

Common Music defines six kinds of collections, which can be specialized or enhanced by the user. The existing collections are: `Thread`, `Merge`, `Heap`, `Algorithm`, `Network`, and `Layout`. A `Thread` represents a line of events to be sequentially processed. A `Merge` represents parallel or multiple temporal command lines. A `Heap` is a collection that represents a random grouping that is converted into a temporal line after mixing its substructure and then processing it sequentially like a `Thread`. An `Algorithm` represents a program description, that is the temporal line of events is produced by calling a user-specified program. A `Network` is a collection that represents a user-defined order, the temporal line of events is produced by the call to a condition that can be expressed through a pattern or a function. Finally, a `Layout` refers to arbitrary chunks of an existing structure.

The act of translating the high-level information introduced by the user into a lower-level information understandable by the synthesizer is known as *realization*. This realization can happen at different levels and through two working modes: *run-time* and *real-time*. In the first mode, events receive a time tag but the clock advances as fast as it can. In the second mode the time tag is exactly that of the real time.

In Common Music, the final result depends both in the events and in the context. This context is known as *stream* and can include outputs such as Postscript or sound files or MIDI ports. All these different streams are controlled by the protocol that is in charge of taking the right actions depending on the current combination of events and stream.

Common Music has different procedures and classes that are designed to give support to the musical composition process. These tools include high-level macros to create musical structures, musical data representation or envelopes. The user can combine these functions with the functionality included in the Common Lisp programming language.

§2.5 Audio and Music Visual Languages and Applications

After having seen quite a few examples of audio and music software frameworks in the previous two sections we will now introduce some environments that fall in the category of *visual languages or applications*. Most of the authors of these environments would argue that they are also frameworks. And as a matter of fact, according to the framework classification we introduced in section 1.3 at first sight we might be tempted to classify them as software frameworks that have evolved to become visual

builders. The main reason why we think that they should not be classified as software frameworks is that they do not grant access to the source code of the engine in a natural way and the extension capabilities they offer are very limited. Note that some of the frameworks presented in the previous sections (such as OSW in section 2.3.2.2) also provide a visual builder, but their underlying philosophy is clearly a different one.

§2.5.1 Kyma

Kyma [Scaletti, 1991, Scaletti and Johnson, 1988, Scaletti, 2002] is a visual sound design language that was started by Carla Scaletti at the Illinois University at Urbana Champaign and is now commercialized by the authors through Symbolic Sound Corporation [www-SymbolicSound,]. Although Kyma is much more than a graphical application and it contains a complete conceptual model very much in the line of the already commented MODE or Siren, it is best known and only used as a graphical application, and for that reason it has been included in this category, even at the risk of disappointing the author who clearly states that Kyma would still be Kyma without the graphical interface[Scaletti, 2002] but also qualifies it as a "visual language" [www-SymbolicSound,].

Kyma is an object-oriented music composition and sound synthesis environment written in Smalltalk-80 that can be used with a microprogrammable digital signal processor called Capybara. According to the authors [www-SymbolicSound,] Kyma is being used to do sound design for music, film, advertising, television, virtual environments, speech and hearing research, computer games, and other virtual environments. Although this may be so, the truth is that Kyma has been designed and it is mostly used as a music composition tool.

The first version of Kyma was started in 1986 on a Macintosh. In 1987 it was modified to make use of the Platypus signal processor. In 1989 they decided it made no sense to keep the project in the university so they founded a company, Symbolic Sound Corporation, that started operating on their student apartment and moved to their first office in 1992. Since the first version of Kyma in 1986, there have been five major software revisions, a port from MacOS to Windows in 1992, and ports to five different hardware accelerators including the currently supported Capybara. Currently Kyma runs either on a Macintosh or PC and on the Capybara, a general-purpose multiprocessor computer that can support from 4 to 28 parallel processors.

Kyma provides ways to create and manipulate sound objects graphically with real-time sonic feedback via software synthesis. In order to offer real-time synthesis without having to compromise flexibility, the authors decided for a specialized hardware DSP for being both efficient and programmable.

But, as the authors acknowledge, the main reason for using the Platypus platform was that in the computer with Smalltalk they couldn't even get 20k samples per second.

When designing Kyma, the authors intentionally kept away from Music-N languages and music notation based systems [Scaletti and Johnson, 1988]. According to them the main reasons that on the one hand, Music-N languages seldom offer immediate feedback and can be frustrating for composers trying to experiment new sounds and they make it difficult to control higher level-aspects of music such as phrases or lower-level features such as those related to timbre, which have to be controlled from a different file. And on the other hand, environments based on music notation do not offer enough flexibility as an acoustic event cannot be fully specified with traditional music notation.

In Kyma there is no clear distinction between Instrument and Score. Everything in Kyma, from a single timbre to the structure of the whole composition is a *Sound Object*. The sound object in Kyma is inspired on the *objet sonore* of Pierre Schaeffer [Schaeffer, 1966]. A Sound Object can be manipulated, transformed, and combined into new Sound Objects. Objects that were encapsulated into another object can be brought back to the top level object, and top level objects can be combined and hidden in a yet higher-level object. Sound objects are uniform so any given Sound object can be substituted for any other Sound object.

A Kyma Sound Object can be either a `SoundAtom` or a `Transform`. While a `SoundAtom` is a regular sound or collection of samples, a `Transform` is the sound result of applying a given function to its *subsounds*. In this sense, a Sound in Kyma is represented as a directed acyclic graph (DAG) with a single root node. Each edge in the graph represents the relation "is a function of". A subsound can be shared among several `superSounds`. A Sound DAG is similar to an expression tree in that the evaluation of the higher nodes depends on the result of the lower nodes.

In order to hear a Sound object it is necessary to "evaluate" it, that is, convert it to a sample stream. Every sound object knows how to compute its next sample. The nodes of the DAG are evaluated in post-order. When a Sound object DAG includes Delay nodes the DAG is first expanded into a series of time-tagged DAG's.

A `SoundAtom` has no `subSounds`. For instance a `LiveSound` is defined as the input from the analog-to-digital converter. A `PotentialSound` is one that does not respond directly to a `Play` message. When it receives such a message it creates an new `Sound` from its `Subsound` and then sends it the `play` message. A `PotentialSound` node expands into a subgraph during evaluation. An interesting example of `PotentialSound` is the `FrequencyTransform`, it contains as an attribute a function of time, frequency and the duration of its subsound.

A Kyma `Lifted` object is a `Sound` object with variables. A lifted object can work as an

"instrument" that is instantiated and scheduled from a score as in the Music-N style languages. In Kyma a `MusicN` is a `Sound` object whose parameters include a score and a collection of subsounds to be used as instruments. Any variable parameter of a subsound can be set from the score. In the score language an event is specified as the subsound name, a start time and any number of (parameter, value) pairs. Parameter values can also be subsounds.

A Kyma `Transform` can be unary or N-ary. An N-ary `Transform` has an `OrderedCollection` of sounds (named `subSounds`). The primary N-ary transform are `Mixer`, `Concatenation` and `Multiplier`. On the other hand, an unary transform has a single sound in its attribute `subSounds` (e.g. Delayed, Amplitude Scaled...).

It takes a finite amount of time to compute a sound object, therefore, real-time cannot be guaranteed. Kyma attacks this problem in two ways: one option is to lower the sample rate and try again, the other is to allow the output samples to be stored on disk (the sample file on the disk can be treated just like any other object).

The conceptual object-oriented model in Kyma is a central issue and it is in fact the only thing that has not changed since its first versions. According to the authors, this demonstrates its flexibility and how well it suits the creative process. For example, in the first versions although structures could be time-varying, parameter values were constant. But they realized that making parameters event-driven the language would be more interactive and would open to external control such as MIDI. In version 4.5 they added an event language. The improvement on the quality of the new sounds demonstrated it had been a good decision. But the original data structure was robust enough to accommodate to this major shift.

One of the ideas behind Kyma is that one can easily plug new algorithms. Kyma has proven open enough to accommodate new algorithms that the authors have developed during the years. It is important to note that Kyma is not designed to allow the user to extend the environment by adding new objects and does not offer access to its source code. This is one of the reasons we cannot classify as a framework.

But apart from its conceptual model, Kyma has an extensive and flexible user interface. The current graphical representation has evolved over the years. The abstract structure came first, and the graphics evolved in order to represent the structure.

In the first versions a Sound was specified as Smalltalk code. This quickly evolved into a "selection-from-list" interface. In order to reduce the need for typing, the interface was changed to the "Russian Doll" style where Sounds were represented as containers of other sounds. Double clicking opened a new window showing this sound. This interface was good at reveling the recursiveness but

not the overall structure. Then it was changed to the box interconnection paradigm ala Max with flow direction from top to bottom, representing a DAG (directed acyclic graph). Later the direction was changed from left to right. Although the underlying structures did not change, changing something as simple as the direction made people understand the structure differently and therefore create differently. Apart from the central structural representation, Kyma also includes a timeline representation.

§2.5.2 Max

Max [Puckette, 1991a, Puckette, 2002, Zicarelli, 2002], together with PD and jMax that are particular instances of the same model, is, apart from CSound (see section 2.6.1.2), probably the most widely used computer music environment and with one of the longest development life. Max is, according to its author Miller Puckette, "a graphical programming environment for developing real-time musical applications" [Puckette, 1991a]. Max was first written for Macintosh, then ported to NeXT workstations and it is now available in any of its forms for most operating systems and platforms. At the time Max was proposed, graphical applications existed but they did not address the real-time issue. On the other hand, other systems that had advanced real-time strategies did not have graphical interface. Max was born as a compromise in between both approaches.

Although Max has been classified as a graphical language/application, its definition is a matter of controversy. According to the author Max was not intended to be a programming language [Puckette, 1996]. On the other hand, in [Déchelle, 2000] we find that Max can be seen as "an imperative graphical programming language" or a "graphical interface generator". Our opinion is that Max is not more than a graphical application although its graphical model has had so much success that it has almost given place to the Max paradigm. As a matter of fact Max can now be seen as a conceptual model that has had and still has many different implementations, before explaining any of them it is interesting to understand its development history.

Max's predecessor was called the Patcher and was presented by Miller Puckette in 1988 [Puckette, 1988]. The Patcher was a Macintosh software for treating and controlling MIDI. The Patcher was acquired by the Opcode American company where David Zicarelli added many new features and enhancements and converted into the popular Max/Opcode.

In 1989 Ircam started the Ircam Sound Processing Workstation (ISPW) project in order to develop a complete workstation for audio and music working on a NeXTSTEP environment. This project enabled to develop a new version of the Patcher known as Max/ISPW. The main novelty of this version was its client server architecture: on one hand the graphical interface and on the other a

real-time sound processing environment called FTS.

Once the ISPW started to evolve, it became clear that Max/ISPW had to become more portable to different platforms. For that reason a new version was written called Max/FTS. The version was distributed for Silicon Graphics computers.

Miller Puckette started about the same time with the development of the free PD environment [Puckette, 1996, Puckette, 1997a, Puckette, 1997b] in order to give answer to some Max limitations, namely its lack of support for dynamic data structures. PD used many of the results of the Animal project and introduced a portable graphical interface based on Tcl/Tk. In 1997, he enhanced PD's audio processing module and named it MSP (Max Signal Processing). MSP was added to Max/Opcode, introducing therefore real-time audio processing.

The graphical interface in Max/FTS was re-implemented in Java and that gave rise to jMax, a new implementation of the Max language. At the moment of this writing the Java interface in jMax is being re-written in Python as it depended on some proprietary libraries.

Although from the beginning Max was intended to be a unified environment both for signal and control flow, it was historically developed as a MIDI program. Later on, Max was extended to include signal processing capabilities [Puckette, 1991a], these capabilities are now in Max through the MSP extension [Dobrian et al., 2000].

It is beyond our scope to give a complete view of Max and related environments. For this reason we will mostly concentrate in its signal processing capabilities and on PD, its free version.

The most important issue in MAX is its graphical model. The fundamental item is the *patch*, a collection of boxes connected by lines. A patch can either be in "run" or "edit" mode. The boxes represent *objects* that wait for messages to be passed to them. They may respond by taking an action or by passing messages to other boxes. Boxes may have *inlets* and *outlets*, which are represented as dark rectangles. Lines connect inlets to outlets and a message passed to an outlet is transferred to all inlets connected to it. The messages are an ordered list of atoms, each of which may be a number or a symbol.

Apart from sending/receiving messages through the outlets/inlets, objects can access the clock and the MIDI I/O. The clock is accessed with a callback mechanism. In order to receive input MIDI messages the object must be inserted in the MIDI callback list, output MIDI messages are sent by calling a library function.

By convention, if an object has more than one inlet, its leftmost one is the "active" one. Passing a message to it causes something to happen, passing it to the other inlets just changes the state of the object. In a similar way, output messages are always written to outlets from right to left.

Before MSP was introduced signal processing in MAX was carried out by a collection of *tilde* classes implemented in the FTS engine. These objects communicated through inlets and outlets using the *signal* message. But the calculations on signals required more communication bandwidth than the message passing mechanism can offer [Puckette, 1991b]. For that reason they used a special scheduling mechanism based on a duty cycle that is carried out regularly to compute a new set of output signals.

Tilde objects executed processing tasks on fixed-size vectors typically between 16 and 64 samples. The DSP computation period was set to the sampling rate divided by the buffer size. Tilde objects intercommunicated at setup time in order to determine a calling order and the addresses of input/output signals to be used. This communication was done through the signal message using regular inlets/outlets. The signal message took two integers: a selector (`COUNTINPUTS`, `COUNTOUTPUTS` or `DOIT`) and the address of a signal object. DSP objects cannot initiate messages when in the duty cycle. This may introduce a small but non-zero delay between message and actual processing. Tilde classes communicated with control objects through their member attributes.

For building the duty cycle call list the `signal` message was used. The signal message acted like a token used by tilde objects to simulate data-driven dataflow networks. Each tilde object could "run" only when it had data in all its inputs and then produced into all its outputs. Each time a tilde object was run, it appended itself to the call list.

When a dac object received the `start` message, it traversed the list of all tilde objects causing them to send the `COUNTINPUTS` message to all its outlets. Each inlet also counted the number of outputs connected to it. The list was traversed a second time and any tilde object that had no inlets or whose inlets count to zero was added to the call list. A change in the network is reflected as a change in the DSP duty cycle call list.

Each time a tilde object was added to the list, new signals were allocated for all its outlets and the outlets were first passed a `COUNTOUTPUTS` message and then a DOIT message. The first message was simply passed to count the number of inlets connected to each inlet. The `DOIT` message passed the address of the newly allocated signal and informed the patch that the signal was ready to be used. Therefore when a tilde object received a `DOIT` message it could determine whether all its inputs were available. After decrementing the inlet's count, if all inlets count were set to zero the object could be added to the call list, knowing already the addresses of all its inputs and outputs.

The signal allocated for a signal output could be freed as soon as the last tilde object having it as input was put on the call list. This way, a chain of tilde objects each with one input and one output would typically use the same signal inplace (important for processors with limited memory).

If at the end of the call list building process, a tilde object was not on the list, it meant that a

signal loop had been detected. This was an unwanted situation as if a signal loop was actually needed a delay read/write pair had to be used (set to the minimum of one duty cycle).

Tilde objects carried out their DSP actions synchronously so execution order of inputs and outputs did not matter. But control parts of a patch could be activated in different ways yielding different results.

While the previous explanation is basically about Miller Puckette's design, David Zicarelli started working in parallel with another implementation of Max signal processing capabilities. This new design finally gave place to MSP (Max Signal Processing). MSP is now a basic extension of Max, up to the point that the current environment is known as Max/MSP.

MSP [Dobrian et al., 2000] includes over 170 Max objects for digital signal processing. In the graphical environment MSP objects are very similar to Max objects with the only difference that their name ends with a '~'. MSP objects are connected the same way than Max objects but intercommunication is conceptually different. Instead of establishing a path for messages, MSP connections establish a relationship between objects and that relationship is used to calculate the audio information necessary at a particular instant. The configuration of MSP objects is known as the *signal network*.

MSP are in constant communication. Max objects sit idle waiting for a message to occur, but MSP objects are always active, constantly computing the current output samples. For that reason, an MSP signal network can be understood as a portion of a patch that runs at a faster (audio) rate than Max. Max (and thus the user) can only affect the signal portion of the patch every millisecond. What happens in between those milliseconds is calculated and performed by MSP.

Some MSP objects provide a link between Max and MSP and to translate between control rate and audio rate. MSP inlets can accept both signal and Max messages (e.g. they can be turned on and off with the Max messages `start` and `stop`). A Max patch can contain both Max and MSP objects. Nevertheless, for organization, MSP objects in a signal network are often encapsulated in a subpatch.

Pure Data (or Pd for short) [Puckette, 1996, Puckette, 1997a, Puckette, 1997b, Puckette, 2004, www-PD,] is a real-time graphical programming environment for audio and graphical processing. It is very similar to the Max/MSP system but is simpler and more portable. Pd also has two basic features that are not available in Max/MSP: first, via the GEM package, Pd can be used for simultaneous computer graphics animation and computer audio; second, an experimental facility is provided for defining and accessing data structures.

Pd is an effort to solve some problems in Max while keeping its strength. Pd was designed by Max's author Miller Puckette and although he mentions that it was not his intention to make a Max clone he recognizes that whenever there was no real reason to make something different, the solution

available in Max was used [Puckette, 2004]. In parallel the GEM [Danks, 1997] project started to develop a real-time graphical synthesis/processing/rendering environment which would run with Pd.

The main weakness in Max are reported to be its difficulty of maintaining compound data structures and of integrating non-audio signals like video or audio spectra [Puckette, 1996]. In order to use Max to process data, the project Animal started [Lindermann, 1991]. Many ideas in PD come from this program. The first thing introduced in PD was the ability to plot graphics and figures.

The main goals when starting the Pd and GEM projects are summarized in [Puckette, 1997b]: (1) a real-time patchable environment ala Max: (2) management of audio and image processing in the same environment; (3) adaptability to a wide range of platforms; (4) long-term stability; (5) newer and more flexible set of tools to manipulate data.

In Pd, any data structure (called "Pure Datum") can become a message handled the usual way. Also, users may create their own "DSP blocks" in which sample rate and vector size vary.

The `clock` allows users to attach a symbolic name to a specific combination of sample rate and vector size. Any `clock` can be turned on/off. The `reclock` object simply converts any signal to the desired `clock` (for instance, preparing an input signal for overlapped FFT; the overlap-add process in the synthesis is accomplished by simply reclocking to the original clock).

Pd is designed in two parts: the "real" Pd and the Pd-gui. The real Pd, does real-time computations using a Max-like interpreter and scheduler. The Pd-GUI talks to the computer window system through the tk toolkit.

Pd shares with MSP a few objects for audio analysis [Puckette et al., 1998]. The `fiddle` and `bonk` objects, for instance, are two basic implementations, the former for pitch detection and the latter for bounded-Q analysis. Their main goal is to get predictable and acceptable behavior with easy-to-understand techniques that will not place a heavy load on the machine. The output of both objects appears as Max-style control messages.

`Fiddle` is a maximum-likelihood pitch detector similar that can also be used to obtain a raw list of sinusoidal components. `Bonk` performs a bounded-Q analysis of the input signal to obtain onsets of percussion instruments.

Finally jMax [Déchelle et al., 1998, Déchelle et al., 1999b, Déchelle et al., 1999a, Déchelle, 2000, Déchelle et al., 2000, Déchelle and Tisserand, 2003] is a new implementation of Max/FTS in which the graphical interface is re-implemented in Java. jMax reuses Max/FTS and PD's two component architecture (client/server). This architecture allows decoupling the two components and executing the processing component independently from the user interface (for example inside a VST plug-in). Communication between the two components is done through TCP/IP or UDP.

jMax's interface offers basically the same functionalities than the other Max versions: a patch editor for constructing and controlling program patches and a set of specialized editors for objects with complex data structures (such as tables or sequences).

jMax introduces a textual scripting language for controlling structures that were difficult to represent graphically such as operator banks and other repetitive operations. jMax used Tcl scripting. But because of integration problems of Tcl and Java it was decided to change to Scheme (a Lisp dialect). Currently, and because of some problems with proprietary Java libraries, the interface is being ported to Python.

In any case jMax is available for Silicon Graphics over the IRIX system, PC or Macintosh over Linux (using either OSS or ALSA sound devices).

§2.5.3 WaveWarp

WaveWarp[Jafry, 2000, www-SoundsLogical,] is a modular real-time PC-based audio processing software tool intended for use by audio effects developers, signal processing engineers, musicians, and educators. It is a commercial application available only for the Windows operating system.

Because of its targeted uses, in WaveWarp flexibility is essential. Because of this, it includes an interface to Matlab. Through the Matlab interface users can create their own components. It also allows to create DirectX plugins

WaveWarp has a modular architecture that includes precompiled DSP components which can be connected in any desired fashion (series, parallel, feedforward, and feedback). All components are processed sample-by-sample. The audio engine is multi-rate, enabling on-the-fly integer-factor sample-rate conversion between components. The software architecture is multi-channel enabling "surround sounds". It runs real-time on a standard Pentium PC over Windows.

WaveWarp includes more than 250 components divided into the following categories: Audio Files, I/O devices, Basic Connections (summer, multiplier, arithmetic operators...), Delays, Digital Filters, Displays and scopes, Distortions, Dynamic range controllers, Flangers and chorus, MATLAB, Mixers, Multirate, Noise reduction, Panners, Phasers, Pitch shifters, Reverbs, Signal generators, and Spectral transformers.

WaveWarp also includes many sample patches implementing things like: reverberation based on a random FIR filter, MATLAB-in-the-loop signal processing effects, controllable audio playback and granular synthesis, educational demonstration of aliasing, or four octave band equalizer.

§2.5.4 OpenMusic

OpenMusic [Agon and Assayag, 2002, Assayag and Agon, 2000, Déchelle, 2003] is an object-oriented visual environment for musical composition based on Common Lisp. It represents a completely different approach from other applications in this category as it is restricted only to the symbolic manipulation of musical material and problems such as performance and real-time processing are outside of its scope. OpenMusic deals with models and proposes a composition based on structures and relations. OpenMusic is available for most operating systems [Sarria and Diago, 2003].

OpenMusic is a continuation of Patchwork. This environment was used and valued by many composers but several limitations evidenced that it needed a major rework. First, Patchwork was the result of numerous incremental improvements and it was difficult to maintain. Also, Patchwork was centered around the patch editor window where functional modules could be connected to perform any calculation. These functions were made available as standard kernel boxes, as standard Common Lisp functions, as parts of user-libraries, or could be directly programmed by the user in Common Lisp. But the main problem was that it was extremely difficult to collect information from one patch to use in another.

OpenMusic aims at improving the patches relationships, to dynamically create object definitions, to facilitate communication between different forms of the same musical structure and to place all this into an easy editable temporal context. The other problem of relations between patches is solved with the Maquette Editor (see below).

A program (patch) is a graphical layout on the screen. This layout is made of *simple frames* or *composed frames,* which contain simple frames or are empty. All the computing objects in OpenMusic are represented as simple or composed frames and several different frames can be produced for the same object. A simple frame representing an object is called *object view* and generally appear as icons. Composed frames representing an object are called *object containers* and generally provide a graphical editor for the object. The container for a class is an ordered collection of simple frames representing *slots*. Slots have information about their name, type, a default value, and a flag that indicates if the slot is public or private.

Users can create instances of a class with the aid of a particular box called a factory. They can also create new methods or modify existing ones. In the same way users can create and redefine their own classes.

Instead of a simple graphical interface to CLOS, OpenMusic is regarded as an extension of CLOS with metaprogramming techniques. A protocol of generic functions applicable to all OM objects

has been defined, some of them to relate classes and their graphical representation. By using metaprogramming the user can also make extensions to the OpenMusic language. The following tools are available: subclassing inside the static class definition and redefining functions in the dynamic protocol.

Open Music contains a simple musical class hierarchy. `Note`, `Rest`, `Sound` and `Midifile` derive from `Simple-score-element`. `Voice`, `Measure` and `Group` derive from `Sequence`. And `Chord` and `Poly` derive from `Superposition`.

A set of musical operators are also defined. They work on musical structures such as voices. For instance the operator `fusion` merges two different voices and the result is a voice that contains simultaneously the notes in both initial structures. The `masking` operator also operates on two voices and performs a masking effect on one of them (data voice) using the other (masking voice) as the mask.

The Maquette is an interface that allows the creation of blocks placed in spatial and/or temporal relationships. These blocks are linked to patches. The Maquette is an original concept in OM used for combining the design of high-level musical structures, arrange musical material in time, and specify musical algorithms. A Maquette editor is an editor with 2D surface with time on the x-axis on which temporal boxes are laid. Temporal boxes can reference a temporal object (object with start time and duration such as chord or voice), patches (which are not temporal objects but can deliver one as result), and another Maquette (a Maquette can be embedded in another Maquette). The user can choose to see the Maquette as a score (in traditional or graphical notation) or a set of interconnected processes.

It is also interesting to note that OpenMusic has been used as a control language for sound synthesis [Agon et al., 2000]. In this case the basic idea was not to implement synthesis engines but to handle musical structures visually and then generate low-level parameters to send to whatever engine is available.

§2.6 Music Languages

A language can be defined as a communication framework that defines a particular syntax, grammar and vocabulary. According to this definition, many of the environments presented until now could be classified as music languages. As a matter of fact, the previous section includes music graphical programming languages.

In this section though, we use the term music language in a restrictive manner, to define the category of music environments (or languages) that do not offer a graphical environment or do not qualify for being software frameworks.

We have separated this category of environments into two subcategories: Music-N Languages and Score Languages. In the first category we will present languages that somehow respond to the Music-N paradigm as presented by Max Mathews [Mathews, 1969]. The second one includes languages with a narrower scope as they only provide constructs for specifying music scores.

§2.6.1 Music-N Languages

The Music-N paradigm was introduced by Max Mathews as a result of his continuous iterations over his Music program. Out of all of them it was probably Music-V the one that had most impact on the community and most influenced the future evolution of computer music [Mathews, 1969]. Music-V was written in Fortran and could therefore run on any computer while previous attempts had been written directly in assembler and could only run on specific hardware. Music-V and its derivatives, thereafter known as Music-N languages, became popular in the 70's but are still in use today.

While designing Music-V, Max Mathews addressed two fundamental issues: first, the great amount of data needed to specify a sound function; and second, the need for a simple, powerful language to specify complex sequences of sound. The way he tried to give solution to these problems was to store functions to speed up computations, to use *unit generator* building blocks to provide flexibility, and to define the concept of *note* for describing sound sequences [Pope, 2004].

The concept of unit generator is a central issue in Music-N languages. A unit generator can be defined as the minimum functional entity in a Music-N system. Traditional unit generators receive input control signals and produce sound at their outputs and include functionalities such as simple oscillators or envelope generators. Unit generators can be combined into composite structures called *instruments* or *patches*.

Another fundamental issue in Music-N languages is that a sound structure is defined or programmed in two different parts: the *instrument* or *orchestra definition* and the *score* or note list. This model implicitly assumes that the composer can express everything as a list of notes and that all sound processing or generation can happen inside an instrument. In the synthesis process the composer uses a group of sound objects (or *instruments*) known as the orchestra. This orchestra is controlled from a score and is defined using a programming language with specific functions. These functions or *modules* can be organized and combined in multiple ways in order to define instruments. These instruments sounds can be controlled from the score parameters or from other parameters in the same instrument. A traditional Music-N orchestra file is very similar to a program source code. The score initializes the system (with information usually contained in the header) and then contains a list of time-stamped

notes that control the different instruments.

In this section we will present some music languages that comply to the Music-N paradigm although most of them extend it in some particular way.

§2.6.1.1 Supercollider

Supercollider [McCartney, 2002, Pope, 2004] is a language for sound and image processing developed by James McCartney. It can be considered as a Music-N style language in its use of unit generators and other concepts such as instrument, orchestra or events. Nevertheless it presents important differences in respect to traditional Music-N languages such as CSound (see 2.6.1.2). The main differences are: (1) most Supercollider programs can run in real-time and process live sound and MIDI inputs/outputs; (2) Supercollider is a comprehensive general-purpose programming language with facilities for file input/output, list processing, and OO programming; and (3) Supercollider is an integrated development environment.

Supercollider has been implemented in Apple Macintosh and Be computers though more ports are planned. It is a high-level programming language with a syntax derived from C++ and Smalltalk. Its development environment includes a program text editor, rapid compiler, run-time system and a GUI builder. Supercollider Instruments can take their inputs from real-time MIDI controllers and can process audio files and live sound input.

Motivations for the design of Supercollider were the ability to realize sound processes that were different every time they played, write pieces describing ranges of possibilities rather than fixed entities and to facilitate live improvisation by a composer/performer.

Supercollider computes control functions and other values at a lower rate than the sampling rate called the "sub-frame" size. Its default value is 64 though it can be set to any value between 4 and 256. In Supercollider, data is played as it is generated.

SC's syntax is an OO programming language, with a syntax mixture of C++ and Smalltalk. In SC one can program in two styles: function-oriented or message-passing. As in most programming languages, there are different kinds of statements: comments, declarations, assignments and control structures. The language includes everything you would expect to find in a general programming language but also includes specific functions for music and signal processing.

In a typical SC program there are several parts: (1) Header: title, comment, date, version...; (2) Declarations: declare output buffers, sound files, function tables...(required); (3) Init function: run at compile time if present (optional); (4) Start function: called at run-time if present, runs the instruments (normally present though optional); (5) Instrument functions: can be called from the Start function.

Unit generators are regular OO objects with constructors, and evaluation methods. SC also has support for OO classes and inheritance. The `UGen` class provides the abstraction of a unit generator, and the `Synth` class represents a group of `UGens` operating as a group to generate an output. The unit generator API is a simple C interface.

An *Instrument* is constructed functionally. When writing a sound-processing function one is actually writing a function that creates and connects unit generators. This is different from a static object specification of a network of unit generators. Instruments in Supercollider can "generate" a network of unit generators.

A *composition* can be considered as a sequence of events and this abstraction is accomplished via the concept of a stream. A *stream* is an object to which the next message can be sent to get the next element. A stream ends when it returns nil. By default, all objects in Supercollider respond to next by returning themselves so any object can be used as an infinite stream of itself. An event stream returns dictionaries that map symbols to values, the composition code does not need to know anything about an instrument argument list and may contain any set of parameters.

Supercollider was originally designed to combine a high-level language and a synthesis engine. But later some reasons were found to separate the composition language from a synthesis engine [McCartney, 2002]. The main reason is that some synthesis processing time must be consumed generating events, if the composition language is separated it can run in the background generating events.

That is why in Supercollider Server the synthesis engine and the language were separated and are now two applications that communicate via a slightly modified version of Open Sound Control (OSC) [Wright, 1998a]. This allows to run several instances of the synthesis engine either in different processors or machines. Controlling the synthesis engine is as simple as opening a socket and sending commands, so any program (Max, a C++ program...) could control it.

In Supercollider 3 Synth Server, while synthesis is playing new modules can be created, destroyed or repatched and sample buffers can be created and reallocated. All commands are received via TCP or UDP using the simplified version of OSC. If MIDI is desired, it is up to the client to convert it to OSC commands for the synthesis engine.

There are two versions of the Supercollider Server synthesis engine. One uses a block computation model and unit generator plug-ins. Instruments are loaded as files that describe patches of these unit generators. This version has a control rate and audio rate. The other version implements single-sample computation with the instruments loaded as compiled plug-ins. The synthesis class library can generate C++ code to be loaded by the synthesis engine. Instead of a single control rate any unit generator may run at any power of two division of the audio clock rate. Only source unit generators

need to specify the computation rate.

All running modules are ordered in a tree of nodes that define an order of execution. There are two types of nodes: Synths and Groups. A Synth is just a collection of unit generators that can be addressed together. A Group is a collection of Nodes.

Synths send audio and control signals to each other via a a pair of global arrays of audio and control buses. Using buses allows to connect Synths without a priori knowledge about them. The lowest-numbered audio busses get written to the audio hardware outputs, then there are the audio input buses.

The Supercollider user interface has a program text editor, a message output view and an instrument user interface view. But a GUI can also be created with Supercollider.

§2.6.1.2 CSound

CSound [Vercoe, 1992] is a Music-N language that was first designed to make this technology portable and available in any platform with the only requirement of having a C language compiler. CSound has suffered several revisions during the years and now includes contributions from many different collaborators, most of them computer music composers. But, above all, CSound is the project of Barry Vercoe [www-BarryVercoe,].

CSound has gained a great acceptance in the academic community but has not become a mainstream tool, possibly because of the relative complexity of the language and the requirement of some previous programming experience.

In CSound an orchestra definition file is a regular text file written according to a specified protocol. This orchestra file is made up of a header and a list of instruments. In the header we can find global parameters such as the output sampling rate (sr) or the control frequency (kr).

The instrument definition starts with an instrument label and finishes with an end declaration. In between those two statements, different declarations are included. These declarations define the modules or functions that will be used and the way that they will interact.

The CSound score file is also a standard text file that follows a particular syntax and protocol. As a matter of fact it is just an ordered list of labels and numbers separated by a whitespace.

The score file can begin by defining a set of function tables that will be used to generate the synthesis waveforms. From the score body itself there is the possibility of sending messages to these functions or to an instrument in the orchestra. If the message is to be sent to a function it will start with "f(x)" and if it is intended to affect an instrument with "i(x)", where in both cases "(x)" is the number of the function or the instrument in the orchestra where the message is addressed. The next elements

in a message are the temporal definition of the event start and its duration. The following values are addressed to the different parameters and their meaning is determined by the particular instrument definition.

In the following example we can see the two different kind of instruments available in a score. The first one is addressed to a table that generates sinusoidal waveforms and the second one is addressed to an orchestra instrument, asking it to produce a 5 seconds note starting at time 0.

```
f1 0 4096 10 1
i1 0 5 ... ... ... ... ... ... ...
```

§2.6.1.3 SAOL and SASL

SAOL and SASL are languages included in MPEG-4's Structured Audio component. In section 5 we will comment on the whole conceptual model behind Structured Audio. But we will now concentrate on its Music-N facet represented by the SAOL language for defining orchestras and the SASL language for musical scores.

The Structured Audio Orchestra Language (SAOL) [Scheirer, 1999b, Scheirer, 1998a] is a general purpose synthesis language derived from CSound. Compared to CSound, SAOL presents improved syntax, a smaller set of core functions, and a number of additional syntactic features. SAOL is optimized for digital audio synthesis and digital audio effects but any digital signal processing that can be expressed as a signal flow can be expressed in SAOL [Scheirer, 1998b].

SAOL is a two rate synthesis language: every variable represents either an audio signal that varies with the sampling rate or a control signal that varies at the control rate. The sampling rate limits the audio frequencies and the control rate limits the speed with which the parameters can vary.

Stored function tables are called wavetables. SAOL has about 100 primitive processing instructions, signal generators, and operators. A typical decoding process may also include a step that resembles linking a high-level language with a fixed library of abstract functions. These primitives hold most complexity for SAOL and can be optimized for specific implementations.

In practice, two different implementations of SAOL decoders exist at the moment of this writing. The reference software included in the standard uses the interpreter approach resulting in a very inefficient application. The other implementation is called sfront [Lazzaro and Wawrzynek, 2001] and consists on a program that translates SAOL into a C program that is then compiled and executed.

SAOL defines a two-level syntax: at the bit level to describe the messages that will be streamed into a network and at a higher level to provide an understandable representation of the language. We will briefly mention this latter, which is more directly related to our interests.

The language is a BNF (Backus-Naur Form) grammar. It contains punctuation signs that

are used to give messages a particular syntax; identifiers that define orchestra symbols; numbers that describe constant values; comments that add internal documentation; and whitespaces that lexically separate the different text elements.

The orchestra is the set of signal processing routines and declarations that conform a procedure description in Structured Audio. It is made up of four different elements:

The *Global Block* contains the definition of those parameters that are global to the orchestra. It must be unique for an orchestra and it can hold five different kinds of messages: *global parameters* such as sampling rate, control rate, or number of audio inputs and outputs; definition of *global variables* that can be used from different instruments; *path* definitions describing how the instrument outputs will be addressed to the buses; and *sequence* definitions in order to control instruments on real-time.

After the Global Block we find the instrument definitions where the necessary sequences in order to process SASL or MIDI instructions are defined. An instrument declaration is made up of the following parts (in the given order): an *identifier* that defines the instrument name; a *list of identifiers* that define the names of the parameters involved in that particular instrument (pfield); an optional value to specify the *MIDI preset*; an optional value specifying the *MIDI channel*; a list of *variable declarations*; and a set of messages that define the instrument functionality.

An *Opcode* is simply a function that can have several inputs or variables and a single output or result. Opcodes can be used from any instrument in the orchestra. SAOL offers a set of ready-to-use Opcodes called *Core Opcodes* that include things such as mathematical functions or noise generators. The user can use the implemented opcodes or define new ones.

A function declaration has different elements in the following order: a number that defines the *velocity* with which it is executed; an *identifier* that defines the name of the function; a list of the *formal parameters* in the function; a list of *variable declarations*; and a set of *messages* that define the functionality.

Template Instruments describe multiple instruments that are made slightly different using a limited syntax of parameters.

Elements in the orchestra can appear in any order. For instance, a function definition can appear before or after being used.

The other language in Structured Audio is the Structured Audio Score Language (SASL), an event description language that will be used to generate sounds in the orchestra. SASL syntax has been kept very simple and includes very few high-level control structures, this is left for the implementer of the specific tool (sequencer, editor...).

Just as SAOL, SASL describes a two-level control language although we will just mention the

user level, based on a list of text messages.

Any event in a SASL score has a temporal statement that defines at what moment it takes place. This time statement can only be specified in musical notation and therefore the absolute time depends on the value of the tempo global variable.

A SASL score has different kinds of lines: *Instrument Lines*, *Control Lines*, *Tempo Lines*, *Table Lines*, and *End Lines*.

An Instrument Line (InstrLine) specifies an instrument initialization at a particular moment. It has the following elements: the first identifier is the label that will be used to refer to the instrument; the first number is the initial time of the instrument; the second identifier is the instrument name that is used to select one of the instruments described in the SAOL file; the second number is the temporal duration of the instrument initialization (if it is -1, the initialization has no temporal limit); and finally it has a list of parameters (pfields) that will be passed to the instrument for its creation.

A Control Line specifies an instruction that is sent to the orchestra or a set of instruments. It is made of the following elements: the first number specifies the initial time of the event; the first identifier (optional) specifies what instruments will receive the event; the second identifier is the name of the variable that will receive the event; finally the second number is the new value for the control variable.

A Tempo Line specifies the new value for this global variable for the decoding process. It has two elements: the first number is the time when the tempo change will be applied; and the second number is the new tempo value specified in ppm.

A Table Line specifies the creation or destruction of a wavetable. It contains the following elements: the first identifier is the name of the table; the second identifier is the name of the table generator or the "destroy" instruction; the list of pfields id the list of parameter for the wavetable; and the sample refers to the sound from which the wavetable is extracted.

And finally a Final Line specifies the end of the sound generating process.

Structured Audio also offers a simpler format for music synthesis. A format for representing banks of wavetables, the Structured Audio Sample Bank Format (SASBF) was created in collaboration with the MIDI Manufacturer's Association. Wavetables can be downloaded into the synthesizer and controlled with MIDI sequences.

§2.6.1.4 Common Lisp Music

Common Lisp Music (CLM) [Schottstaedt, 2000] is a musical synthesis and signal processing language of the Music-N family that is written in Common Lisp.

The language includes different components: the generators, the instruments, the lists of notes, tools for sound file input/output, and different graphical interface options based on the sound editor Snd (see [Schottstaedt, 2004]).

CLM's instrument description language can access sound processing functions such as oscillators or envelopes. Instruments can be used in the Common Lisp environment or can be compiled as C language code. CLM instruments are Lisp functions, therefore a list of notes is just an expression that calls these functions. Notes do not even need to be sorted because the process is performed note by note. In the same way, it is possible to process different instruments in parallel.

CLM works on the following platforms: NeXT, Macintosh, SGI, Sun and PC's with GNU/Linux or NeXTSTEP. CLM is a sound compiler written in Common Lisp. In the Lisp environment, while you write a command, the compiler tries to interpret is and perform and action that returns a result. CLM can be used from the Lisp environment or from Common Music but, in any case, you need to learn the basics of Lisp programming.

§2.6.1.5 Nyquist

Nyquist [Dannenberg, 1993] is a music representation language that is as a matter of fact not a Music-N language. Nevertheless, as it can be understood as an evolution of the basic Music-N paradigm, we have decided to include it in this category. Nyquist is distributed as Open Source software.

Nyquist can be seen as the natural evolution from Artic, Canon and Fugue. All of them have been designed by Roger B. Dannenberg and use functional programming for describing temporal behavior. Using some general mechanisms composers can create temporal structures such as notes, chords, phrases, trills, and synthesis elements such as granular synthesis, envelopes and vibrato functions. Previous implementations had great limitations that made them unpractical, for instance Canon did not handle sampled audio and Fugue needed vast amounts of memory and was hard to extend.

Nyquist does not need to preprocess an orchestra or patch. Lisp-based Nyquist programs can construct new patches on the fly and users can execute synthesis commands interactively. Nyquist can work both in real-time and non-real-time modes.

Nyquist uses a declarative and functional style in which expressions are evaluated to create and modify sounds. For instance, for summing two sinusoids the following expression should be introduced: `(s-add (osc) (osc))`.

In Fugue space was allocated for the entire result when adding two signals, this was only practical for small sounds. In Nyquist the addition and synthesis is performed incrementally so that at any time there are only a few blocks of samples in memory. This approach is similar to Music-N

languages such as CSound and, in fact, Music-N *unit generators* are very similar to Nyquist functions. The main difference is that in Music-N the order of execution is explicit while in Nyquist it is deduced from data dependencies. Also Nyquist sounds are more flexible and can be considered as values that can be assigned to variables or passed as parameters.

A Nyquist expression can be represented as a graph of "suspended computation" that is, a structure that represents a computation waiting to happen. When samples are needed, the *suspension* recognizes it needs samples from the different nodes and asks for a block of samples (note that this works by pull or lazy evaluation). Since the order is determined at evaluation the computation graph may change dynamically. In particular, when a new note is played the graph is expanded.

In Nyquist samples are stored in a linked list of sample blocks that is accessed sequentially by following list pointers. Each reader of a sound uses a sound object header to remember the current position in the list and other state information. Incremental lazy evaluation is still used, placing the suspension at the end of the list. When a reader needs to read beyond the last block of the list the suspension is asked to compute a new block which is inserted between the end of the list and the suspension. All readers in the list see and share the same samples regardless of when they are produced or which reader reads first. To prevent lists from exhausting storage resources, reference counting is used to move blocks from the head of the list to a free list from which they can be allocated for reuse.

Nyquist can add sounds with different start times so signal addition must be efficient in the frequent case where one signal is zero. When one operand is zero, the sound block from the other operand can be simply linked into the result with no multiplication or zero filling. Most signal generators produce a signal only over some interval and Nyquist semantics say that the sound is zero outside the interval. A signal that goes to zero is represented by a list node that points to itself, creating a virtually infinite list of zeros. When a suspension detects that its future output will be zero, it links its sound list to this terminal node and then deletes itself.

Nyquist sounds also have *logical stop times* (LST). A seq operator allows to add sound together setting the start time of one sound to the LST of the other. The LST may be earlier or later than the terminating time (e.g. in a note it may correspond to the release time, leaving out the decay).

Nyquist allows various transformations such as shifting in time, scaling, and stretching. The sound headers contain transformation information: to scale a sound, the header is copied and the copy scale-factor field is modified. A drawback is that all operators must apply the transformation to raw samples. Different strategies are used in order to minimize the cost.

Sample rate is specified in the header of each sound and Nyquist allows arbitrarily mixed sample rates. It is the responsibility of the suspension to interpolate samples linearly when rate conversion is

necessary.

Multichannel signals are represented by Lisp arrays and Nyquist operators are generalized in the expected ways.

§2.6.2 Score Languages

In the previous section we reviewed a number of languages that complied with the Music-N paradigm. All of them offered ways to at least define instruments and describe some sort of music score. In this section we will present three languages that just concentrate on the second functionality: describing a score. Although their scope may seem a bit narrow, their approach and ideas are important enough and have influenced some of the decisions later taken in our work.

§2.6.2.1 ZIPI

ZIPI [Mc Millen, 1994] was a comprehensive musical protocol similar to MIDI [MMA, 1998] in its scope but much more flexible and structured. Some of its concepts are also very relevant for the MetriX language that will be presented in chapter 6. This latter is the main reason for including ZIPI in this review.

The first ideas of what would later become ZIPI started in 1989 as a collaboration between Zeta Music and CNMAT (Center for New Music and Audio Technology at the University of California, Berkeley). It all started when trying to break the chain keyboard-MIDI-sampler that was already foreseen at that time. The initial idea was to find a protocol that allowed to naturally control synthesizers from a guitar.

In 1993 and 1994 it was presented at the NAMM event and some enhancements were added responding to suggestions by some companies. But soon after and when the protocol seemed to be about to see the light, Zeta Music disappeared and ZIPI with it.

MDPL or Music Parameter Description Language is a musical language embedded in the ZIPI protocol. In it, a package is made up of three different kinds of information: a header or *Network Overhead*, an *Note Address* and a *Note Descriptor*.

Each MDPL package reserves some bits at the beginning and some at the end. These bits are called Network Overhead and contain low-level information related to the network where the package is being transmitted. It includes the address of the device in the network that is supposed to receive the package.

MDPL defines a three level hierarchy: families, instruments and notes. Each element in each category has a physical address that identifies it. This identifier is known as the Note Address. An MDPL

message can therefore be addressed to any of the 63 available families, any of the 127 instruments in each family or to any of the 127 notes in each of the instruments. (There is also a special address to send a package to all 63 families at once).

Finally, an MDPL package can contain as many Note Descriptors as necessary. A Note Descriptor is made up of an identifying byte and an undetermined number of data bytes. The identifier defines the parameter that is to be modified and the data bytes contain the parameter value.

In MDPL there is a clear distinction between parameters directly related to the musicians actions and those other low-level parameters that are only related to features in the sound. We will finish our brief review of ZIPI by commenting the most interesting and innovative parameters included in MDPL.

- Articulation: There are three kinds of articulation (Trigger, Reconfirm i Release) that allow to play legato notes.
- Pitch: Contrary to what happens in MIDI, a note pitch is not tied to any other message and is represented in 16 bits, the first 7 being the MIDI note and the others semitone fractions.
- Loudness: It is specified in musical notation (pppp~ffff).
- Amplitude: It describes the final gain applied to the output sound.
- Even/Odd Harmonic Balance: Useful to describe some particular timbre features in acoustic sounds.
- Pitched/Unpitched Balance: Amplitude relation between harmonic and inharmonic components of a sound.
- Space Position: The MDPL offers different parameters to control a note position in the space, either in rectangular or polar coordinates.
- Timbre Control: There are different parameters to control the timbre-related features: Brightness, Roughness, Attack character. Furthermore, a timbre space of up to three dimensions can be used to interpolate the different timbres.
- Higher-level messages: The MDPL provides different higher-level messages that provide means of applying modulations or functions.
- Synthesizer requests: ZIPI establishes a bidirectional communication with the synthesizer. With this set of messages, the synthesizer can be queried about its internal features or state.

- Comments: Any kind of comments can be added using this special message that does not affect the synthesizer in any way.

§2.6.2.2 SKINI

SKINI [Cook, 2004] is a score language integrated with the already commented STK environment and developed by the same author, Perry Cook (see section 2.3.2.1).

SKINI is a textual language that is presented both as compatible and as an extension of MIDI. Its main advantages in respect to MIDI are first that text-based messages are easy to read and understand, and second that it uses floating point numbers to enhance resolution.

A SKINI message is therefore a line of text where there are three compulsory fields: the kind of message (ASCII string), the time (in absolute or relative notation) and the channel number (integer that is used to identify the MIDI channel, the synthesizer identifier or even its serial number).

Different fields in a SKINI messages are delimited by spaces, commas or tabs. SKINI operates one line at the time. Any message can be of one of the following kinds: a regular MIDI message such as NoteOn or NoteOff; an extended MIDI message (messages that look different from regular MIDI messages but at the end have to be converted to one) such as LipTension or StringDamping; and No-MIDI messages such as SetPath, OpenReadFile (to, for instance, insert effect files), or Trilling and HammerOn (messages that are directly translated into musical gestures and can be controlled by a MIDI pedal).

SKINI is implemented in C and there is an interface that allows for its extension.

§2.6.2.3 NEXT MusicKit ScoreFile Language

ScoreFile [NeXT, 1990] is a musical score language included in the already commented Music Kit from NeXT Computer Inc. (see section 2.4.1). A NeXT ScoreFile is a regular ASCII text file that can be edited with any text editor.

A ScoreFile is divided in two sections: the *Header* and the *Body*. These two parts are divided by the reserved word BEGIN, and the file ends when the END word or and EOF (regular end-of-file) is found.

The Header is made up of different kinds of statements: *Score Information* Statements, *Part Statements*, *Part Information Declarations*, and *Range Statements*. Score Information Statements are used to assign values to global score parameters such as tempo or sampling rate. Part Statements declare the names of the parts that will be used throughout the score. Part Information Statements are made up of the previously declared name and a collection of parameters. And finally, Range Statements allow

to define note ranges.

The Body consists on a sequence of time-ordered statements. These statements are usually *Temporal Statements* or *Note Statements*. Temporal Statements specify a particular moment in which all the following Note Statements will take place until the next Temporal Statement is received. They use the following syntax:

t[+]expression

Where the reserved word "t" means the current time in musical notation (0.0 at the beginning of the score). If the "+" sign is included, current time is increased by the expression value, else it will directly be assigned the value.

Every time a Note Statement is read, a "note object" is created. A Note Statement is made up of the name of the part where it belongs, a type and name declaration, and a list of parameters.

Finally, there are some Statements that can be included both in the Header or the Body. These are: *Variable Declaration and Assignments*, *Envelope Statements*, *WaveTable Statements*, *Object Statements*, *Include Statements*, *Printing Statements*, *Tuning Statements*, and *Comment Statements*. Variable declaration and assignments follow syntax conventions of C language. The available variable types are: double, int, string, env (envelopes), wave, object, var (generic variable that can be of any type). Envelope Statements allow to declare envelopes from a list of points and a smoothing value. In the same way, WaveTable Statements are used to declare a wave table from a list of partials with their frequency, amplitude and (optionally) phase. Object Statements allow to add any kind of user-defined object. Include Statements are useful to include other scores into the current one while Print Statements include control statements to use when printing out the document. Tuning Statements are used for defining scales other than the temperate twelve tone scale that is used by default.

The ScoreFile language has a number of operators for those statements that need to use them in an expression. These operators include the arithmetic ones plus a special list that adds dB computation, pitch transposition or envelope computation.

§2.7 Summary and Conclusions

In this chapter we have presented a thorough overview of audio and music processing environments. Although all of them have different scopes and motivations, we have presented a classification in different categories. These categories are summarized in the following list:

(1) *General purpose signal processing and multimedia frameworks*: software frameworks for manipulating signals or multimedia components in a generic way. The most important examples in this

category are Ptolemy and ET++.

(2) *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 is the most important framework analyzed in this subcategory.

   (b) Synthesis Oriented: Audio processing frameworks that focus on generating output audio from input control signals or scores. Here it is important to mention STK.

   (c) General Purpose: General purpose Audio processing frameworks offer tools both for analysis and synthesis. Out of the ones presented in this subcategory both SndObj and CSL are in a similar position, having in any case some advantages and disadvantages but no being very mature.

(3) *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.

(4) *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.

(5) *Music languages*: In this category we present different languages that can be used to express musical information. We have excluded those having a graphical metaphor, which are already in the previous category.

   (a) *N-Music languages*: Music-N 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 included in this section, because of its acceptance, is CSound.

   (b) *Score languages*: These languages are simply ways of expressing information in a musical score, usually based on a textual or readable format.

As a conclusion we must observe that many different environments exist and as already commented most of them with different goals, motivations and scope. Many of these environments are the result

of a single person's effort and therefore offer a very personal view on music or audio processing. Few of them can truly qualify as software frameworks as defined in section 1.3 and also very few employ software engineering methodologies or advanced programming techniques.

On the other hand, a few of them (namely Max/Pd and CSound) have been able to build a relatively important community of users that is constantly adding new features to these environments and may be seen as an added value.

The basis that we have set in our analysis of the state of the art for our particular domain will be used for both constructing our proposals and also comparing the final results. In particular, in section 3.3 we will compare our CLAM framework to many of these environments.

CHAPTER 3

The CLAM Framework

CLAM (C++ Library for Audio and Music) is a framework that aims at offering extensible, generic and efficient design and implementation solutions for developing Audio and Music applications as well as for doing more complex research related with the field. CLAM is both the origin and the prove of concept of the Digital Signal Processing Object-Oriented Metamodel that is central to this Thesis and will be presented in the next chapter.

In the current chapter we will give a thorough overview of CLAM's main features and we will introduce some contextual information that will help in understanding why some design decisions have been taken. We will finish the chapter by comparing our framework to some of the similar solutions presented in the previous chapter.

It is important to note that most of the more conceptual and generic models and metamodels that will be presented in the next chapters were obtained during the design and development of the CLAM framework, usually as the generalization of solutions found for particular situations. It is therefore important to understand what the framework offers and what were the design decisions. In that sense we repeat here again our hypothesis that "frameworks generate metamodels".

§3.1 Introduction

At the time CLAM was started the Music Technology Group of the Pompeu Fabra University had about thirty researchers involved in different projects all of them related to the development of algorithms and applications for Signal and Music Processing. Most of these applications were implemented in C++. It was then clear that the amount and quality of the lines of code coming from the different projects was becoming hardly manageable. Although the code had been written using an OO-might-be language like C++ and some classes existed here and there, basic design principles had

not been observed and the result was highly unstructured, difficult to understand and hardly reusable. Flexibility and re-usability had been sacrificed in the sake of what was though to be "more efficient". This situation made it extremely difficult and time-consuming to integrate newcomers or new projects into the group. Bearing those ideas in mind, the CLAM project was started . Since then, an average of five programmers-developers have been working on it. (See more details on those practical issues in Annex A).

For designing the framework we had to fight against an idea that still has high acceptance in the DSP community: unstructured C-like code assures high performance at run-time. One of the main goals of our work has been to prove that a clean and structured design in general, and OO techniques in particular, do not have to imply an overhead in computational efficiency nor a limitation in flexibility for implementing different models belonging to a particular domain.

The framework has been already used successfully for a number of internal projects - some of them with high run-time performance requirements - like high quality time-stretching, sax synthesizer and high-level feature analysis, and it seems to have reached a somewhat mature stage. The project saw its first public release in November 2002, in the course of the AGNULA IST European project. AGNULA [www-Agnula,] (A GNU Linux Audio distribution) focused on offering a complete Linux distribution, both in Debian and RedHat versions, promoting the use of free software in the audio and music domain.

The initial goal of the CLAM project was "To offer a complete, flexible and platform independent Sound Analysis/Synthesis C++ platform to meet current and future needs of all MTG projects" (quoted from CLAM's first Working Draft). Those initial objectives have slightly changed since then, mainly to accommodate to the fact that the library is no longer seen as an internal tool for the MTG but as a library that is licensed under the GPL [Free Software Foundation,] (GNU Public License) terms and conditions.

§3.1.1 Another Audio Library?

What makes CLAM different from other similar solutions that already exist? In section 3.3 we will then make a more extensive study on the differences between CLAM and the environments presented in chapter 2. But it is interesting to highlight from the start what are the most important features of our framework that make it different to anything else.

(1) To begin with, CLAM is truly *object-oriented*. Extensive software engineering techniques have been applied in order to design a framework that is both highly (re)usable and understandable.

Although the term *sound object* has been around for many years, and OO techniques have also been applied in many audio and music related applications, few of these have conceptually applied the "everything is an object" maxima. In our framework, all data types and processing or flow control entities are well-behaved objects.

(2) CLAM is *comprehensive* since it not only includes classes for processing but also for audio and MIDI input/output, XML serialization services, algorithm and data visualization and interaction, and multi-threading handling.

(3) CLAM deals with a wide variety of *extensible data types* that range from low-level signals (such as audio or spectrum) to higher-level semantic structures (such as musical phrase or segment).

(4) The framework is *cross-platform*. All the code is ANSI C++ and it is regularly compiled under Linux, Windows and Mac OSX using the most commonly used compilers. Even the code for input/output, visualization and multi-threading is cross-platform down to the lowest possible layer.

(5) The project is licensed under the *GPL* (GNU Public License) terms and conditions. Although we maintain the option of double licensing the framework (i.e. offering an alternative commercial license), everything offered in the public version is GPL and the project is therefore Free Software, open-source, and collaborative.

(6) CLAM is *bound to survive*. Even though its public success is by no means guaranteed, CLAM will surely remain the basis for all future developments in the MTG and thus will be maintained and updated on a regular basis.

(7) The framework can be used either as a regular C++ *library* or as a *prototyping tool*. In the first mode, the user can extend, adapt or optimize the framework functionality in order to implement a particular application. In the second mode, the user can easily build a prototype application in order to test a new algorithm or signal processing application.

§3.1.2 CLAM as an Object-Oriented Framework

CLAM is both a black-box and white-box framework (see section 1.3.3 for a definition of these terms). But as already mentioned, this is a common property of all software frameworks. It is a black-box framework in the sense that already built-in components can be connected with minimum programmer-effort in order to build new applications. And it is white-box in the sense that abstract classes can be derived in order to extend the framework components with new processes or data containers.

CLAM is a framework for audio and music programming. It offers a general infrastructure that can be used for any signal processing application. But the ready-to-use components and tools focus on the music and audio domain.

The framework can be used to construct general audio or music applications for end-users or artists. But its main focus in on developing prototype applications for research purposes. CLAM can be used as a framework for rapid prototyping, testing new algorithms or ideas and general research on the music and audio field.

The framework is truly object-oriented and its design promotes and to some extent ensures modularity, re-usability, separation of aspects, and conceptual encapsulation. It is important to note that targeted users of the framework are mainly signal processing engineers that may not have acquired good programming and design practices before they are exposed to the framework. A lot of care has been put in enforcing these good practices and ensuring that the framework, when used as white-box, is extended without violating its philosophy. For example, data classes are declared using a mechanism called Dynamic Types (see 3.2.2.2). For declaring a member attribute in a Dynamic Type, a set of special macros must be used. These macros (apart from doing much more interesting things that will later be explained) expand a private attribute and GetX()-SetX() accessors with a user-defined visibility.

Apart from the object-oriented metamodel, or rather as a consequence of, CLAM presents and enforces a particular graphical Model of Computation. The CLAM MoC helps engineers in modeling signal processing systems and as a consequence has immediate benefits on both the analysis of the system itself as its design and implementation. Both the object-oriented model and the MoC derived from CLAM will be introduced in the next chapter.

As already mentioned, CLAM is implemented in C++. This allows for the design of very demanding or time-critical applications. Several real-time applications such as a vocal processor with spectral analysis/synthesis or a sax synthesizer have implemented in the framework. In these cases implementations based on the framework have proven to be even more efficient than the original implementations.

CLAM is also completely cross-platform and portable. A CLAM application developed in GNU/Linux is immediately portable to Microsoft Windows and Mac OSX. Even system-level primitives such as access to sound devices or multi-threading handling are encapsulated in such a way that the target operating system is transparent to the user.

The framework is also completely Free Software (as for the definition given by the Free Software Foundation [Free Software Foundation,]) and licensed under the GPL. Thus it can be used under no restriction as long as the resulting application remains Free. Access is therefore granted to the source

code and the knowledge accumulated during the framework design process is transmitted and reused without restriction.

§3.2 What CLAM has to offer

At the time of this writing, around 300 C++ classes (70.000 loc) exist in our CVS repository.

CLAM brings the world of software design and engineering to DSP developers who could care less about it. For doing so, it offers some general infrastructure like ADT's, XML serialization or a GUI module. But, most importantly, it forces users to follow some good coding principles and it provides a general model for easy (re)usability. Thus, the user of the framework only has to concentrate on writing signal processing algorithms and, eventually, modeling new data structures or implementing particular flow control schemes.

CLAM offers a repository of ready-to-use components that conform the black-box aspect of the framework. These components include containers for data related to signal processing data and algorithms properly encapsulated. But, most importantly, CLAM also includes infrastructure for adding newly developed components that can be easily integrated into the framework. This white-box aspect of the framework does not only include infrastructure for extending the available data and processing classes but also ways of interconnecting components and building applications.

The core of the library is a repository of digital signal processing algorithms related to audio and music. These algorithms can be used for a wide range of applications but, at the time of this writing, they are mostly related with the MTG's research field, which is mainly spectral analysis synthesis and transformations. Nevertheless, the framework has been designed so that further additions can be done without much hassle. This is mainly due to the fact that a CLAM processing network can acknowledge any kind of processing data as long as it complies to the required interface.

Processing classes encapsulate all processing algorithms in a CLAM system. A CLAM system can therefore be seen as a set of interconnected Processing objects. Processing objects offer scalability so that any final CLAM network can be looked at as a Processing composite that includes any number (and levels) of Processing components inside. Processing objects respond to synchronous processing data and asynchronous control events.

CLAM also offers a Visualization module that allows the decoupling of any particular third party toolkit from the system model. This service can be used with any toolkit. The Visualization module implements a model abstraction based on a modified version of the MVC called Model-View-Presentation. The decoupling between the view and the presentation is accomplished through the

implementation of a template functor based callback library. It also implements some particular presentations for basic data types (like audio or spectrum) using FLTK [www-FLTK,] or QT [www-QT,] and OpenGL [Shreiner, 2004].

In this section we will describe in more details the different components in the repository, and the infrastructure, including general tools and sample applications that are offered as part of the framework.

§3.2.1 Repository

As almost any framework (see section 1.3.3) CLAM presents black-box and white-box features. Black-box usage allows direct reuse of already provided components, building a CLAM system means simply connecting existing functionality. This is accomplished through the class *repository* included in CLAM. Just as a CLAM system can be seen as a set of interconnected Processing objects and Processing Data objects the repository can also be divided into these two categories: a Processing repository and a Data repository. In this section we will review what components are included in each of these categories.

§3.2.1.1 Processing repository

CLAM offers a set of signal processing algorithms encapsulated as CLAM Processing classes. These Processing classes are classified in the following general categories:

(1) General Processing

   (a) Analysis

   (b) Transformations

      (i) SMS

   (c) Synthesis

   (d) Generators

   (e) Arithmetic Operators

(2) Input/Output

   (a) Audio File IO

   (b) Audio IO

   (c) MIDI IO

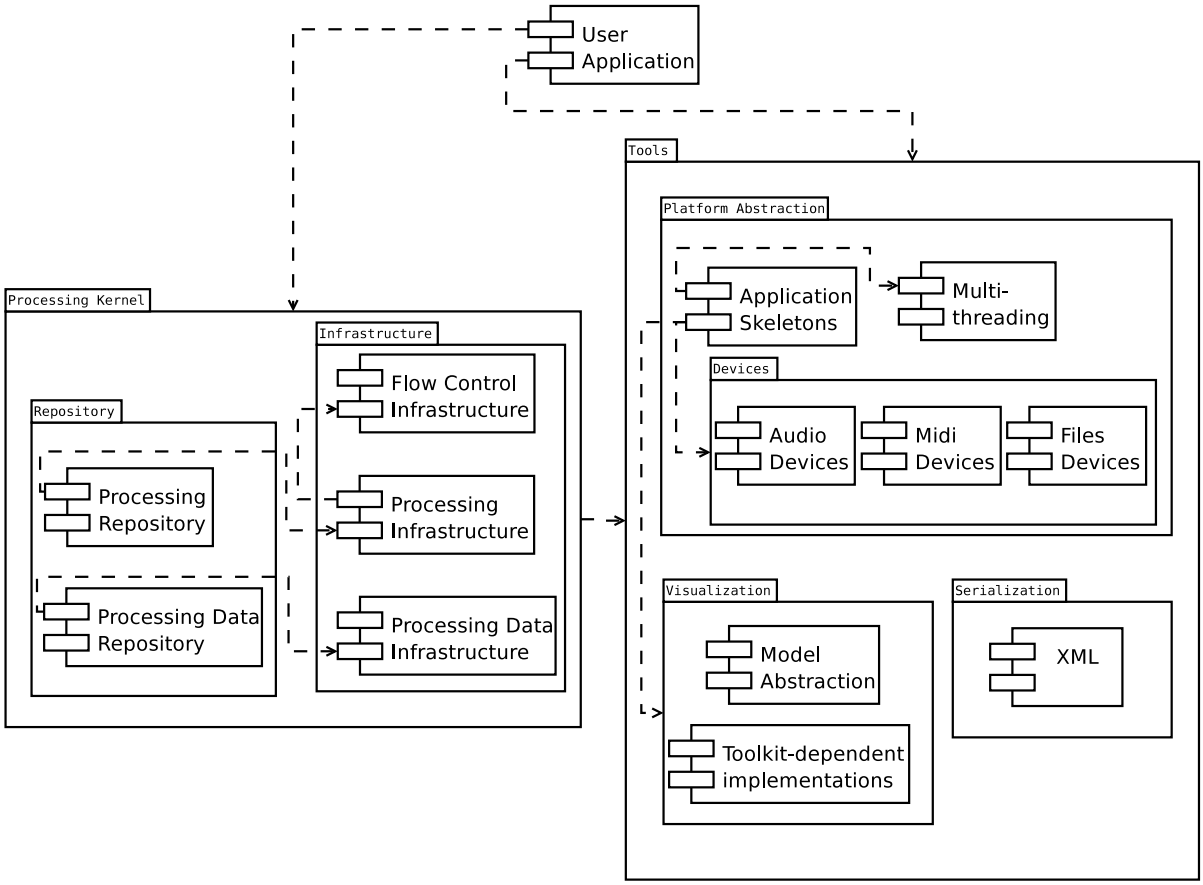

Figure 3.1: CLAM components

(d) SDIF IO

(3) Controls

The processing repository is illustrated in the Component UML diagram in Figure 3.2.

§3.2.1.1.1 General Processing

In this first category different signal processing algorithms can be found. Although these algorithms can be considered general purpose and can be applied to different domains, there is a clear focus on spectral processing of audio and more precisely on the Spectral Modeling Synthesis (SMS) technique (see Annex B).

Analysis

The Processing classes included in this category are used to extract different features from the input signal. All the necessary algorithms related to the SMS analysis process are included. Therefore, we can find Processings for: window generation, signal shifting, performing the FFT, detecting and tracking spectral peaks, detecting the fundamental frequency, extracting the spectral envelop, performing the LPC autocorrelation, and segmenting the audio signal.

The spectral analysis and the SMS analysis Processings represent a special kind of Processing that do not encapsulate a single algorithm but rather use existing simple Processings to build a new one out of their composition. Such Processings are called Processing Composites. In the case of the spectral analysis Processing is composed from the following Processings: window generator, circular shift, FFT and audio multiplier. The SMS analysis Processing adds another layer of composition as it includes the previously commented spectral analysis plus other Processings such as spectral peak detection or fundamental frequency detection.

Transformations

Transformations are divided into general purpose transformations and those that can only be applied to the output of the SMS analysis process. General transformations include some time-domain algorithms such as delay, normalization or envelope modulator. It also includes spectral transformations like frequency domain filtering by applying an ideal function or an spectral envelope.

But using the SMS model more interesting transformations can be applied also in the spectral domain [Amatriain et al., 2002b]. The repository offers SMS-based transformations such as time-stretch, morphing or pitch shifting. These transformations are sub-categorized under the Transformations->SMS category.

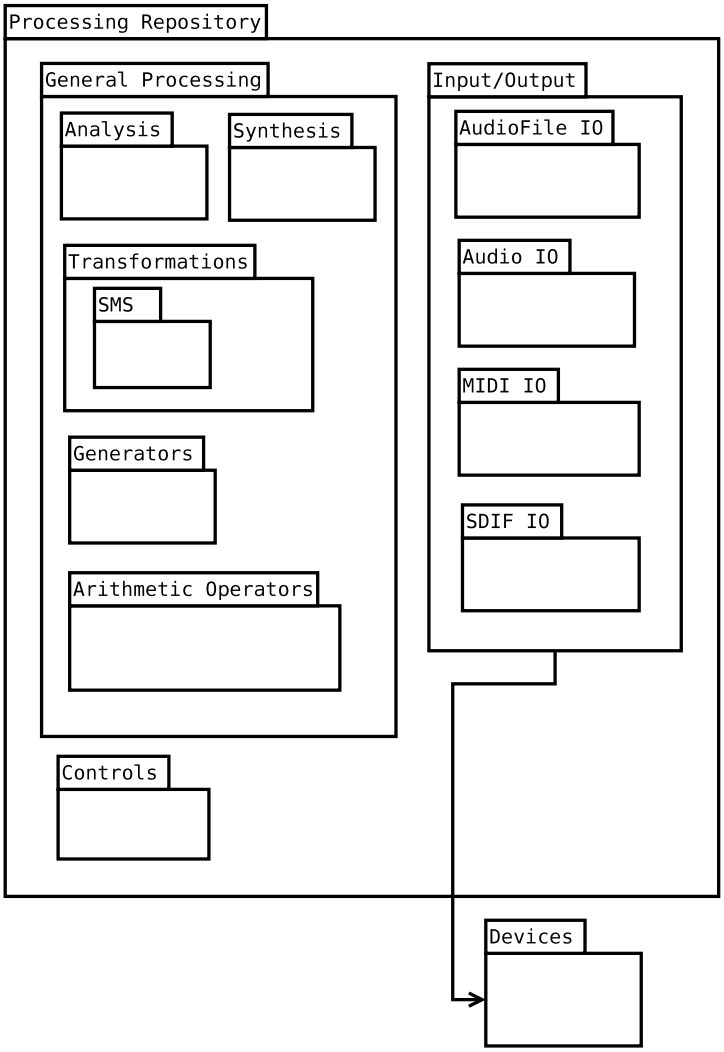

Figure 3.2: CLAM Processing Repository

Synthesis

The synthesis algorithms are all the necessary for the SMS model. Nevertheless, some of them (such as the inverse FFT or the overlap and add) can be applied to different spectral models.

Generators

Generators are special Processings that produce but do not consume data. The CLAM repository of generators includes algorithms like oscillators, wave generators or ADSR (attack-decay-sustain release) envelopes.

§3.2.1.1.2 Input/Output

The second category is related to the input and output of different data into and from the framework. These Processings do not encapsulate algorithms but rather tools or system-level services like access to files or soundcards so they can be transparently used as any other process in the framework. As a matter of fact, the usefulness of these Processings is on the particular service they encapsulate and that will be explained in section 3.2.2.4. The input/output CLAM Processings just add the necessary interface so as to treat these services as a regular CLAM Processing object.

§3.2.1.1.3 Control

Finally, this last category includes Processings that are used for handling control events in some way or another (see section 3.2.2.1)

§3.2.1.2 Data repository

The CLAM data repository contains the necessary data for the Processing repository. These are encapsulated in Processing Data classes and can be classified into two different categories:

- Basic Processing Data
- Descriptors

The first category includes basic data containers like Spectrum and Audio while the second includes higher-level descriptors that are not directly related to the signal such as Melody.

We will now review the most important Processing Data classes included in this repository.

§3.2.1.2.1 Audio

In short, the `Audio` class has three basic attributes (`SampleRate`, `BeginTime`, and `Size`), and one buffer. The `Size` attribute is a structural attribute, therefore a change in its value implies a change in the size of the existing buffer. On the other hand, the attributes `BeginTime` and `SampleRate` are purely informative and thus, a change in their value only implies a change in the attributes but not on the buffer.

The `Audio` class has some additional interface for working with time tags instead of indices or sizes. The Getters/Setters for `EndTime` and `Duration` do not belong to an associated attribute but are rather different ways of changing the size of the `Audio`.

There is also an interface for working with *audio chunks* and *audio slices*. An audio chunk is defined as another `Audio` object that has a copy of a subset of the data in the original audio. An audio slice is also an `Audio` object that has the same values as those found in a portion of the original Audio. But in this case instead of holding a copy, the audio slice references a position in memory of the original one.

§3.2.1.2.2 Spectrum

The `Spectrum` is possibly the most complex PD class in the CLAM repository. It is an important class for the framework's purposes and some extra care and effort have been put into it.

A CLAM `Spectrum` can be represented in one of the following formats: array of complex numbers, array of polar numbers, a pair of magnitude/phase arrays, and a pair of magnitude/phase BPF's (Break Point Function). The `Spectrum` class is designed in such a way so as to always keep consistency of the data in its different representations. This is accomplished through some public methods to change the *type* of the representation and *synchronizing* all existing representation to the data in one of them.

There is an accessory interface for accessing/setting magnitude and phase regardless its internal representation. These methods are not efficient but help in keeping different representations transparent for users or some generic algorithms that do not worry about efficiency.

Because of this complexity the `Spectrum` class is the only PD class that has an associated configuration. This configuration is used for initialization purposes and a local copy is not kept in the object. Whenever the `GetConfig(SpectrumConfig& c)` method is called, the argument passed and used as output of the method is synchronized with the internal structure of the spectrum.

§3.2.1.2.3 SpectralPeak and SpectralPeakArray

A `SpectralPeak` is a very basic storage PD class that has the following attributes: scale, frequency, magnitude, phase, bin position, and bin width. It has also some operators such as product, distance and log/linear scale converting routines. By itself it is seldom used and should be preferably used through the `SpectralPeakArray` class.

The `SpectralPeakArray` is not, as its name may imply, just a simple array of `SpectralPeaks`. As a matter of fact, the `SpectralPeakArray` class does not hold `SpectralPeak` inside but rather a set of buffers containing magnitude, frequency, phase, bin position, bin width and index. This representation has been chosen for efficiency reasons in order to be able to operate on contiguous memory.

Apart from these buffers, it includes other attributes such as `Scale`, `nPeaks` (number of peaks currently available) and `nMaxPeaks` (maximum number of peaks allowed).

Even though the most efficient way to deal with a peak array is working directly on the buffers , two accessory interfaces are offered: first, you can access/modify any of the attributes of a given peak by using the interface offered by methods like `GetMag(index)` or `SetPhase(index)`; but also, you can use an interface using `SpectralPeak` objects through the `GetSpectralPeak(index)` and `SetSpectralPeak(index)` methods. Note that these methods do not return a pre-existing peak but rather construct the peak object on the fly. Therefore, they are far from efficient.

Another particularity that needs mention is the `IndexArray`. It is a multi-purpose array of indices that is used for fundamental detection, peak continuation and track assigning. It is sometimes indeed a very convenient way of dealing with many insertions/deletions of peaks into the array as they can be substituted by a simple change in index. The `SpectralPeakArray` class offers an accessory interface (consisting of several methods) for working through indices.

§3.2.1.2.4 Fundamental

The `Fundamental` class is a basic storage PD class used for storing the result of a fundamental (pitch) detection algorithm: a set of candidate frequencies and the computed estimation error if present. It has two integer attributes that hold the current number of candidates and the maximum allowed and two arrays: one of frequencies and the other one containing the errors.

§3.2.1.2.5 Frame

The `Frame` class has two time related attributes that are instantiated by default: CenterTime and Duration. Apart from that, it has many other attributes that belong to one of the PD classes explained in the previous sections. Namely, we have: two `Spectrum` (one for the general spectrum and the other for the residual component), a `SpectralPeakArray`, a `Fundamental` and an `Audio` attribute that is usually used for storing the windowed audio chunk that has been used for generating the other data.

All other methods are just shortcuts for the getters and setters of the previous attributes and may come in handy for some applications that do not bear efficiency requirements. See structure of the `Frame` class in the `Segment` UML diagram in figure 3.3.

§3.2.1.2.6 Segment

A `Segment` consists basically of an audio frame (`Audio` attribute) and an aggregate of `Frames`. This aggregate is implemented as a list so as to favor fast insertions and deletions and supposing that access is usually going to be sequential. This list of frames can be searched upon, using its begin time as the sorting criteria. Apart from this, the `Segment` follows a composite pattern so a segment can in turn hold an aggregate of other `Segments` (which are known as *children*). In the composite structure, only the root segment may hold data (frames and audio) but this data may be accessed from a child located at any level. For doing so, all children have a pointer to their parent. In order to know if the Segment holds data or not, a structural attribute is included, which may be accessed through the `GetHoldsData`/`SetHoldsData` interface. The `SetHoldsData` method is not just an accessor, if set to true, the child will actually detach itself from its parent and copy the data that corresponds to its time interval. If set to false the child will remove the data attributes (frames and audio).

A `Segment` also has two informative attributes: `BeginTime` and `EndTime`.

The UML class diagram[1] in figure 3.3 illustrates the inner structure of the `Segment` class and its associates:

[1] Please refer to Figure 1.1 for a basic explanation of the UML notation

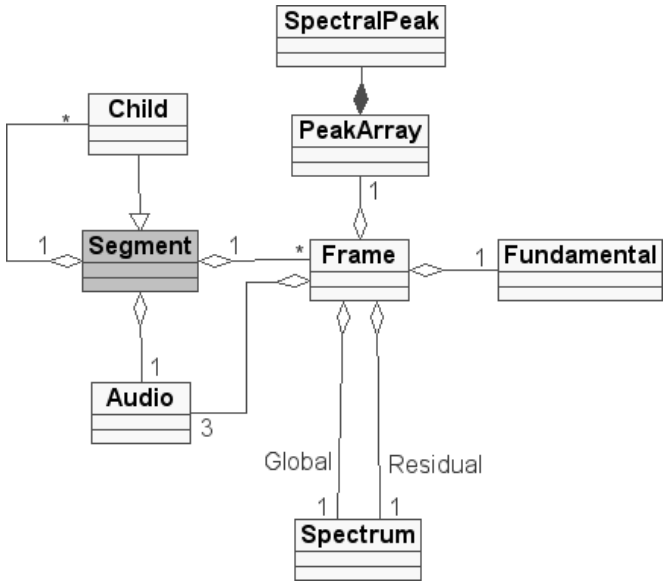

Figure 3.3: CLAM Segment

§3.2.1.2.7 Descriptors

Descriptors are a special kind of ProcessingData that describe numerical attributes computed from the data in a related Processing Data object. This computation is accomplished using 'basic' statistical computations such as mean or nth order moments. For computing these statistics efficiently, CLAM offers a statistical computation module that uses functor objects with memory and template metaprogramming techniques.

Descriptor classes are storage classes where the values of these descriptors are kept. They make extensive use of the XML storage facilities in CLAM as a basic functionality of descriptors is storing them in an appropriate way for later using information retrieval techniques. Furthermore every descriptor class has an associated *extractor* where the actual value computations are implemented as combination of basic statistics.

CLAM offers low-level Descriptors and associated extractors for Audio, Spectrum, Spectral Peaks, Frame and Segment. This latter processing data has special descriptors as it includes temporal statistics such as mean and deviation for other descriptors computed on a frame base.

This is a list of the low-level descriptors currently implemented in the framework:

- **Audio Descriptors**: Mean, Variance, Temporal Centroid, Attack, Decay, Sustain, Release, Log, Attack Time, Energy, Zero Crossing Rate, Rise Time, Decrease.
- **Spectral Descriptors**: Mean, Energy, Centroid, Second to Sixth Order Moments, Irregularity, Tilt, Flatness, Kurtosis, Strong Peak, High Frequency Coefficient, Mel Cepstrum Coefficients, Mel Cepstrum Coefficients Derivative, Band Energy, Maximum Magnitude Frequency, Low Frequency Energy Relation, Spread, Skewness, Rolloff, Slope, Pitch Contour Profile.
- **Spectral Peak Array Descriptors**: Magnitude Mean, Harmonic Centroid, Spectral Tilt, Harmonic Deviation, First to Third Tristimulus, Odd Harmonics, Even Harmonics, Odd to Even Ratio
- **Frame Descriptors**: Spectrum Descriptors, Spectral Peak Descriptors, Residual Spectrum Descriptors, Sinusoidal Spectrum Descriptors, Audio Frame Descriptors, Sinusoidal Audio Frame Descriptors, Residual Audio Frame Descriptors, Synthesized Audio Frame Descriptors, Morphological Frame Descriptors.
- **Segment Descriptors**: Mean Frame Descriptors, Maximum Frame Descriptors, Minimum Frame Descriptors, Variance Frame Descriptors, Fundamental Frequency, Audio Descriptors.

Apart from these low-level descriptors the same infrastructure is used for higher-level descriptors such as melodic or rhythmic description.

§3.2.2 Infrastructure

While the components described in the previous section allow to use the framework in a black-box manner, connecting already existing classes, most of the strength of the CLAM framework lays in the infrastructure that allows to use it in a white-box manner, extending base classes for particular uses. In this section we will outline this infrastructure and its extension capabilities.

§3.2.2.1 Processing infrastructure

All algorithms in CLAM are encapsulated in a class derived from the abstract `Processing` base class. Thus it can be said that every concrete derived class is a "Processing class" and each of its instances is a "Processing object". The main goal of this abstract class is to minimize the developer's effort when introducing a new algorithm while enforcing good programming practices that produces a code that results both efficient and understandable. For this reason, the class provides some services and enforces some particular implementations through its interface.

The Processing base class forces all derived classes to implement a *port* mechanism. Processing objects communicate with other Processing objects through the use of ports. Input ports introduce data into a Processing unit and the result of the process is written into the output port.

All concrete Processing classes also need to declare a related configuration class. This class is derived from the base `ProcessingConfig` class and its name is recommended to be the same as the one of its associated Processing class adding the "Config" suffix. In some cases several processing classes may share the same configuration class. The FFT is a clear example of this. For example, several FFT classes exist for different algorithm implementations in the CLAM repository, but they all derive from a common `FFT_base` class, and they share a common configuration class: `FFTConfig`.

The configuration will be used for initializing the Processing object before its execution. For enforcing the use of this configuration mechanism, the base Processing class declares a `Concrete-Configure()` pure virtual method that must be implemented in any concrete derived class. In this method all initialization operations related to the configuration stage must be implemented. As a matter of fact, the configuration is accomplished by calling the `Configure()` operation in the base class. This operation implements the Template Method design pattern [Gamma et al., 1995]: general operations like internal state modification is modified in the base class while concrete and particular

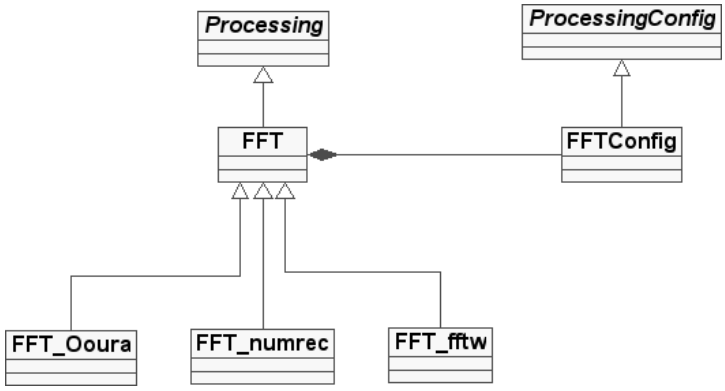

Figure 3.4: FFT's and FFTConfig

configuration issues are delegated to each derived class. It is also very common for Processing classes to keep an internal copy of the configuration object to be able to respond when queried about their current configuration.

Two operations are used in order to get a Processing into execution state and to force it to leave this state: `Start()` and `Stop()`. These operations also implement the Template Method design pattern by implementing the default behavior (i.e. a simple change of state) in the base class and leaving specific issues to the `ConcreteStart()` and `ConcreteStop()` method in derived classes. These methods, though, are not abstract and its implementation is therefore not mandatory.

Processing classes must provide two constructors: a default constructor and a constructor with the configuration class as its argument type. In most cases the former will call the `Configure` operation with a default constructed configuration while the second will call it passing the received configuration object. It is also usual to include some explicit calls to member constructors for initializing members like Controls or Ports that do not have default constructors. Apart from that, most of the initialization functionality will be left to the `ConcreteConfigure()` method.

The main execution methods in a Processing class are the `Do` methods. They are the ones which actually perform the processing action.

There are two different kinds of `Do` methods:

(1) A `Do(void)` method, with no arguments. This is the standard way of using Processing objects connected to a Network (see 3.2.2.3).

(2) `Do(...)` methods taking data objects as arguments. They will have some input data arguments first, and then some output data arguments. A typical processing class will need a single `Do` method of this kind.

Both kinds of `Do()` methods operate in the same way: they read a certain number of data objects from each of the inputs, and write a certain number of data objects to each of the outputs. The difference is that the non-network `Do()` method takes this data objects as arguments (and thus does not use ports), while the network `Do()` method has no arguments, an accesses the Data through the Ports objects. It is very usual that this latter version of the `Do` operation calls the explicit argument version after having extracted the data from the corresponding ports.

§3.2.2.1.1 ProcessingComposites

Sometimes a Processing class needs to become a hierarchical structure that contains other Processing classes. Such container Processing classes are called Processing Composites. For implementing such structure, instead of deriving from the `Processing` base class, the `ProcessingComposite` class must be used.

The only difference from a regular Processing class is that child Processing objects must be configured in the `ConcreteConfigure` operation (usually through the use of an auxiliary `Configure-Children` operation) and they also need to be "attached" to their parent.

§3.2.2.1.2 Controls

Some processing classes need to allow external entities to change the behavior of the objects asynchronously during their execution. Input *controls* are the mechanism to perform this kind of run-time changes. Also, a processing class may be used to detect some kind of event. Output controls are the way to make notifications on asynchronous events. A Processing object output controls can be connected to the input controls of other Processing objects. In CLAM control values are floating point numbers that are sent using an event-driven asynchronous mechanism.

There are two different mechanisms to implement input controls. Regular controls simply store a value, and allow an externally connected output control to change this value.

On the other hand, "extended controls" use a callback-based mechanism. For using such controls call-back method that will be called whenever a new value is sent to the input control must be added to the class. Some reasons for wanting to use such mechanism might be:

- An output control must be sent as a response to an incoming control.

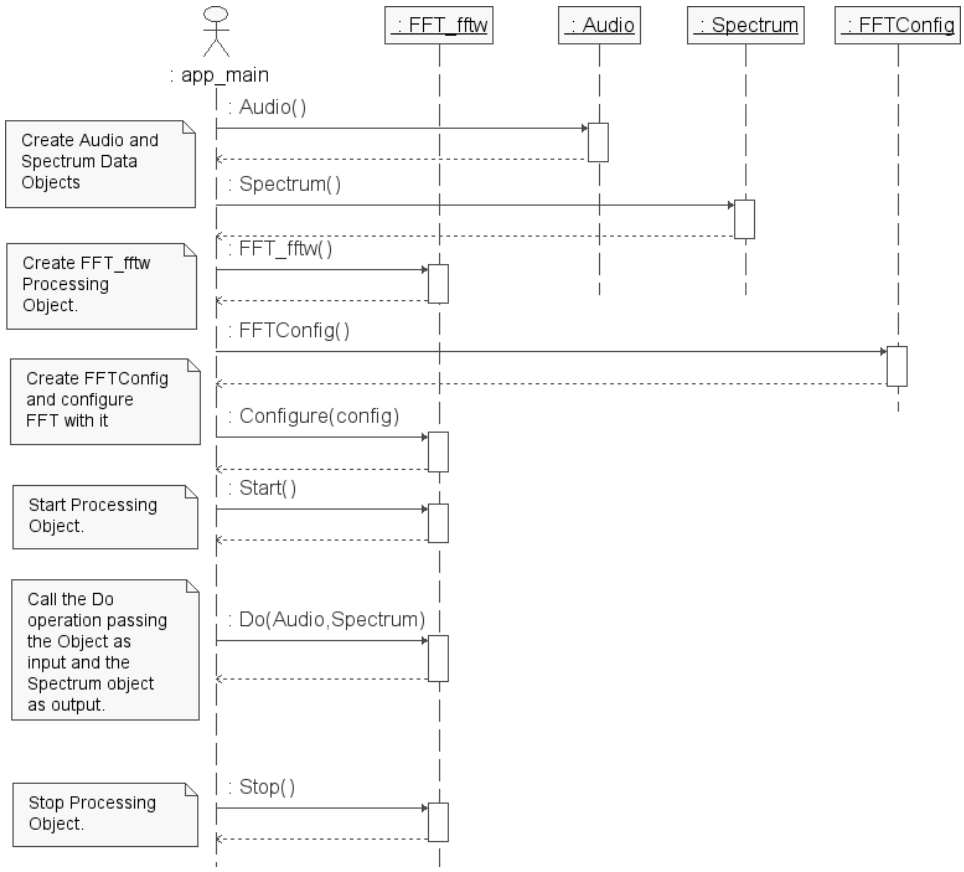

Figure 3.5: Typical CLAM execution sequence

- The processing class needs to have some sort of memory where control events received between two consecutive calls to the `Do()` operation must be stored.

- To avoid having to check all the controls every time a `Do()` operation is called in order to find the ones that have changed.

In order to use extended controls, three steps must also be followed:

Output controls are added in the same way than regular input controls, but taking into account that the name of the control class is now `OutControl`. Output controls are usually sent from time to time in the `Do()` method, using the `SendControl(TControlData val)` method of the `OutControl` class.

Input controls must be initialized in the `ConcreteStart()` method. If the initial value of a control should be chosen by the user, a configuration attribute can be provided in the associated configuration class for this task.

§3.2.2.2 Data Infrastructure

The goals of the data infrastructure are basically the same as the ones of the processing infrastructure: to minimize programmer's effort by providing a set of built-in services and to enforce good coding practices and a homogeneous interface. In the case of data though, two particular goals are also important: to enforce data encapsulation (i.e. no data attribute in a class should be allowed to be public) and to guarantee a homogeneous interface for data access (we want all accessors to have a common interface).

Just as in the processing infrastructure, here we promote white-box behavior by providing an abstract base class that must be derived and made concrete. But in this case the base class makes use of a complex and powerful mechanism: Dynamic Types. Dynamic Types are a key issue in the data infrastructure so we need to understand its purpose and uses in order to understand this part of the framework.

§3.2.2.2.1 Dynamic Types

Though it might be a quite controversial issue and its name may after this explanation seem not the most appropriate[2], there are three main reasons for the decision of implementing and using Dynamic Types (DTs for short) in CLAM.

[2]The "Dynamic Type" name has been kept in the CLAM framework for historical reasons nevertheless, and as it will be seen later, we are not dealing here with classes with a dynamic type behavior but rather with dynamic attributes.

(1) There is a need in some core classes of the library, of working with classes with a large number of attributes, i.e.: the descriptors of audio segments, where in most cases only a small subset is needed. In this sense it could represent a waste of space if memory is always allocated for all attributes. DT can instantiate and de-instantiate attributes at run-time, and do it in such a way that its interface is the same as if they where normal C++ attributes.

(2) We want support for working with hierarchic or tree structures. That means not only composition of DTs but also aggregates of them (lists, vectors, etc. of DTs). With such compositions of DTs, we can use assignation, and two clone member functions: `ShallowCopy()` and `DeepCopy()`, the good thing is that they come free; we don't need to write these members in none of the DT concrete classes.

(3) We need introspection of each DT object. That is the ability to know the name and type of each dynamic attribute, to iterate through theses attributes, and to have some type specific handlers for each. One clear application of introspection is storage support for loading from, and storing to a file, of a tree of DTs. Of course all this is implemented generically, so it appears transparent to the user. XML support, for instance, is implemented. Other profits we take from introspection in DT are debugging aids.

All Processing Data classes in CLAM are DT, as well as the configuration classes for both Processings and Processings Data.

For instantiating and de-instantiating dynamic attributes the developer declaring a Dynamic Type class has to use a set of macros that then, on pre-compile time, expand all of the functionality.

We will describe how Dynamic Type classes work and how they can be used through an example. Imagine we want to model a musical note with a DT[3]. We declare the DT class like this:

```
class Note : public DynamicType
{
public:
  DYNAMIC_TYPE    (Note, 5)

  DYN_ATTRIBUTE   (0, public, float Pitch)
  DYN_ATTRIBUTE   (1, public, unsigned, NFrames)
  DYN_ATTRIBUTE   (2, public, ADSR, Envelope)
  DYN_CONTAINER_ATTRIBUTE (3, public, std::list<Frame>, FrameList, noteFrame)
  DYN_ATTRIBUTE   (4, private, Audio, Wave)
};
```

As it can be seen, three different macros are used in Dynamic Types: `DYNAMIC_TYPE` for expanding the concrete DT constructors, `DYN_ATTRIBUTE` for declaring each dynamic attribute and `DYN_CONTAINER_ATTRIBUTE` for declaring any STL interface compliant container.

[3]This example is not of the Note class available in the CLAM repository. It is a fictitious class created in order to illustrate the different services and behaviors of Dynamic Types.

1. `DYNAMIC_TYPE` is a macro that expands the default constructor of the concrete DT being declared. The first parameter is the total number of dynamic attributes, and the second one the class name. If the writer of a DT derived class sees the need of writing a customized default constructor or other constructors it can be done using special purpose initializers.

2. `DYN_ATTRIBUTE` is used to declare a dynamic attribute. It has four parameters, the first one is the attribute order (needed for technical reasons of the DT implementation), the second one is the accessibility (public, protected or private) the third one is the type: it can be any C++ valid type including typedef definitions but not references or pointers. The forth and last parameter is the attribute name, it is important to begin in upper-case because this name (let's call it XXX) will be used to form the attribute accessors `GetXXX()` and `SetXXX()`, thus the XXX must start in upper-case.

3. `DYN_CONTAINER_ATTR`: The purpose of this one is to give storage (only XML by now) support to attributes declared as containers of objects. For that, we need that container to fulfill the STL container interface, so all the STL collection of containers is usable. This macro has five parameters, one more that `DYN_ATTRIBUTE`: the attribute numeration, accessibility, the type, the name of the attribute and finally the new one: the label of each contained element that will be stored.

Returning to the example above, each `DYN_ATTRIBUTE` macro will expand a set of usable methods:

```
float& GetPitch(), void SetPitch(const float&);
void AddPitch(), void RemovePitch(), bool HasPitch();
void StorePitch(Storage&), bool LoadPitch(Storage&);
```

Of course `GetPitch` and `SetPitch` are the usual accessors to the data. `AddPitch` and `RemovePitch` will instantiate and de-instantiate the attribute, combined with `UpdateData` that will be explained latter on. `HasPitch` returns whether Pitch is instantiate at this moment. Finally `StorePitch` and `LoadPitch` are for storage purposes.

Once, the concrete DT Note has been declared, we can use it like this:

```
Note myNote; // create an instance of the DT Note
Now myNote, have no attribute instantiated. We can activate attributes this way:
myNote.AddPitch(); myNote.AddNSines(); myNote.AddSines();
```

Or in the case that we want all of them, is better to use `AddAll`. This method is not macro generated as `AddPitch`, but is available in any concrete DT.

```
myNote.AddAll();
```

As this kind of operations requires memory management we want to update the data, with its possible reallocations only once for every modification of the DT shape or structure (what can mean lots of individual adds and removes). We'll use the DynamicType member UpdateData for that purpose:

```
std::cout < < myNote.HasPitch() // writes out: 'false'
myNote.UpdateData();
std::cout < < myNote.HasPitch() // writes out: 'true'
```

And now all the instantiated attributes can be used normally using the accessors `GetXXX` and `SetXXX`. For example:

```
myNote.SetNSines(10);
myNote.SetPitch(440);

// lets use some std::list operations:
myNote.GetSines().push_back(440).push_back(440*2);
myNote.GetSines().push_back(440*3).push_back(440*4); // etc.

int i=myNote.GetPitch();   // error! GetPitch() returns float
int j=myNote.GetNSines();  // ok.
```

§3.2.2.2.2 Processing Data

Therefore, one of the main traits of CLAM is the ability to process multiple data types related to the audio and music domain. All these data types are subclasses of the ProcessingData class. In CLAM terminology, a Processing Data class is a class designed for storing all sort of data that will be used in the processing system. All Processing objects are input Processing Data objects (either through Ports when connected in a Network or as arguments of the non-network `Do()` operation). Examples of already provided Processing Data classes include `Spectrum`, `Audio`, `SpectralPeakArray`, `Fundamental`, `Segment`, or `Frame` (see section 3.2.1.2).

Any PD class is in fact a concrete Dynamic Type class therefore and as just explained most of their attributes are macro-derived dynamic attributes (i.e. in the code you will see something like `DYN_ATTRIBUTE(1,public, Spectrum, ResidualSpectrum)`, which means that the given class has a public dynamic attribute called `ResidualSpectrum` that is an object of the `Spectrum` class).

All dynamic attributes have associated automatically derived Setters and Getters that may be used from outside the class. Furthermore, they can be Added and Removed at run-time as explained in the previous paragraphs.

Some classes have private dynamic attributes that cannot be accessed directly but through a given public interface. If you encounter a private or protected attribute with a name starting with the 'pr' prefix (i.e. `prSize`) you should look for its associated public interface (i.e. `GetSize()` and `SetSize()`). Also very rarely, some PD class have an attribute that is not dynamic. In that case, the corresponding Set/Get interface should be offered so its usage is not different than that `of a Dynamic Attribute`.

Most PD classes offer convenient shortcuts for accessing and setting elements in their buffers that should be very useful in a developing stage but should be avoided if seeking efficiency in a given algorithm.

A data storage class derives publicly from ProcessingData. Thus, it is a concrete Dynamic Type class and must use the `DYNAMIC_TYPE_USING_INTERFACE` macro.

Ex:

```
class SpectralPeak: public ProcessingData
{
public:
  DYNAMIC_TYPE_USING_INTERFACE (SpectralPeak, 6, ProcessingData);
  DYN_ATTRIBUTE (0, public, EScale, Scale);
  DYN_ATTRIBUTE (1, public, TData, Freq);
  DYN_ATTRIBUTE (2, public, TData, Mag);
  DYN_ATTRIBUTE (3, public, TData, BinPos);
  DYN_ATTRIBUTE (4, public, TData, Phase);
  DYN_ATTRIBUTE (5, public, int, BinWidth);
 (.)
```

Apart from the default constructor (already available as a result of the Dynamic Type macros), other constructors may be implemented. All these constructors must call the constructor of the `ProcessingData` base class using the member initialization syntax and passing the number of Dynamic Attributes as parameter. Apart from that, these constructors must call a macro-derived method called `MandatoryInit()`, which is in charge of initializing concrete Dynamic Type's internal structure.

Another initializer that is often useful is the `DefaultInit()` method. This method has to be implemented by the developer and is in charge of initializing default attributes and values. This method is automatically called from the Processing Data's default constructor and may also be called from all other constructors.

The most usual non-default constructors that a Processing Data class is bound to have are the Copy constructor and the Configuration constructor. The former is already implemented in the `ProcessingData` base class and this implementation is sufficient as long as all attributes of the concrete class are Dynamic and require no initialization. If not (for example if the class has a non Dynamic member), the developer may make use of the `CopyInit()` method. This method has to be implemented by hand, but is automatically called from the macro derived Copy constructor.

The configuration constructor is sometimes desirable for constructing a Processing Data out of its associated configuration object or out of some sort of initial value (flags, size.). In this case the constructor must explicitly call the `MandatoryInit()` method and then call any other necessary configuration methods.

§3.2.2.2.3 Configurations

Configurations are a special case of Dynamic Types that are used to configure Processing objects and, eventually, Processing Data objects (only complex Processing Data classes such as spectrum use this option). Configurations make use of all mechanisms provided by Dynamic Types and already mentioned. Configurations also include default initializers that add compulsory attributes and set them with default values.

Configurations are particular to each Processing class. As a matter of fact, every Processing class in the CLAM repository must have an associated Configuration class. Furthermore, when extending the framework, this requirement must also be made: every new Processing class introduced must be accompanied by a related Configuration class. Processing configuration classes must derive from the base `ProcessingConfig` class. The name of the configuration class should be the same as the name of the Processing object, adding the "Config" suffix. In some cases several processing classes may share the same configuration class. The FFT is a clear example of this. For example, figure 3.4 shows that several FFT classes exist for different algorithm implementations, but they all derive from a common `FFT_base` class, and they share a common configuration class: `FFTConfig`.

The role of the configuration class is to store all the necessary information to instantiate an object of the related Processing class. In fact, configuration classes may be described as a place-holder for parameters which would otherwise be specified as individual arguments in the processing class constructors.

Usual attributes that are implemented when writing a configuration class are number of inputs or outputs, values for parameters of the processing algorithm which the user may want to specify, and which will be fixed during the execution.

§3.2.2.3 Network infrastructure

A CLAM Network is a dynamic composition of Processing objects. In a Network Processing objects can be added, deleted, connected and reconfigured on run-time. All this functionality is implemented in the Network class and associated classes, which will be explained in the following paragraphs.

Apart from an identifying name, a Network has two important attributes and an interface for interacting with them. These attributes are:

(1) A *Map* of Processing objects. This is a standard stl vector than contains pointers to instantiated Processing objects and uses a unique name for each Processing object as the key.

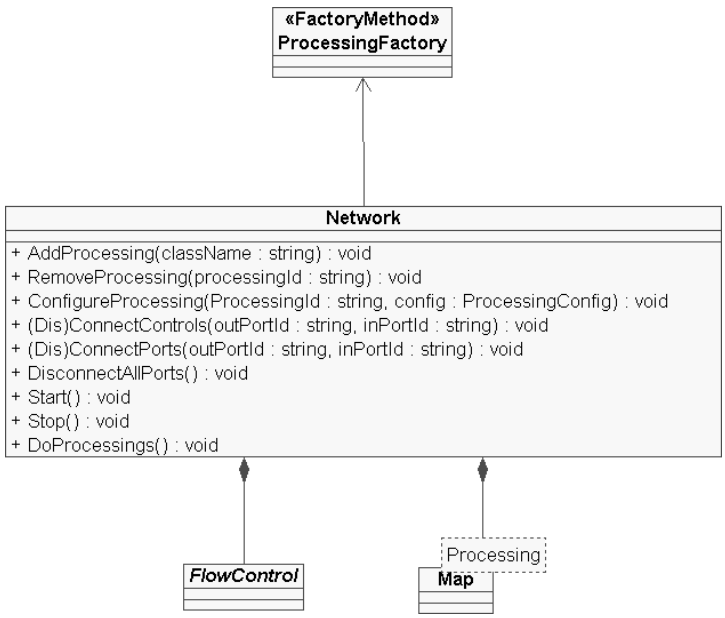

Figure 3.6: CLAM Network class diagram

(2) An associated *Flow Control*. A Network can follow different execution and control policies. This policies are encapsulated in the `FlowControl` class that will be later explained.

A Processing object is added to the Network by indicating its class name. The Network then uses a Processing Factory in order to instantiate a new object belonging to that class and assigning it a unique identifier. The Processing Factory follows the Factory Method design pattern (see [Gamma et al., 1995]) and in particular the idiom described as Object Factory in [Alexandrescu, 2001].

Different operations are also offered to remove, check existence or configure existing Processing objects. An existing Processing object can be accessed by its unique name or using an iterator interface for traversing the whole collection.

All Ports and Controls in the contained Processing are also identified by a name string. Their unique identifier is formed by the concatenation of the Processing unique identifier and the Control or Port Name. Ports and Controls can be accessed by this unique identifier. Furthermore all the InPorts connected to a given `Outport` or all the `InControls` connected to a given `OutControl` can be accessed. The `Network` interface offers methods for connecting and disconnecting Ports and Controls by using these unique identifiers. It also offers a convenience operation for disconnecting all Ports at once.

The Network also offers an interface similar to a regular Processing object. The `Start` and `Stop` operations iterate through all the Processing objects calling their `Start` or `Stop` operations. The `DoProcessings` operation also iterates through all the Processing objects calling their `Do` operation. This iteration, though, is performed in different ways, depending on the configured Flow Control policy.

A Network is a CLAM `Component` and can therefore implement storage facilities through the `StoreOn` and `LoadFrom` operations (see "XML Support" in section 3.2.2.4). A Network is passivated into XML as a collection of Processing objects and their connections. See the following example:

```
<network id="FooNetwork" >
  <processing type="AudioMultiplier" id="multiplier" />
  <processing type="Oscillator" id="oscillator" >
    <Frequency>440</Frequency>
    <Amplitude>1</Amplitude>
    <ModIndex>1</ModIndex>
    <Phase>0</Phase>
    <SamplingRate>44100</SamplingRate>
  </processing>
  <processing type="AutoPanner" id="panner" >
    <Frequency>440</Frequency>
    <SamplingRate>44100</SamplingRate>
    <Phase>0</Phase>
    <FrameSize>512</FrameSize>
  </processing>
  <port_connection>
    <out>oscillator.Audio Output</out>
    <in>multiplier.First Audio Input</in>
  </port_connection>
  <control_connection>
    <out>panner.Left Control</out>
    <in>oscillator.ModIndex</in>
  </control_connection>
  <control_connection>
    <out>panner.Right Control</out>
    <in>oscillator.Amplitude</in>
  </control_connection>
</network>
```

Finally, and as already mentioned, the execution policy in a Network is encapsulated inside the `FlowControl` class. A Flow Control object must be attached to a Network before its execution. This Flow Control object is notified whenever a new Processing object is added or removed from the Network therefore changing its topology. It is also notified when a Processing object is configured, in which case the Flow Control is in charge of reconfiguring its Ports if necessary.

The `FlowControl` class is an abstract class with a single abstract method, the `DoProcessings`. This method has to be implemented in the derived classes that will implement a particular scheduling policy. At this moment two different policies have been implemented: a lazy-evaluation or pull policy and an eager or push execution policy.

§3.2.2.4 Tools

§3.2.2.4.1 XML support

Any CLAM *Component* can be stored to XML as long as a `StoreOn` and `Load` methods are provided for that particular type [Garcia and Amatrian, 2001]. Dynamic Types, and therefore Processing Data and Processing Configs, are Components. But DTs can make use of their introspection capabilities and offer automatic built-in XML persistence. When implementing a new Dynamic Type

class, XML storage facilities are obtained for free, without adding a single line of code. Nevertheless if the user wants a different output from the one automatically derived, the `StoreOn` operation of the concrete component can be overridden.

For passivating a Component an `XMLStorage` object has to be instantiated. Then the `Dump` operation of this object must be called, sending the component that has to be passivated, and the filename as arguments of the operation.

For activating a Component the interface is very similar. An `XMLStorage` must be instantiated and then the `Restore` operation must be called sending the component and the filename as arguments of the operation.

It is interesting to note that other application frameworks such as Ptolemy (see 2.2.1) have also relied on XML for data persistence.

§3.2.2.4.2 Platform abstraction

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

Audio I/O

The core of CLAM audio input/output is the `AudioManager` class. The Audio Manager takes care of all administrative tasks concerning the creation and initialization of audio input and output streams, using the internal, system dependent `AudioDevice` class.

The first thing that needs to be done in order to use audio is create an `AudioManager` object. While this object is present all subsequent audio I/O objects created will use it. Samplerate and latency should also be specified. The latency is used to control the internal buffer size, and depends on the hardware.

The actual audio I/O classes, called `AudioIn` and `AudioOut`, can than be uses to create processing endpoints to retrieve audio from, or write audio to. The `AudioIn` and `AudioOut` objects have to be created with an `AudioIOConfig` object that can be used to specify the device, the channel and the sample rate to use.

The device is referred to with a string that has the following form:

```
"ARCHITECTURE:DEVICE"
```

The available devices depend on the hardware and system configuration. A list of available devices for the platform in use can be obtained from the `AudioDeviceList` class. However, if the device

is not specified, or the string "default:default" is used, the Audio Manager will choose the device that seems most adequate for the current architecture.

In order to have a flexible multi channel system, the channel for each AudioIn and AudioOut can be specified. The Audio Manager will use this information to initialize the internal audio handling.

We can consider the AudioFile I/O tools as a complement to the general Audio I/O ones. Nevertheless, these tools do much more than this. To start with, using some external libraries (see A.2.4) the CLAM AudioFileIn and AudioFileOut classes are able to handle multiple audio formats such as MPEG1-Layer3 (alias mp3), Ogg-Vorbis (a similar yet better and Free compressed format), or any flavor of the PCM formats.

On the other hand, using the AudioFile class, we access other interesting information available in the file other than what is strictly the audio signal. ID3 metadata is handled and can be used for any content-based application.

MIDI I/O

The MIDIIO approach has several similarities with the AudioIO one. The core of CLAM midi input is the MIDIManager class. The MIDI Manager takes care of all administrative tasks concerning the creating and initialization of MIDI input streams, using the internal, system dependent MIDIDevice class. Inorder to use MIDI, a MIDIManager object has to be instantiated. This object will be a Singleton (see [Gamma et al., 1995]), and all subsequent MIDI input objects created will use it.

The actual MIDI input class, called MIDIIn, can be used to parse incoming MIDI data, and handle it in any way. A derived class, MIDIInControl, has one or more OutControls, and can be used to convert the MIDI data to CLAM ControlData. The MIDIIn objects have to be created with a MIDIInConfig object, that can be used to specify the device, and settings for the filter that decides which MIDI data will be parsed to this MIDIIn object. The MIDI Manager and MIDI Devices use this information to create a very efficient MIDI parsing table.

The device is referred to with a string that has the following form:

"ARCHITECTURE:DEVICE"

Currently, the implemented architectures are ALSA and Port MIDI, and the "virtual" MIDI file device. The available devices depend on the hardware and system configuration. A list of available devices for the platform in use can be obtained from the MIDIDeviceList class. However, as in the AudioIO layer, if no device is specified or if "default:default" is selected, the MIDI Manager will choose the device that seems most adequate for the current architecture.

The MIDIInConfig class has three parameters that control which MIDI data will be delivered to

a certain MIDIInput object: ChannelMask, MessageMask, and Filter. ChannelMask and MessageMask are bitmasks, and Filter optionally specifies a filter on the second byte of the MIDI message. The ChannelMask allows to create a MIDIIn that receives MIDI messages on a certain channel or channels only. The MessageMask allows to create a MIDIIn that receives MIDI messages of a certain type only. The Filter allows to create a MIDIIn that receives MIDI messages where the second byte (first data byte) has a certain value. This is particularly useful for control change messages, where the second byte specifies the type of control change.

The derived class MIDIInControl implements MIDIIn with one or more OutControls. The actual amount depends on the filtering used. Outputs will be generated for each message that the MIDIInControl will receive. If. for example, a MIDIInControl is configured for eNoteOn messages, two OutControls will be used, one for key, and one for velocity.

When the MIDI input comes from a device, typically live input, MIDI messages gets delivered through the controls as soon as they come in. In the case of the special "virtual" FileMIDIDevice, this situation is slightly different, and a MIDIClocker should be used to control the sequencing of the data in the MIDI file.

SDIF SUPPORT

SDIF or Sound Description Interchange Format is a binary format defined and supported by various research teams [Schwarz and Wright, 2000, Wright, 1999, Wright, 1998b]. It was created with the goal of having a common format for exchanging synthesis samples, usually spectral domain data coming from a previous analysis.

The mapping of CLAM data to a SDIF File is fairly simple; it is always done from a CLAM::Segment object (see section 3.2.1.2). The Segment internal structure can very easily be mapped to SDIF as it basically holds inside an array of time-ordered frames. Out of the different data inside a frame, only the necessary for the synthesis process is stored into SDIF. That is, residual spectrum, sinusoidal peaks with track number and fundamental frequency. Due to the SDIF specification, all magnitude data needs to be stored in linear scale (as opposed to what is usual in CLAM, where data is in dB).

All this is done using two CLAM Processing: SDIFIn and SDIFOut. SDIFIn takes a Segment in its output port because it needs a single reference where to store the created frames. SDIFOut takes frames in its output port and enables storing frames even from different segments.

§3.2.2.4.3 Visualization

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

The usual way to work around this issue is to decide on a graphical toolkit or framework and add support to it, offering ways of connecting the framework under development to the widgets and other graphical tools included in the graphical framework. In CLAM though we aimed at offering an toolkit-independent support. This is accomplished through the CLAM Visualization Module.

In order to do so a variant of the Model-View-Controller architectural pattern was implemented. In this new version the main actors are the *Presentation*, the *Model Adapter*, and the *Model Controller*.

A Presentation is a graphical metaphor through which some information contained in the model object is shown to the user. A Presentation can be anything from a simple widget to a full application graphical interface, depending on the complexity of the model object to be presented. A Presentation can be activated and deactivated, therefore its existence does not imply its visibility.

The `ModelAdapter` class defines the interface that is common to all model object adapters in CLAM Visualization Module. It offers the interface required by the Observable actor in the GOF Observer pattern [Gamma et al., 1995]. The Adapter concept was chosen in order not to taint the model object interface and to separate effectively the model objects from its representation. The main operation in the `ModelAdapter` class is the abstract `Publish` operation that must be implemented in all subclasses in order to publish the updated model object internal state.

The `ModelController` class is similar to the `ModelAdapter` except in that, besides from publishing the model object state, it also allows to modify it. For that reason it adds the `Update` operation to the previously mentioned `Publish`.

The CLAM Visualization Module also implements a Signal&Slots mechanism similar to that offered by frameworks such as QT [Blanchette and Summerfield, 2004]. The basic rationale behind the Signal&Slot mechanism is the following: Sometimes it is required that an object notifies a change in internal state or the reception of a message to any number of *listeners*. This situation can be modeled in different ways but most of them suffer from a major drawback: coupling. In this sense, the caller must know to some extent the callee interface. Because of this, reuse capabilities are reduced. The Signal&Slot idiom gives solution to this problem. The Signal models the concept of "event notifying", and signals are connected to Slots that represent "event handlers".

In CLAM the Signal&Slot idiom is implemented through three main classes: the `Signal` class,

the `Slot` class and the `Connection` class. The `Signal` and `Slot` classes model the obvious concepts previously explained. On the other hand, the `Connection` class models the knowledge a signal has about who has to be notified whenever a client invokes the `Emit()` operation on it. Each time a `Signal` and `Slot` objects are bound together a `Connection` object is created, tagged by a Global Unique IDentifierIdentifier (GUID). This particular implementation was loosely derived from [Hickey, 1995].

Apart from the previous tools, the non-dependency from graphical toolkit implementation is also accomplished through the use of a Widget Toolkit Wrapper. This Creator/Singleton class produces objects that are abstract wrappers for accessing a GUI Toolkit low-level functionality such as triggering the event loop, triggering the execution of a single iteration of the event loop or setting the refresh rate for graphic displays.

All this general Visualization infrastructure is completed by some already implemented presentations and widgets. These are offered both for the FLTK toolkit [www-FLTK,] and the QT framework [Blanchette and Summerfield, 2004, www-QT,]. An example of such utilities are convenience debugging tools called Plots are also offered. Plots offer ready-to-use independent widgets that include the presentation of the main Processing Data's in the CLAM framework such as audio and spectrum.

§3.2.2.5 Application infrastructure

When implementing a new application, the developer has two options in order to design the architecture. First, the different examples included in the CLAM repository must be visited, it is very probable that one of the included examples is closely related or can be used as a base for developing the new application at hand. But if a fast development of a completely new application is sought, the framework also includes some application skeletons that can be particularized to the new application.

CLAM includes several Application classes that provide a basic framework for typical application situations, such as audio, or audio + graphical user interface. When necessary, threads or setup, and several virtual functions are provided, which can be implemented by deriving from the relevant application subclass (`AudioApplication` or `GUIAudioApplication`).

The `BaseAudioApplication` class is the base class of all `AudioApplication` classes. It sets up a high priority audio thread, and specifies several virtual functions: an `AudioMain()` that will be executed inside the audio thread, this is where the derived classes implement the actual audio processing; a `UserMain()` that will be executed by the main thread, this is where the derived classes implement the actual (graphical) user interface; an `AppCleanup()` that will be executed when the applications ends, this is where the derived classes implement any extra resource cleanup; and a `bool Canceled()` that can be used in the `AudioMain` to check if the audio thread has been canceled.

This class should never be used directly. `AudioApplication` should be used instead when looking for a simple non-graphical application.

The `GUIAudioApplication` class is derived from `BaseAudioApplication`, and additionally provides a standard user-interface, with start/stop functionality. This user interface must use the FLTK library. The virtual function `UserMain()` by default just calls the execution of the FLTK library, but a derived class could add a more complex user interface,. The operation `Run(int argc,char** argv)` has to be called to execute the application.

§3.2.3 Sample applications

CLAM responded to an urgent internal need for having a structured repository of signal processing tools focused on audio and music. For that reason, it has been used as an internal development framework since its very beginning. Of course, our patient users have had to cope with multiple refactoring periods but, on the other hand, we have been able to implement an spiral iteration process, refining requirements and redesigning our model at each turn.

Thus, CLAM applications have been developed and have been used as benchmarks to test the feasibility of the library under very different requirements and to keep some "real-world" input up to date. In the following paragraphs we will review some of them including an off-line application for analysis, synthesis and transformation, a real-time spectral domain sax synthesizer or other applications developed for research projects or for real live concert situations.

§3.2.3.1 SMS Analysis/Synthesis Example

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

The application has three different versions: SMSTools, which has a FLTK graphical user interface; SMSConsole, which is a command-line based version; and SMSBatch, which can be used for batch processing a whole directory. Out of these three it is clearly the graphical version that can find more usages, the other two are only used for very specific problems. The rest of this section will concentrate on the graphical version and only mention some differences with the other versions where strictly necessary.

The main goal of the application is to analyze, transform and synthesize back a given sound.

For doing so, it uses the Sinusoidal plus Residual model (see section B). In order to do so the application has a number of possible different inputs:

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

2. An SDIF or XML **analysis** file. This file will be the result of a previously performed and stored analysis. The XML parser is rather slow and the XML format is rather verbose. For all those reasons the storing/loading of analysis data, although fully working, is not recommended unless you want to have a textual/readable representation of your analysis result, else you will be better off using the SDIF format (see next paragraph).

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

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

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

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

2. An XML **Melody** file.

3. **Output sound**: global sound, sinusoidal component, residual component

Figure 3.7 illustrates the main blocks of the application.

The output of the analysis is a `CLAM::Segment` that contains an ordered list of `CLAM::Frames`. Each of these frames has a number of attributes, but the most important are: a `CLAM::SpectralPeakArray` that models the sinusoidal component (including information about sinusoidal track), a residual spectrum and the result of the fundamental frequency detection algorithm.

The output of this analysis can be (1) stored in XML or SDIF format, (2) transformed and (3) synthesized back.

In order to make the application work, a valid configuration XML file must be loaded although the the default one can also be edited through the graphical interface. This configuration includes all the different parameters for the analysis/synthesis process. The following is an example of a valid XML configuration:

```
<SMSAnalysisSynthesisConfig>
<Name />
<InputSoundFile>c:/1_brief.wav</InputSoundFile>
<OutputSoundFile>c:/1_out.wav</OutputSoundFile>
<OutputAnalysisFile>c:/analysis.sdif</OutputAnalysisFile>
<InputAnalysisFile>c:/analysis.xml</InputAnalysisFile>
<AnalysisWindowSize>513</AnalysisWindowSize>
<AnalysisHopSize>256</AnalysisHopSize>
<AnalysisWindowType>Hamming</AnalysisWindowType>
<ResAnalysisWindowSize>513</ResAnalysisWindowSize>
```

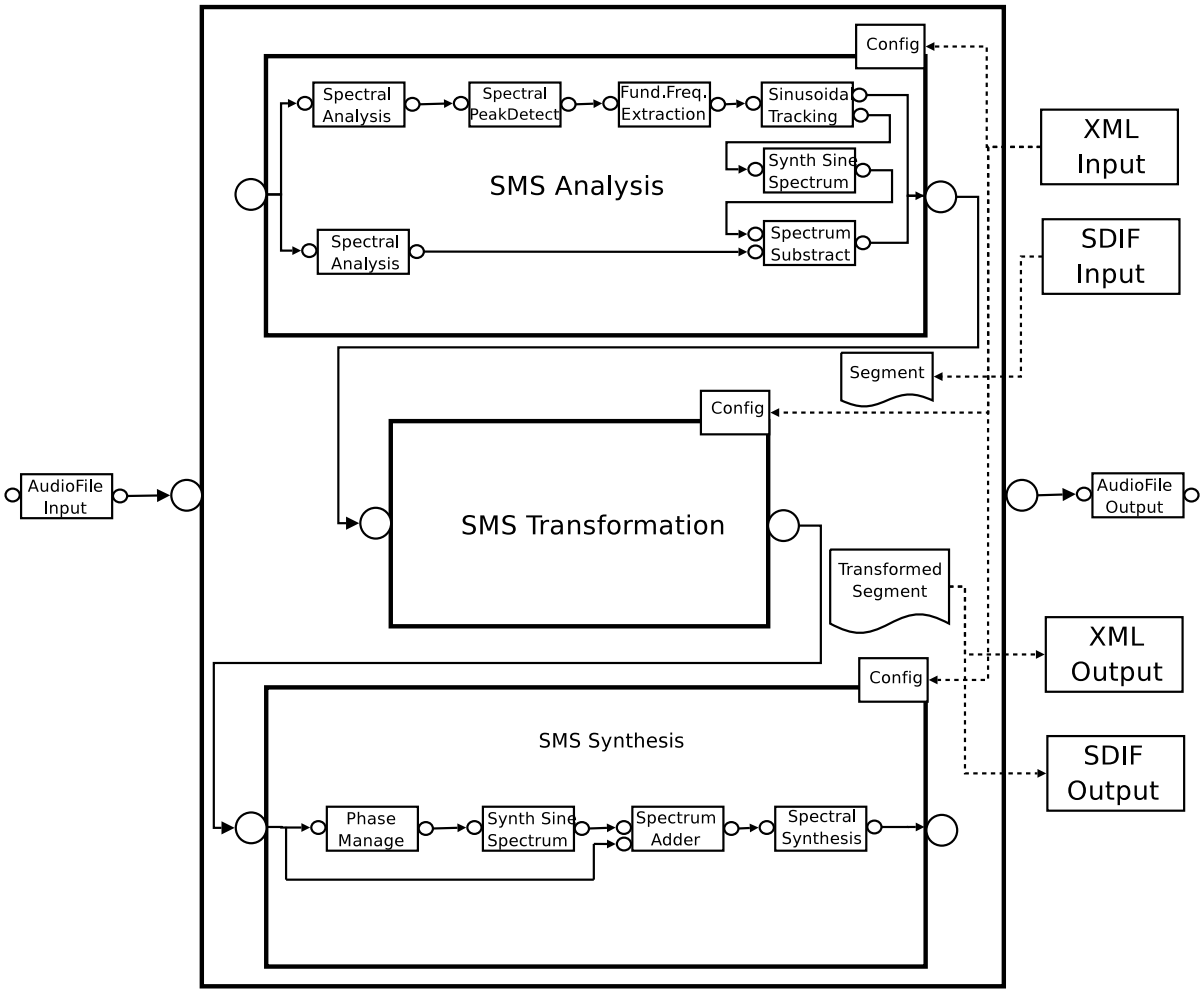

Figure 3.7: SMS Tools block diagram

```
<ResAnalysisWindowType>BlackmanHarris92</ResAnalysisWindowType>
<AnalysisZeroPaddingFactor>0</AnalysisZeroPaddingFactor>
<AnalysisPeakDetectMagThreshold>-120</AnalysisPeakDetectMagThreshold>
<AnalysisPeakDetectMaxFreq>-120</AnalysisPeakDetectMaxFreq>
<AnalysisSinTrackingFreqDeviation>20</AnalysisSinTrackingFreqDeviation>
<AnalysisReferenceFundFreq>1000</AnalysisReferenceFundFreq>
<AnalysisLowestFundFreq>40</AnalysisLowestFundFreq>
<AnalysisHighestFundFreq>6000</AnalysisHighestFundFreq>
<AnalysisMaxFundCandidates>50</AnalysisMaxFundCandidates>
<AnalysisHarmonic>0</AnalysisHarmonic>
<DoCleanTracks>0</DoCleanTracks>
<SynthesisFrameSize>256</SynthesisFrameSize>
<SynthesisWindowType>Triangular</SynthesisWindowType>
<SynthesisHopSize>-1</SynthesisHopSize>
<SynthesisZeroPaddingFactor>0</SynthesisZeroPaddingFactor>
</SMSAnalysisSynthesisConfig>
```

We will now briefly describe the different parameters involved:

Global Parameters:

<Name>: particular name for the configuration file.

<InputSoundFile>: path and name of the input sound file to analyze.

<OutputSoundFile>: path and name of where the output synthesized sound file must be stored. The application will add a "_sin.wav" termination to the Sinusoidal component and a "_res.wav" termination the residual file name. In the graphical version of the program (SMSTools) though, this parameter is not used as when the output sound is to be stored, a file browser dialog pops-up.

<OutputAnalysisFile>: path and name of where the output analysis data is to be stored. The extension of the file can be .xml or .sdif. The application will choose the correct format depending on the extension you give. Not used in the GUI version as it is obtained from the dialog.

<InputAnalysisFile>: path and name of where the input analysis data to is be loaded from. Not used in the GUI version as it is obtained from the dialog.

Analysis Parameters:

<AnalysisWindowSize>: window size in number of samples for the analysis of the sinusoidal component.

<ResAnalysisWindowSize>: window size in number of samples for the analysis of the residual component.

<AnalysisWindowType>: type of window used for the sinusoidal analysis. Available: Hamming, Triangular, BlackmannHarris62, BlackmannHarris70, BlackmannHarris74, BlackmannHarris92, KaisserBessel17, KaisserBessel18, KaisserBessel19, KaisserBessel20, KaisserBessel25, KaisserBessel30, KaisserBessel35.

<ResAnalysisWindowType>: type of window used for the residual analysis. Available: Same as in sinusoidal. Recommended: as sinusoidal spectrum is synthesized using the transform of the BlackmannHarris 92dB, it is necessary to use that window in the analysis of the residual component in order to get good results.

<AnalysisHopSize>: hop size in number of samples. It is the same both for the sinusoidal and residual component. If this parameter is set to -1 (which means default), it is taken as half the residual window size. Recommended values range from half to a quarter of the residual window size.

<AnalysisZeroPaddingFactor> Zero padding factor applied to both components. 0 means that zeros will be added to the input audio frame till it reaches the next power of two, 1 means that zeros will be added to the next power of two etc...

<AnalysisPeakDetectMagThreshold>: magnitude threshold in dB's in order to say that a given peak is valid or not. Recommended: depending on the window type and the main-to-secondary lobe relation and the characteristics of the input sound, a good value for this parameter may range between -80 to -150 dB.

<AnalysisPeakDetectMaxFreq>: Frequency of the highest sinusoid to be tracked. This parameter can be adjusted, for example, if you are analyzing a sound that you know only has harmonics up to a certain frequency. Recommended: It depends on the input sound but, in general, a sensible value is 8000 to 10000 Hz.

<AnalysisSinTrackingFreqDeviation>: maximum deviation in hertz for a sinusoidal track.

<AnalyisReferenceFundFreq>: in hertz, reference fundamental.

<AnalyisLowestFundFreq>: in hertz, lowest fundamental frequency to be acknowledged.

<AnalyisHighestFundFreq>: in hertz, highest fundamental frequency to be acknowledged.

<AnalyisMaxFundFreqError>: maximum error in hertz for the fundamental detection algorithm.

<AnalyisMaxFundCandidates>: maximum number of candidate frequencies for the fundamental detection algorithm.

<AnalysisHarmonic>: if 1, harmonic analysis is performed on all segments that have a valid pitch. In those segments the track number assigned to each peak corresponds to the harmonic number. On unvoiced segments, inharmonic analysis is still performed.

Synthesis Parameters:

<SynthesisFrameSize>: in number of samples, size of the synthesis frame. If set to -1, it is computed as (ResAnalysisWindowSize-1)/2.

<SynthesisWindowType>: type of window used for the residual analysis. Available: Same as

in sinusoidal.

Morph Parameters

<MorphSoundFile>: Optional name of the second file to do a morph on. Only necessary if a morphing transformation is planned. Note that the file to morph will be analyzed with the same parameters as the input sound file and that it must have the same sampling rate.

Apart from storing the result of the analysis, more interesting things can be accomplished. The first thing that may be interesting to do is to synthesize it back, separating each component: residual, sinusoidal, and the sum of both.

To transform your sound an XML transformation score must be loaded or created using the graphical transformation editor available in SMSTools. New transformations can be implemented and added to the CLAM repository very easily.

We will now comment how the application architecture is organized. Figure 3.9 illustrates its UML class diagram.

The main class of the application is the `SMSApplication` class. This is an abstract class (thus cannot be instantiated), but contains the core of the process flow. The three derived classes, `SMSTools`, `SMSBatch` and `SMSStdio` implement the particular versions of the base class.

So let us briefly mention what this base class holds inside. All the methods illustrated in the diagram (`LoadConfig`, `Analyze`,...) correspond to functionalities of the program that, in the case of the GUI version, are mapped directly to menu options.

The associated `SMSAppState` class is responsible for maintaining the current state of the application. The boolean (`mHaveConfig`, `mHaveInputAudio`,....) attributes of the class hold important values to control the flow of the program because they inform of whether a previous action has taken place and the desired operation can then be invoked.

The class has two `ProcessingComposite` attributes (see 3.2.2.1), instances of the `SMSAnalysis` and `SMSSynthesis` classes. These `ProcessingComposites` are configured when the global configuration is loaded and then run from the `Analyze` and `Synthesize` methods. Some intermediate Processing Data (a `Segment`, a `Melody` and different `Audio` objects) are used to hold the input/output data generated during the process. These data are then stored/played using the corresponding method (i.e. `StoreAnalysis` or `PlayOutputSound`).

Although the `SMSApplication` class concentrates most of the functionality of the application and has a great deal of operations, these methods are fairly simple and rarely need more than 10/20

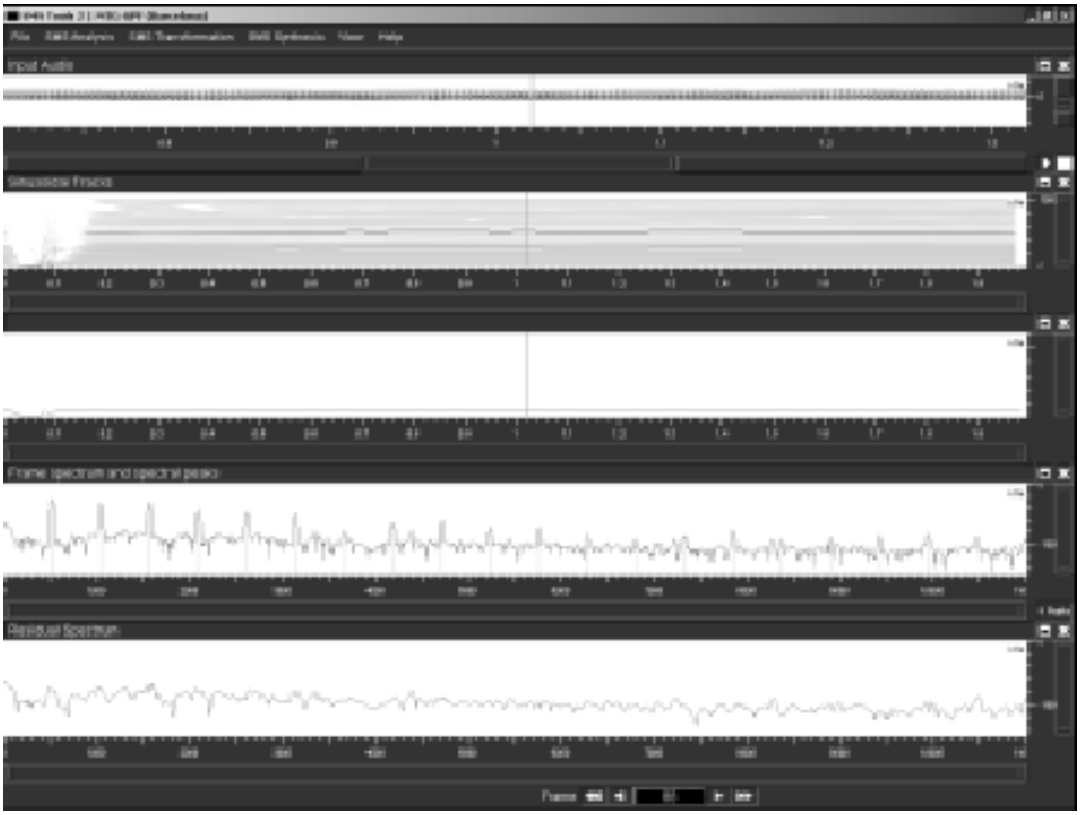

Figure 3.8: SMSTools Graphical User Interface

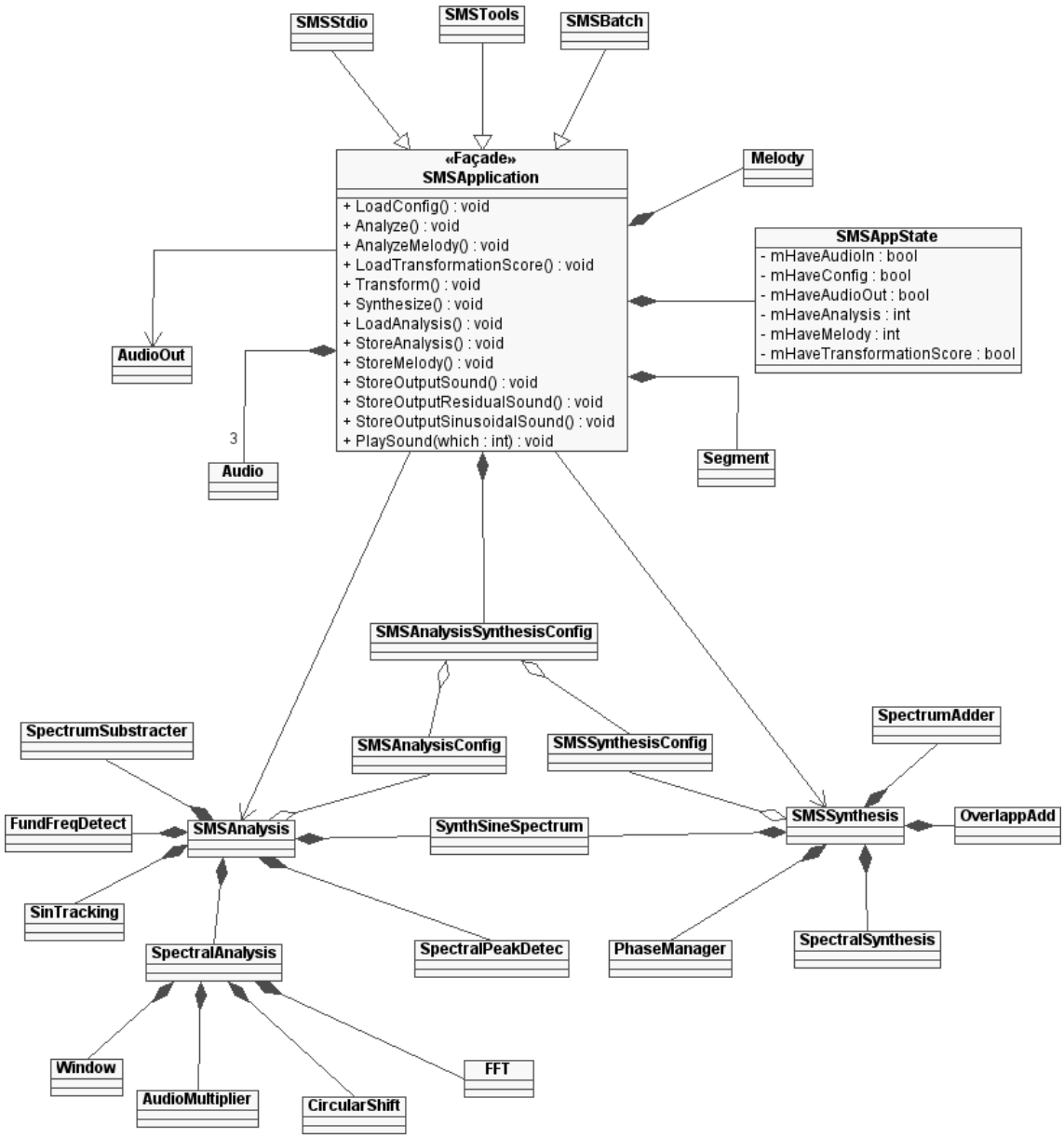

Figure 3.9: SMS tools UML class diagram

lines of code. In order to invoke an analysis, for instance, only the `SMSAnalysis::Do` method needs to be called. This is possible because these Processing Composites hide all the processing complexity. If we take a look again to the UML diagram we see that these classes contain inside a great deal of other Processing classes. Let us enumerate them and their basic functionality.

Inside the `SMSAnalysis` we have:

`SpectralAnalysis`: Performs an STFT of the sound. For doing so, it holds a number of Processing objects inside, namely a `WindowGenerator`, an `AudioMultiplier`, a `CircularShift` (for zero-phase buffer centering) and an `FFT`. Note that the `SMSAnalysis` has two instances of this class: one for the sinusoidal component and another one for the residual. This Processing Composite is quite complex in itself but we won't go into details.

`SpectralPeakDetect`: Implements an algorithm for choosing the spectral peaks out of the previously computed spectrum.

`FundFreqDetect`: Processing for computing the fundamental frequency.

`SinTracking`: This Processing performs sinusoidal tracking or peak continuation from one frame to the next one. It implements an inharmonic and harmonic version of the algorithm.

`SynthSineSpectrum`: Once we have analyzed the sinusoidal component and we have the continued peaks we have to synthesize it back to spectrum in order to compute the residual component. This is the Processing in charge of this synthesis of the sinusoidal component.

`SpectrumSubstracter2`: Once we have the sinusoidal synthesized spectrum and the original one (coming out from the residual Spectral Analysis), we can subtract them in order to obtain the residual spectrum.

The `SMSSynthesis` Processing Composite contains:

`PhaseManagement`: This Processing is in charge of managing phase of spectral peaks, from one frame to the next one.

`SynthSineSpectrum`: As already commented in the `SMSAnalysis`, this object is in charge of creating a synthetic spectrum out of the array of spectral peaks.

`SpectrumAdder2`: It is used to add the residual and the synthesized sinusoidal spectrum.

`SpectralSynthesis`: This processing composite implements the inverse STFT. That is, is the object in charge of computing an output audio frame from an input spectrum. The `SMSSynthesis` class has three instances of this class: one for the global output sound, one for the residual and one for the sinusoidal component. The `SpectralSynthesis` Processing Composite has the following processing inside: an `IFFT`, two `WindowGenerators` (one for the inverse Analysis window and one for the Synthesis Triangular window), an `AudioProduct` to actually perform the windowing, a `CircularShift` to undo

the circular shift or buffer-centering introduced in the analysis and an `OverlapAdd` object to finally apply this process to the output windowed audio frames. It is fairly complex in itself and we would need to go into too many signal processing details in order to explain it completely (see Annex B for more details on these signal processing algorithms).

§3.2.3.2 SALTO

SALTO is a software based synthesizer. It is also based on the SMS technique (see Annex B). It implements a general architecture for these synthesizers but it is currently only prepared to produce high quality sax and trumpet synthesis. Pre-analyzed data are loaded upon initialization. The synthesizer responds to incoming MIDI data or to musical data stored in an XML file. Output sound can be either stored to disk or streamed to the sound card on real-time. Its GUI allows to modify synthesis parameters on real-time. Figure 3.10 reproduces a schematic representation of the application.

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

Apart from this SDIF input, SALTO has three other possible inputs: MIDI, an XML Melody, and the GUI. Using MIDI as an input SALTO can be used as a regular MIDI synthesizer on real-time. SALTO is prepared to accept MIDI messages coming from a regular MIDI keyboard or a MIDI breath controller. On the other hand if an XML melody is used as an input this melody is synthesized back. It is the easiest way to try that SALTO is working correctly. Finally the GUI can be basically used to control the way the synthesis is going to work and to test configurations by generating single notes.

As seen in the screenshot in figure 3.11, the most important part of the interface is on the lower left: the buttons to select what part of the sound you would like to synthesize. The upper part of the interface is just a graphical display of the output. On the right there are two buttons for loading and playing an XML melody. Finally, the central part is designed to manage the database.

§3.2.3.3 Spectral Delay

SpectralDelay is also known as CLAM's Dummy Test. In this application it is no important to actually implement an impressive application but rather to show what can be accomplished using the CLAM framework. Especial care has been taken on the way things are done and why they are done.

The SpectralDelay implements a delay in the spectral domain. What that basically means is that you can divide your input audio signal in three bands and delay them separately, obtaining

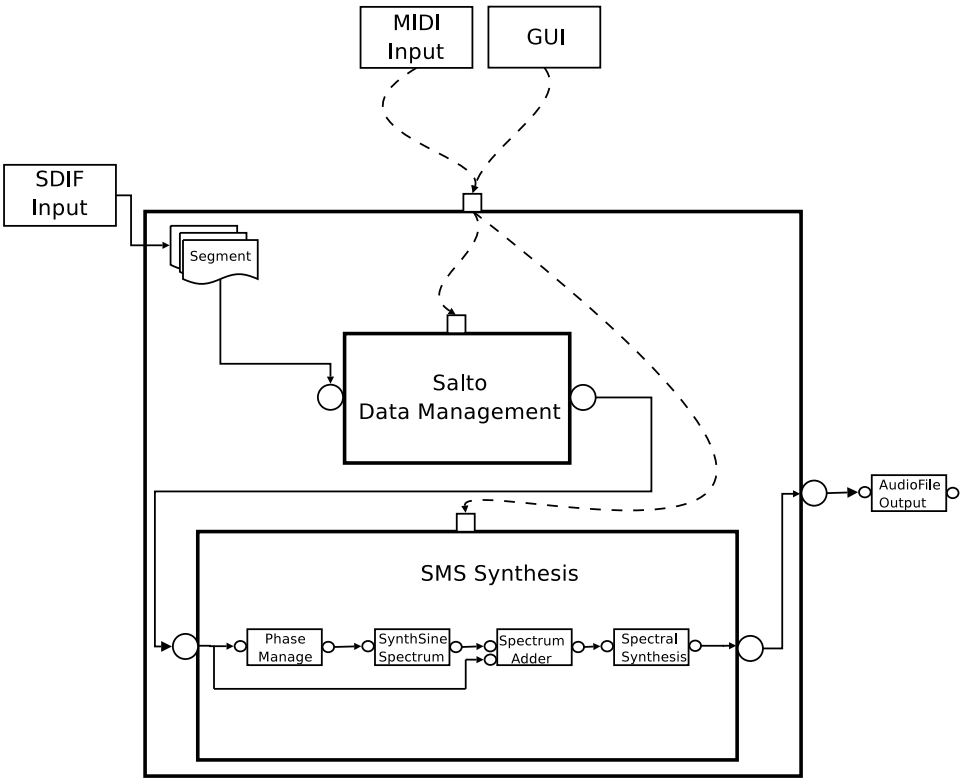

Figure 3.10: SALTO block diagram

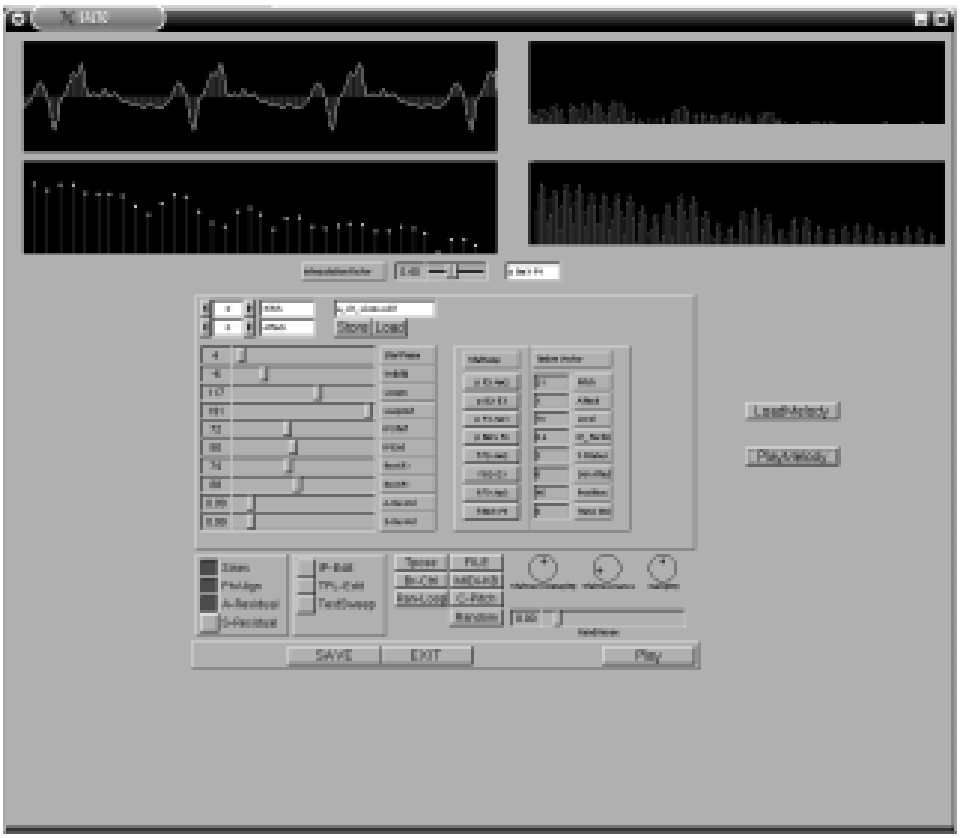

Figure 3.11: SALTO graphical user interface

interesting or weird effects. The block diagram of the process is illustrated in Figure 3.12.

The core of the process is an STFT that performs the analysis of the input signal and converts it to the spectral domain. The STFT is implemented in the `SpectralAnalysis` Processing Composite class, which contains a number of other Processing objects (i.e. `WindowGenerator` or `FFT`, see SMS Analysis/Synthesis Example UML diagram in figure 3.7). The signal is synthesized using a `SpectralSynthesis` Processing that implements the inverse process. It is transformed, in between these two steps, in the spectral domain.

The output data of the `SpectralAnalysis` is read by three `AudioMultiplier` Processing objects that also take the spectral transform transfer function of a pre-defined filter as input. As a matter of fact we apply three different filters: a low-pass, a band-pass and a high-pass. We then have the signal divided into three different bands. Each of them is delayed with a different delay time. Finally, and before the synthesis, these three bands are summed up again.

The graphical interface depicted in Figure 3.13 controls the frequency cut-offs and gains of the filters and the delay times of the delays.

§3.2.3.4 Network Editor

The Network Editor is one of CLAM's most important applications. It is a graphical application for directly interacting with CLAM's graphical model of computation by editing a `CLAM::Network` object (see section 3.2.2.3). It can by itself be an environment comparable (conceptually, not in features) to graphical environments such as Max (see section 2.5.2). Nevertheless, in the context of the CLAM framework, we consider that it is no more than a rapid prototyping tool that can help in modeling signal processing systems that can later be optimized and converted into real applications.

The Network Editor has not been designed with immediate artistic or musical usage in mind. Nevertheless, with slight modifications, it could clearly have application in this field and some experiences in this direction are already under way.

The Network Editor interface (see figure 3.14) is divided into two parts. On the left side there is list of available Processing classes while on the center to right side there is a graphical representation of the system under study.

The list of Processing classes is obtained from the CLAM framework *factory register*, which is a central repository list of available Processing classes. So, in order to be able to use a Processing class from within the Network Editor it must first be registered in the factory. This represents an inconvenience when writing new classes as the application has to be rebuilt. For that reason, dynamic loading of pre-compiled libraries that would allow a plug-in like functioning is on the todo list. Nevertheless the

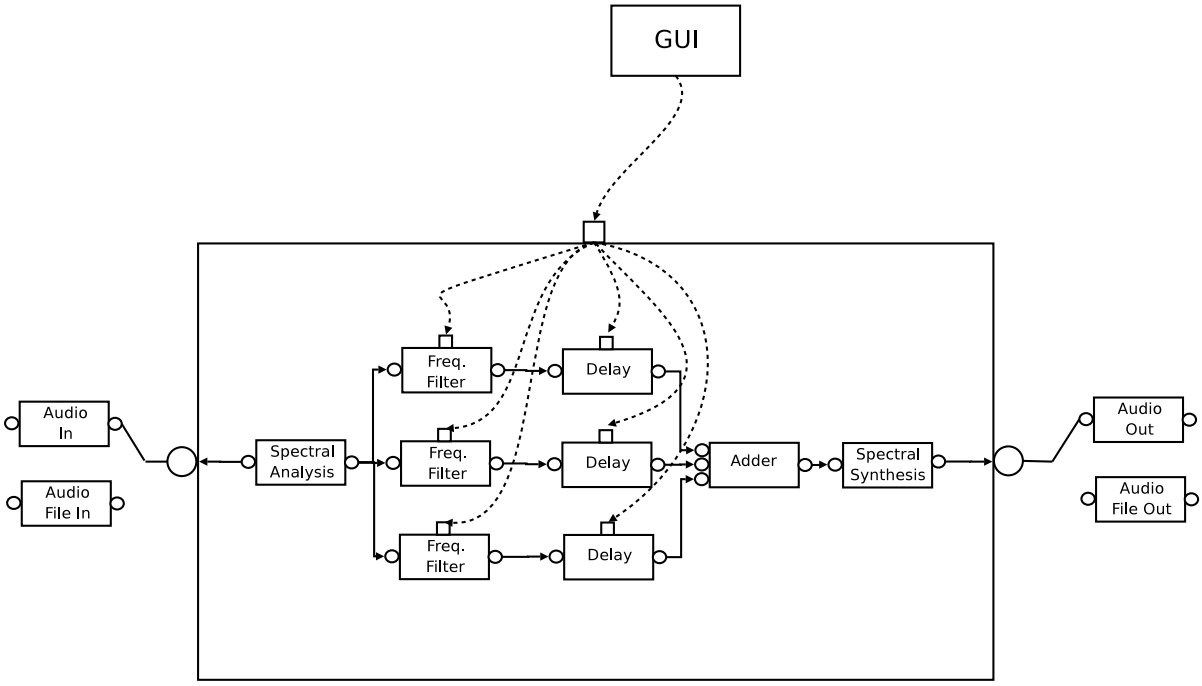

Figure 3.12: Spectral Delay block diagram

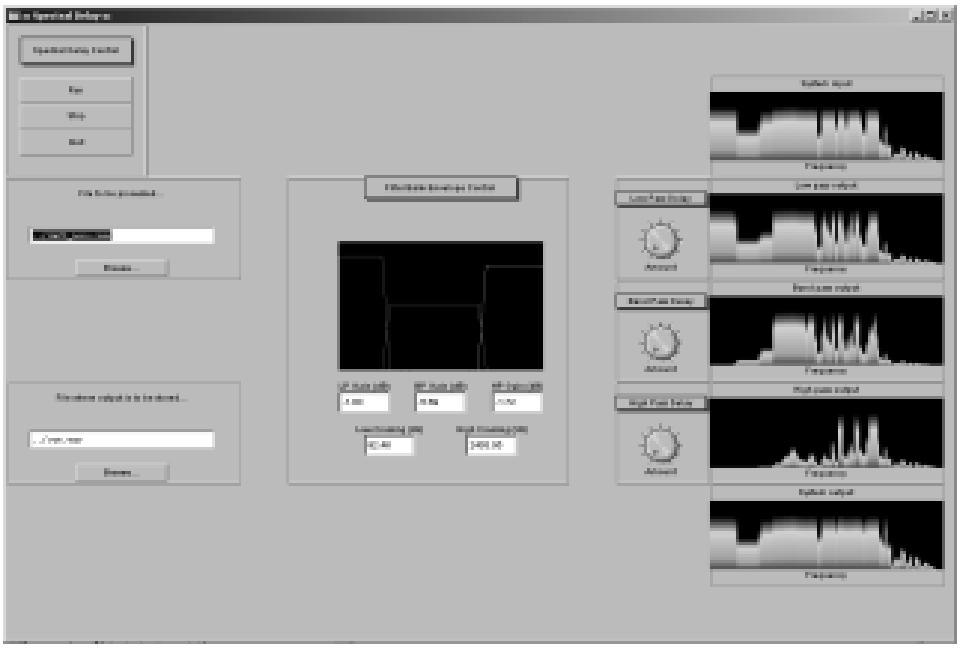

Figure 3.13: SpectralDelay Graphical Interface

Network Editor does allow to load generic Ladspa plugins. Finally, another interesting property of the available class list is that it automatically classifies classes into categories using the available class information.

The graphical representation of a Processing object that is inserted in the graph is also obtained automatically from the class definition. A Processing object is represented by a rectangle that includes the class name on the low inner half. The object name that appears on the top inner side can be provided by the user or else is generated automatically. Input/output ports are represented as small rectangles, input ports located on the left and output ports on the right side of the Processing object. Controls are represented by small triangles, input controls on top of the Processing object and output controls below. and a graphical representation of input/output ports (as small rectangles) and input/output controls (as triangles).

Data connections linking input and output ports are represented by a thick curved line while control connections are represented by a thin line in a different color. The path of the connection is determined automatically and, for the time being, cannot be modified by the user who is just in charge of deciding the input and output ports to connect.

When double-clicking on any Processing object representation an automatic *configurator* object is shown. This graphical widget allows the user to change the configuration of the given Processing object.

The resulting Network description can be stored into XML. This XML file is composed of two parts: the processing Network description and the graphical layout. It therefore acts as a project file that can be loaded afterwards to recall the exact project state that was stored.

The application does not generate code, and therefore nor a compiled binary, but this feature is planned for future releases.

§3.2.3.5 Others

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

The Time Machine project implemented a high quality time stretching algorithm that was later integrated and included in a commercial product [Bonada, 2000]. The algorithm uses multi-band processing and works in real-time. It is a clear example of how the core of CLAM processing can be used in isolation as it lacks of any GUI or audio input/output infrastructure.

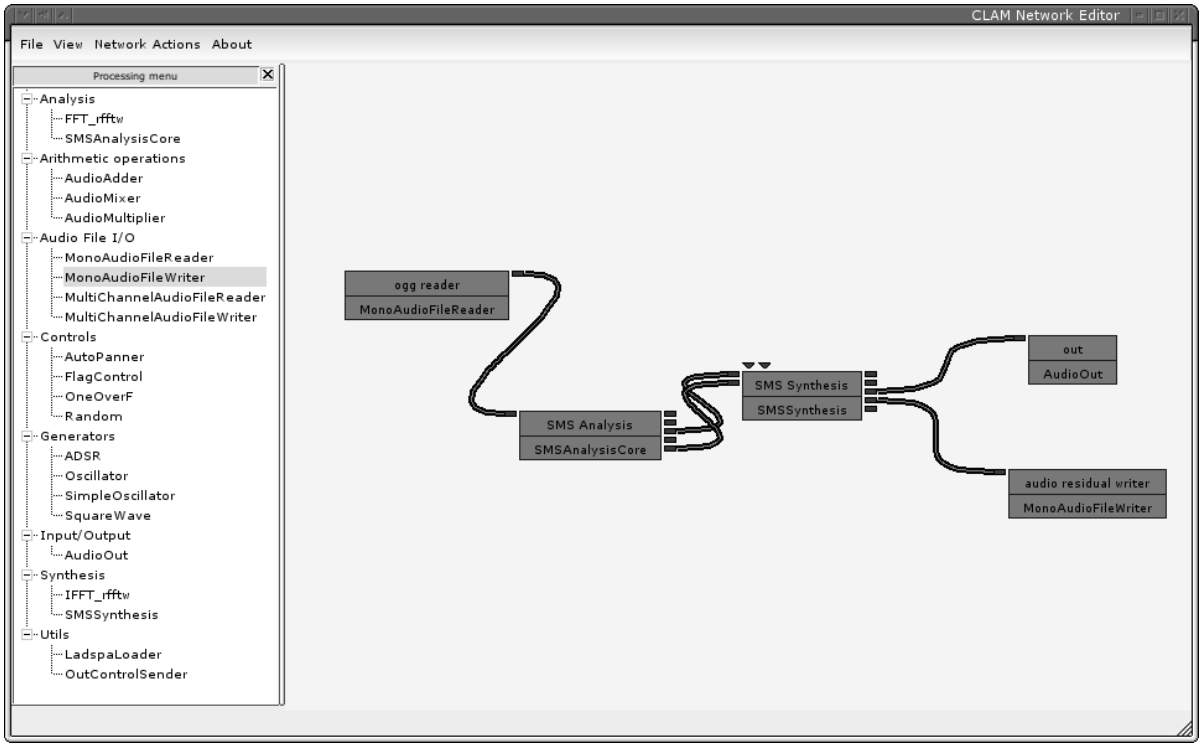

Figure 3.14: Network Editor Graphical Interface

The Vocal Processor (see figure 3.15) is a prototype also developed for a third party. It is a VST plug-in for singing voice transformations. It includes transformations such as smart harmonization, hoarseness, whispering or formant change. This prototype was a chance to test CLAM integration into VST API and also to check the efficiency of the framework in highly demanding situations (see 1.1.5). Most transformations are implemented in the frequency domain and the plug-in must work in real-time consuming as few resources as possible.

The CUIDADO IST European project [Vinet et al., 2002] was completely developed with CLAM. The focus of the project was on automatic analysis of audio files. In particular rhythmic and melodic descriptions were implemented. The CLAM code was integrated as a binary dll into a commercial product named the Sound Palette (see figure 3.16). The algorithms and research applications are currently being integrated into the CLAM project and incorporated into standalone sample applications such as the Swinger, an application that computes rhythmic descriptors from a sound file and applying a time-stretching algorithm is able to change the *swing* of the piece (see figure 3.17).

The Open Drama project was another IST European project that used CLAM extensively. The project focus was on finding new interactive ways to present opera. In particular, a prototype application called MDTools was built to create an MPEG-7 compliant description of a complete opera play (see figure 3.18).

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

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

Finally, CLAM is also currently being used in different internal projects that will also someday be integrated. Probably the most important is the SIMAC IST European project. The software output of this project will be three prototypes: a music annotator application, a music collection organizer and browser, and a music recommendation engine. All of them are being developed in CLAM.

Rappid [Robledo, 2002] is a testing workbench for the CLAM framework in high demanding situations. The first version of Rappid implements a quite simple time-domain amplitude modulation algorithm. Any other CLAM based algorithm, though, can be used in its place. Next picture illustrates the basic diagrams of the application. The most interesting thing about Rappid is the way that mul-

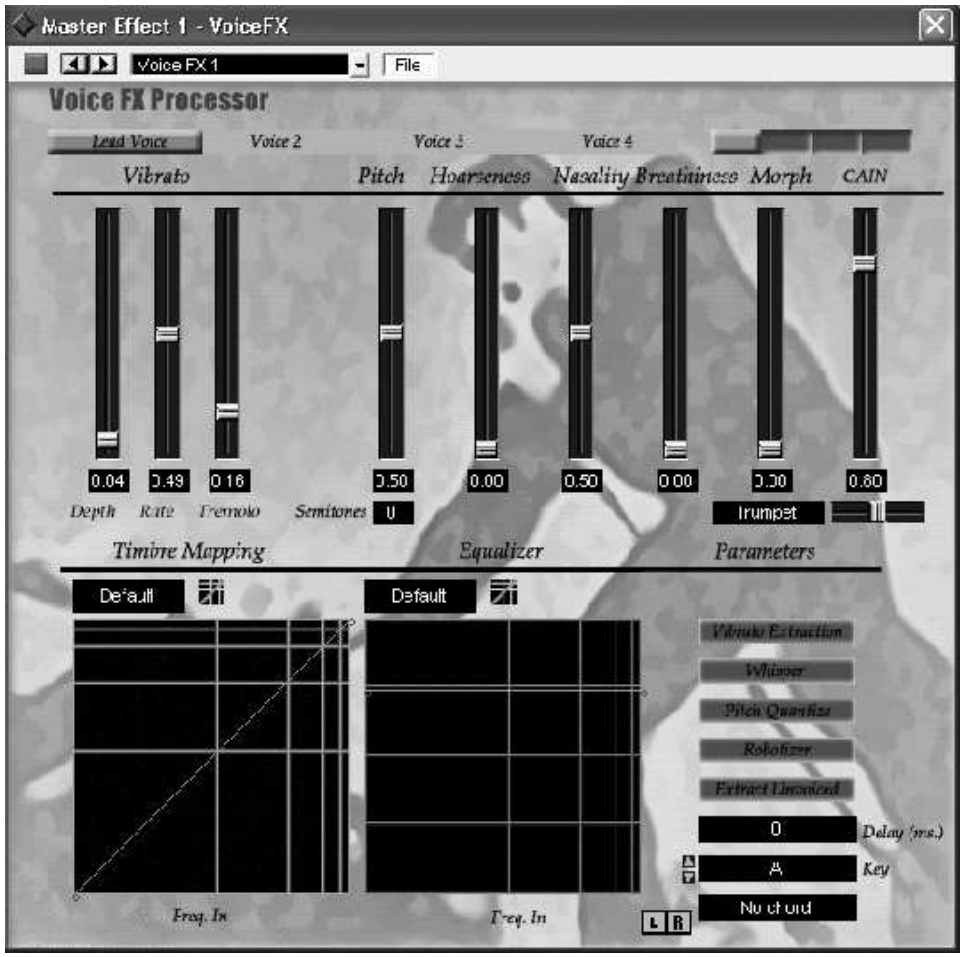

Figure 3.15: The Vocal Processor

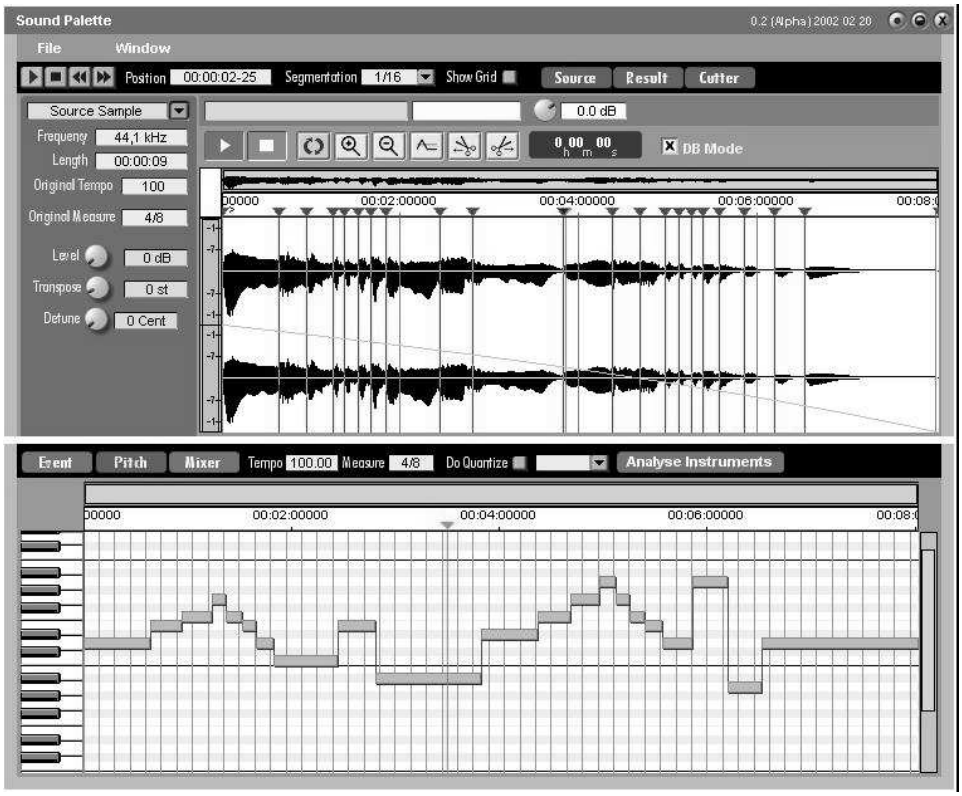

Figure 3.16: The Sound Palette

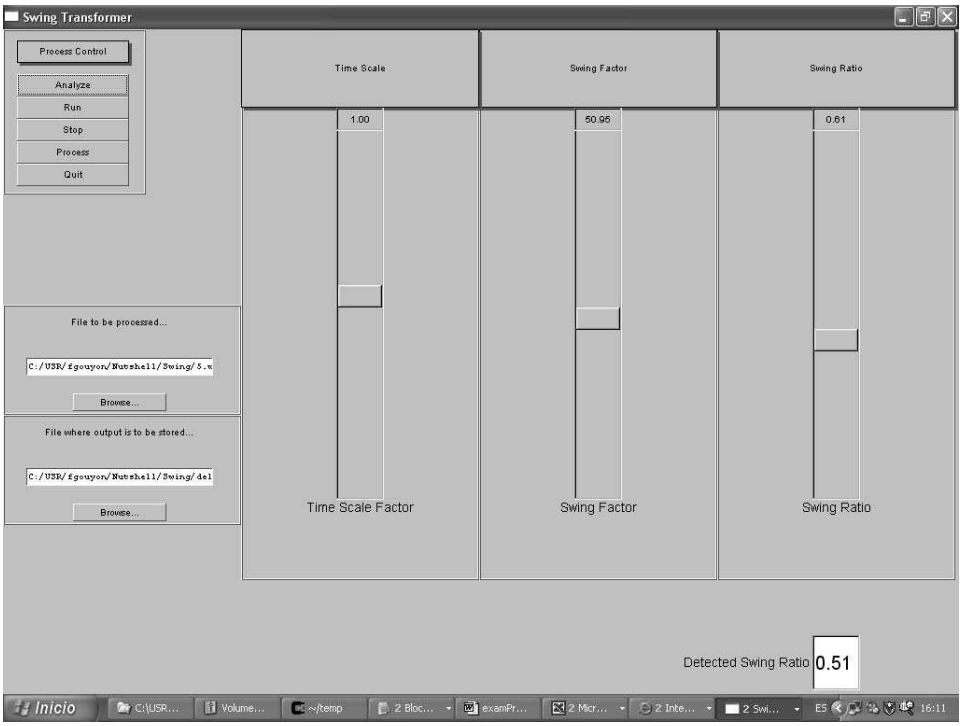

Figure 3.17: The Swinger

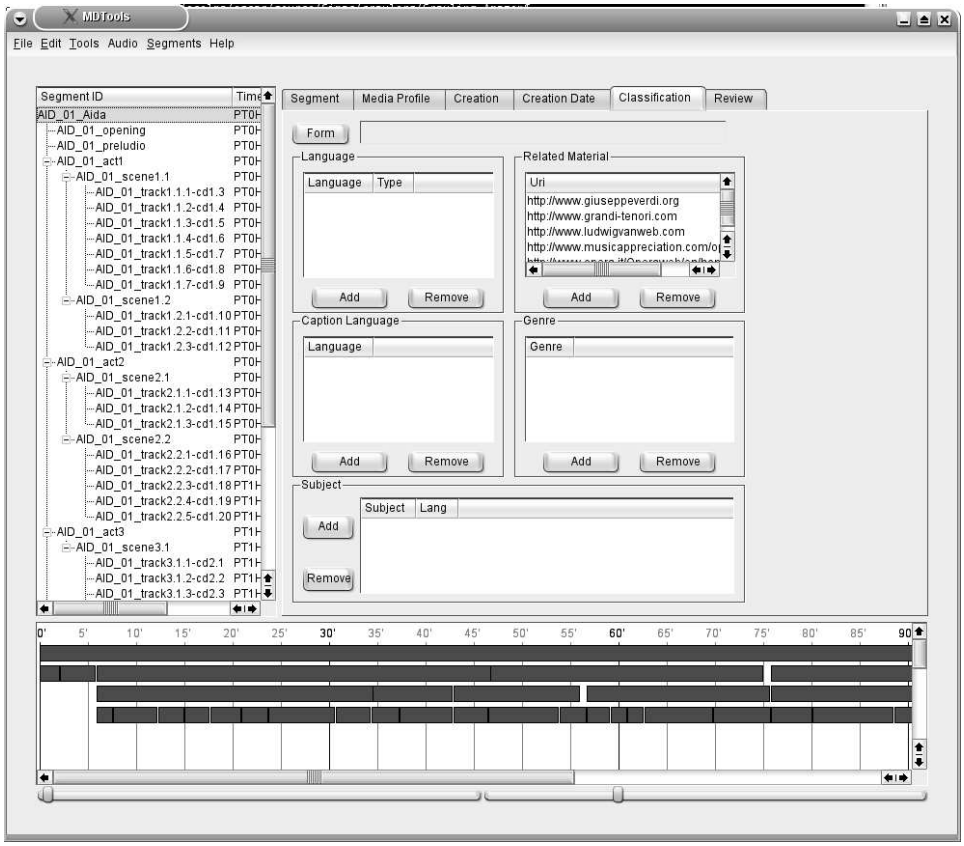

Figure 3.18: MD Tools

tithreading issues are handled, using a watchdog mechanism. The current implementation works only under GNU/Linux. Rappid has been tested in a live-concert situation. Gabriel Brnic used Rappid as a essential part of his composition for harp, viola and tape, presented at the Multiphonies 2002 cycle of concerts in Paris.

§3.3 Is CLAM different?

The goal of this section is to compare the CLAM framework to similar existing tools, particularly those introduced in chapter 2.

First, we will classify and compare the general features of the framework for later concentrating in more particular features that have driven our design process.

§3.3.1 CLAM classification

Out of the different categories presented in chapter 2, CLAM fits better into the "Audio Processing Frameworks" one, in the subcategory of "General Purpose" (see 2.3.3). Nevertheless it shares some properties with environments in other categories and has some particular features that make it different from the other frameworks included in this subcategory.

Let us first highlight the similarities that CLAM shares with other environments not in its subcategory.

First, CLAM is a framework with a clear focus on audio and music but its generic, open, and flexible model and design make it suitable for processing any kind of signal data. Particularly, there are plans to use the framework for image and video processing in the short term. Therefore, it would be possible that, in the mid-term, CLAM would fall into the first "General Purpose Signal Processing and Multimedia Frameworks" category (see 2.2).

Also CLAM is an object-oriented *application framework*[4] as defined in section 1.3. Out of the different audio and music environment reviewed, only MET++ (see 2.2.4) and CSL (see 2.3.3.1) could be naturally defined as application frameworks.

But although we have just classified our framework as a "General Purpose Audio Processing Framework", it includes many analysis tools and features only comparable to the two frameworks included in the "Analysis-Oriented Audio Processing Frameworks" category (see 2.3.1), and particularly the most mature one, Marsyas (see 2.3.1.1).

[4] Although CLAM is clearly an "application framework" it could also in many senses classify as a "domain framework".

Furthermore, even though CLAM is probably more oriented toward audio signal processing than toward musical symbolic data manipulation there are clear examples and applications in which such a data is used extensively. Nevertheless it is clear that CLAM does not, as most of the environments discussed in section 2.4, present a particular model of Music. In this sense it is very similar conceptually to Max (see 2.5.2) although CLAM for being a framework it allows to extend it and integrate different music models in it. In chapter 6 we introduce an Object-Oriented Music model that has been implemented in the framework.

CLAM is not a graphical application, neither is this one of its main features. For that reason it would make no sense to include it in this category (see 2.5). Nevertheless it includes a graphical "network edition" tool similar to that present in all the environments in this category (see 3.2.3). This tool, though important, is seen as just a particular instantiation of the framework and different graphical representations with distinct features could be developed and integrated with not much effort.

Finally, CLAM is definitely not a Musical Language (see 2.6) but it includes one (explained in chapter 6) as a particular instance of the general framework.

In figure 2.1 we presented a graphical representation of how the different environments presented in that chapter classified. In figure 3.19 we illustrate how CLAM would be positioned in respect to the other environments. CLAM extends its scope from the general purpose signal processing category and into the music processing realm, focusing on all audio processing applications, from analysis to synthesis[5]. We have also added to the figure CLAM's particular Visual Builder: the Network Editor. We have done so in order to stress the fact that this particular CLAM application is comparable in its scope to graphical applications such as Kyma or Max.

§3.3.2 CLAM and other environments

In the previous section we have classified CLAM as being a General Purpose Audio Processing Framework. Nevertheless, we have also mentioned that it shares some features with other environments classified in different categories. We may now question whether the overall philosophy of CLAM is adequate or not. Is it really necessary to build an application framework? Wouldn't it be better and simpler to offer a graphical builder à la Max or Kyma? On the other hand, wouldn't it be better to design a completely new programming language more adapted to our domain à la Supercollider?

It is important to remember that CLAM's focus is on research and development of applications. It is not an artistic or creative tool (although such a tool could be developed within the framework).

[5]Note that in the original figure the size of the bounding box is not used as an indication of how broad the scope is. Nevertheless we have used this parameter now to highlight which of the existing environments CLAM could substitute, scope-wise.

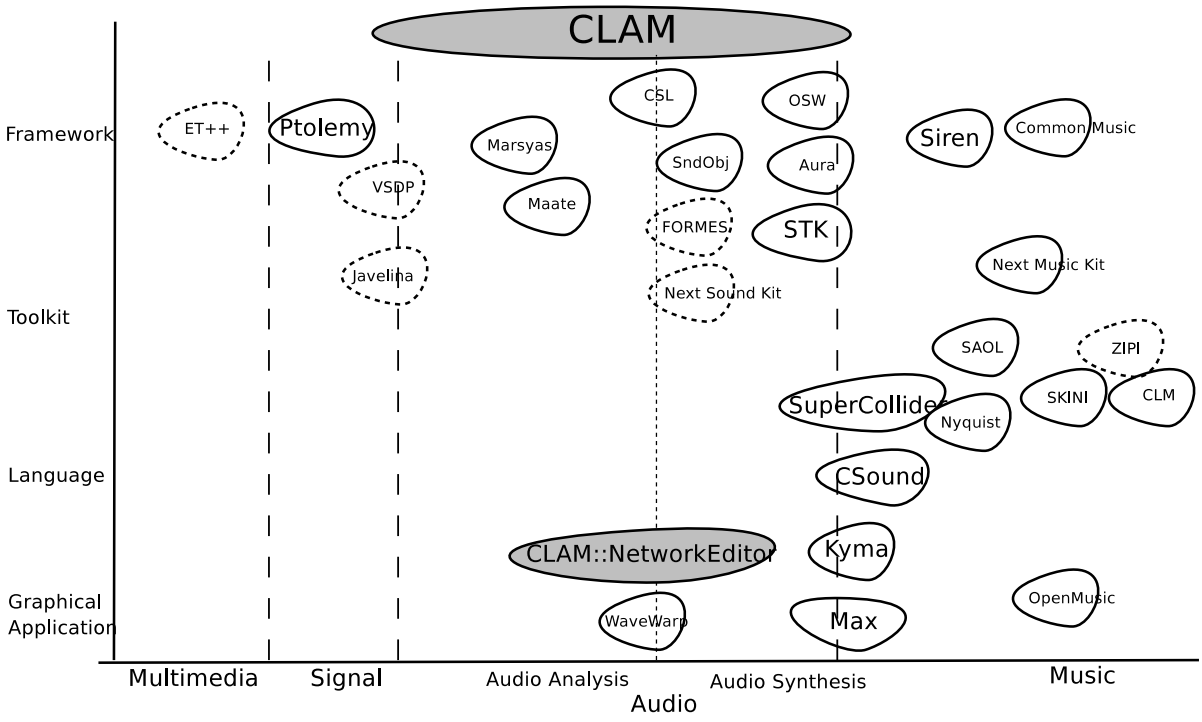

Figure 3.19: CLAM classification in respect to other environments

For developing efficient applications and for doing research on low-level details of the algorithms it is necessary to access the programming language level. Visual programming is indeed of great help and ideal for prototyping, and as already mentioned in 1.3 it is a natural evolution of any application framework. On the other hand, as R. Dannenberg points out [Dannenberg, 2004] the "boxes and arrows" representation for signal processing systems has stood the test of time. This is no surprise as this representation is directly related to the graphical MoCs explained in section 1.5. That is why CLAM offers the Network Editor (see 3.2.3), its particular graphical MoC visual builder, but this is not designed to substitute the code level.

On the other extreme, one may have the temptation to implement a full-fledged programming language, adapted to this particular domain. This would be an interesting idea to explore if it had the slightest chance of being viable. A programming language has many associated tools, from editors with syntax highlighting to debugging and profiling tools. Offering a completely new programming environment based on a new language is a titanic effort that needs of a very large development team. On the other hand, the language has to offer very unique and outstanding features in order to convince new users that the effort of learning it is worth the while. Supercollider represents the clearest example of such an approach. An now its author, James McCartney questions whether that was a good decision. In [McCartney, 2002] he wanders whether specialized computer music languages are necessary. He concludes that the set of abstractions available in general purpose computer languages are sufficient to build useful dedicated frameworks such as CLAM. This is also our opinion, no specialized computer language is needed, it is much wiser to build tools and frameworks around already existing and proven languages.

As for CLAM's model of computation it is interesting to compare it to three environments that have a sufficiently long tradition and that clearly state their conceptual model[6]: Max (see 2.5.2), Siren (see 2.4.2) and Kyma (see 2.5.1). To our purposes they also represent the extremes in between which CLAM's MoC could stand. On the one hand, Max offers a highly intuitive graphical (and graph-based) MoC while Siren and Kyma offer a very formal object-oriented approach[7].

It has already been commented that the main feature of CLAM's MoC is that it is object-oriented. Object-oriented techniques have been applied throughout the development process and we have always followed Alan Kay's maxima of "everything is an object". As already outlined in chapter 1,

[6]For the purpose of our discussion we will here highlight differences and similarities in respect to environments in different categories than CLAM's. In next paragraphs we will point out how similar is CLAM's MoC to the one that other environments in its category present.

[7]Note that we are here referring to the underlying models of computation, not the way that they are actually represented in the graphical interface. In the particular case of Kyma, its complex and sometimes fundamentalist object-oriented conceptual model, which was defended as its main feature in past years is nowadays usually hidden to the user which has access to different views of the system that include temporal and graph-based representations.

we believe that the object-oriented approach not only has many advantages but it also represents the most recommended paradigm when modelling a system. We believe that most opinions against object-orientation come from misunderstanding some basic concepts related to the object-oriented paradigm. It is interesting to note that already in early frameworks such as FORMES (see 2.3.3.3) the benefits of object-orientation were reported. In [Rodet and Cointe, 1991] the author, Xavier Rodet, states that "object-orientation matches most of the requirement of Music Composition Synthesis".

The case of Max is a clear example of this. The authors state that Max is not a dataflow language but rather represents an object-oriented approach [Puckette, 1991a]. Nevertheless they do not use an object-oriented programming language like C++ and decide to base their developments in the C language. Because of this they implement a limited form of polymorphism themselves (see [Puckette, 1991b]) although recognizing that this method has much overhead over using one in a language like C++.

The reasons for not using C++ do not appear clear but two reasons are at least outlined. On one hand in [Puckette, 2002], Miller Puckette, the main author of Max argues against object-orientation in general saying that the transmissibility of practice and ideas does not depend of code reuse, and inheritance is essentially about code reuse. Although the author states that this is a general opinion in the computer science community, we cannot agree with him. The benefits of object-orientation in general and inheritance in particular are always set on two grounds: reuse of code and, most importantly, reuse of concepts. Reuse of code plays no role at all in very important mechanisms related to inheritance like abstract methods and classes or interfaces. Code reuse is a valuable side-effect of inheritance, but not its most important benefit. The author also states that "(...) elegance matters but it has nothing to do with the creative adaptation of the underlying ideas". Again, we cannot agree with this opinion. Elegance, understood as a property related to well-structured code and design, is indeed a crucial point for adaptation of any kind. In order to adapt an idea, a concept, or a piece of code, the first thing we have to do is to understand it. And it is much easier to understand an elegant idea, concept or a piece of code.

On the other hand in [Zicarelli, 2002] we find another reason. The author (David Zicarelli in this case) exposes a rather confusing view about dynamic binding. According to him Max uses dynamic binding while C++ uses static binding. We cannot accept this assertion. C++, as most programming languages, accepts both static and dynamic binding (see section 1.1.3 for an explanation on these concepts). As a matter of fact dynamic binding is the base of two of the four different kinds of polymorphism introduced in 1.1.3. It is true that C++, differently to other object-oriented languages like Smalltalk, uses strong typing, but that has nothing to do with the discussion. Symbols can be

associated to functions on run-time and the mechanism implemented in Max, based on a pointer to a list of symbols, is simply a primitive version of C++ polymorphism and the pointer to virtual function table that is added to any polymorphic object.

The main drawbacks of the Max model that are reported by other music environments developers are the problems it presents related to extensibility and portability (see [Chaudhary et al., 1999] or [Cook and Scavone, 1999]). In this sense it is also interesting to note how the supposed dynamic features in Max are the reasons why some authors qualify it as a basically static environment. Stephen Travis Pope states in [Pope, 2004] that "Max (...) provides some object-oriented features, including dynamically typed data, dynamic binding but no inheritance, limited data types, only one data structure for inter-object messaging and few control structures. Max has a static object structure and although jMax and Pd do something about it they are still generally static."

The authors mention that there is a disagreement in the community whether Max should be considered object-oriented [Puckette, 2002]. Max uses the object-oriented message passing mechanism but not anything else. Because of this and all the reasons previously exposed it is our opinion that Max is not object-oriented.

On the other hand a clear difference between Max or similar environments and CLAM is that their design makes it difficult to process complex data structures. Max's focus is on real-time performance and therefore can handle very well simple control events but its design does not accept naturally other kind of data such as Spectrums and the like.

Finally possibly the biggest diference between Max and CLAM is that CLAM, as it has already been explained, is not a framework but rather a Visual Application.

Siren (see 2.4.2) is another example of a long standing and well-established paradigm that started with the MODE. Siren is a music oriented framework written in Smalltalk and heavily influenced by the "Smalltalk-way-of-doing-things". The motivation for creating the MODE and Siren was "to build a powerful, flexible, and portable computer-based composer's tool and instrument". It therefore represents a different approach from that of CLAM as it is completely music-oriented.

Siren presents a complete and mature object-oriented approach and it is indeed one of the main technical goals of the framework [Pope, 2001]. But the model that is used is not modelling a particular domain (music and music composition) but rather the author's point of view and requirements. In [Pope, 2001] he recognizes that although he is not the only user that he has listened to, this assertion is not completely false: the framework has evolved through many years to meet the needs of the author for a particular musical composition.

Now Siren has been extended to other applications rather than simply musical composition

and it is also being ported to different languages (see more details in 2.4.2). The former possibility does not seem a plausible line of future. The author is promoting another more generic framework in parallel (see CSL in section 2.3.3.1) and we see this as the effort with most guarantees. Nevertheless, the fact that it is being ported to different languages opens up a completely different view as the framework model has always been very much related to that of Smalltalk.

Kyma represents a similar approach, it is not strange as it has also been developed on the Smalltalk environment. Kyma presents a rather extreme object-oriented musical model (see section 2.5.1) where everything is considered a sound object, from a single timbre to the structure of the whole composition or a transform of several sounds. This underlying model is defended in a rather romantic way as the core and fundamental issue in Kyma. The truth is that the final user has no access to the source code and, on the other hand, the graphical interface has evolved in a completely different direction, now basically focusing on the temporal and graph-based views. The "fundamental" model is completely transparent to the final user and is not even mentioned in the program description (see [www-SymbolicSound,])

It is our opinion that Kyma's object-oriented model is a clear example of overdesign. The conceptual model is not clear and, in some sense, is "too much object-oriented"[8], forgetting about the domain that is being modeled and the way that people understand it. There has been a clear tension between the author's model and what the users demanded. This has forced the graphical interface to grow in other directions in order to hide the model. It is clear not the way to go, as these modifications should also have to reach the underlying model.

The fact that CLAM has followed an evolutionary, application and user-driven development process leads to the fact that the framework has not been built around a pre-existing conceptual model but rather the model has emerged as a result of the development process. It has already been stated in 1.3.5 that it is our opinion that frameworks generate metamodels and not the other way around. As already noted, this approach is completely different from that of most musical environments (i.e. Kyma, Siren or Max), which depart from the author's model and build the environment around it.

§3.3.2.1 CLAM as an Audio Processing Framework

In the previous paragraphs we have seen that CLAM presents substantial differences in respect to some other environments approaches. Nevertheless, all those compared environments are not in the same category (General Purpose Audio Processing Framework) than CLAM so these differences may in

[8]By this we mean that the author has been too traditional applying the OO model, which originally did not recommend a separation between data and operations or processes. This already deprecated idea may lead to quite artificial domain models. In section 4.2 we will comment more on this idea and justify why our model is truly object-oriented.

some sense be justified. The previous comparison has been useful to justify CLAM's overall approach.

We will now justify its detailed approach and some design decisions by commenting on the main conceptual similarities and differences that CLAM presents in respect to the different environments in its subcategory. These are: the Create Signal Library (CSL), Open Sound World (OSW), Synthesis ToolKit (STK), Aura, SndObj, FORMES and the NeXT Sound Kit. We will leave the last two out of the general comparison only including them when necessary. Both FORMES and the NeXT are no longer in use, have just been included for historical completeness and particularly the latter has a slightly different focus aiming at providing operating system level audio tools.

Let us first comment what are the main similarities between these frameworks and CLAM. It is important to highlight that most of the frameworks in this category have goals very similar to CLAM, particularly STK, CSL, SndObj and FORMES explicitly recognize subsets of CLAM's goals as exposed in section 3.1. In this sense, for instance, it is interesting to note that all of them are implemented in C++ and aim at being cross-platform (although some of them have still reached this goal). They are also all open source although this assertion is redundant for a framework as you always need the source code to build and extend applications.

In a more conceptual ground, all of them are also object-oriented and offer some sort of graph-based model in which processes are nodes of the graph. The concept of *Processing objects* in CLAM has more or less direct equivalents in all of them: they are called *unit generators* in CSL, *transforms* in OSW, *instruments* in STK, either *unit generators* in Aura, *sound objects* in SndObj, and *processes* in FORMES. Note that this concept is not exclusive of this category of environments but is rather an idea that is repeated in many other environments (they are called *transformations* in Marsyas, *transforms* in Kyma, *objects* in Max, or *EventGenerators* and *EventModifiers* in Mode).

In all of these frameworks, the "processing objects" are connected in some way in order to build a more complex "network" that conforms the base of a given application. As mentioned in this section CLAM offers two different mechanisms for composing with processing objects. If the composition is static we call the result a *Processing Composite Object* while if it is dynamic we call it *Processing Network*. In many aspects CLAM's Processing Composites are equivalent to Aura's *instruments* or FORMES' relation between parent and children processes while CLAM Processing Networks are like CSL's, OSW's or Max's *patches*.

We will now comment the main differences that CLAM presents in respect to these same frameworks. As a general difference, it must be noted that all of them have followed a development process that differs from CLAM's. As a matter of fact, all of them can be considered "one-man-systems", they have been thought out, designed and even developed by one or two people: CSL by Stephen Travis

Pope, Open Sound World by Amar Chaudray, STK by Perry Cook and Gary Scavone, Aura by Roger Dannenberg and Eli Brandt, SndObj by Victor Lazzarini, FORMES by Xavier Rodet and Pierre Conte and the NEXT Sound Kit by M. Lentezner. None of them has had such a large development team as CLAM's (see annex A for more practical information on this issue).

But most importantly, none of them has a defined or explicit policy of acknowledging and adding user feedback into the development life cycle. Moreover, none of them declares having a truly incremental or agile process methodology and the number of releases of the frameworks are much less frequent than in CLAM.

The CSL library (see 2.3.3.1) is still not mature enough and as the authors recognize their experience with the C++ programming language is rather limited and the framework needs further design refactorings [Pope and Ramakrishnan, 2003]. On the other hand although CSL is clearly object-oriented and presents a clean design, the graphical model of computation is not explicit and ends-up being a bit confusing.

Open Sound World (see 2.3.2.2) is, out of all these frameworks, the one that probably presents a cleaner and most mature design. It is clearly object-oriented and the graphical model of computation is clearly stated. It is efficient and offers many tools. Nevertheless, it has some differences with CLAM that should be noted. OSW goal is not to become an application framework but rather to offer a music composition tool ala Max. Therefore by aiming at being at artistic/creative tool, its focus is clearly not that of CLAM, which is to offer a research/development environment. On the other hand, OSW is mostly a "one-man system" and is therefore mostly synthesis-oriented. This developer is no longer working on the framework and although some other people work on it, it is not very active nor updated regularly.

Although the Synthesis Toolkit or STK (see 2.3.2.1) is also a "one-man system" (or more exactly "two-men") it has a long history and it is still updated and maintained on a regular basis. Nevertheless, it presents a fundamental difference with CLAM in being clearly synthesis-oriented. STK offers very few tools for audio or music analysis. Another difference is that in STK there is no clear distinction between process and data, this is possibly a feasible decision for a synthesis-only application but not so if data can be the result of a previous analysis process.

The Aura framework (see 2.3.2.3) is Free Software and is available from the author. Nevertheless, at the time of this writing Aura does still not have a publicly supported version because of lack of documentation and because of its steep learning curve. In its current state it is not truly cross-platform as it is only being developed and tested on the Windows platform. Aura aims at offering an efficient real-time implementation, not only for audio but for general real-time applications. Because of this

the framework sometimes compromises the understandability and easiness of use of the model. Other practical differences are that Aura only operates on fixed size data chunks of audio (it would be difficult to integrate other data such as spectrums) and there is no clear distinction between control and signal data.

Finally, SndObj (see 2.3.3.2) is not a very mature framework and does not offer many tools or examples. This is so because SndObj is the most clear example of an strictly speaking "one-man system". SndObj is object-oriented and graph-based but its model is not very clear. On the other hand it does not focus on efficiency issues and it is not likely to work on complex real-time situations.

§3.4 Summary and Conclusions

In this chapter we present the CLAM framework. This software framework is a comprehensive environment for developing audio and music applications. It may be also used as a research platform for the same domain. CLAM can be seen both as the origin and the prove of concept of the conceptual models and metamodels that are included in this thesis.

CLAM is written in C++, it is efficient, object-oriented, and cross-platform. It presents a clean and clear design result of applying thorough software engineering techniques. The framework can be used as a black-box, relying on the offered *repository*, or as a white-box framework, extending its functionality through its *infrastructure*.

CLAM's repository is made up of a large collection of signal processing algorithms encapsulated as *Processing* classes and a number of data structures included in its *Processing Data* repository. The Processing repository basically includes algorithms for signal analysis, synthesis and transformation. Furthermore it also includes encapsulated platform and system-level tools such as audio and MIDI input/audio both in streaming and file mode. On the other hand the Processing Data repository offers those data types that are needed as inputs or outputs of the processing algorithms. These include classes such as Audio, Spectrum or Fundamental Frequency. It also includes a collection of statistical *Descriptors* that can be obtained from the basic Processing Data objects.

On the other hand CLAM's infrastructure offers ways of extending the already existing repository by deriving new Processing or Processing data classes. In the case of Processing classes this is accomplished by a simple inheritance mechanism in which the user is forced to implement some particular behavior in his/her concrete Processing class. Mechanisms for composing with Processing objects, handling input and output data through *Ports* and control data through *Controls* are also offered. The Processing Data Infrastructure is based on CLAM's Dynamic Types. This is a special C++ class that,

using macros and template metaprogramming techniques, offers a very simple way of creating data containers with a homogeneous interface and automatic services such as introspection or passivation facilities. CLAM's infrastructure is completed by a set of tools for platform abstraction, such as audio and MIDI or multithreading handling mechanisms, a cross-platform toolkit-independent visualization module, XML serialization facilities or application skeletons.

CLAM also offers a number of usage examples and ready-to-use applications. These applications include SMSTools, a graphical environment for audio analysis/synthesis/transformation, and Salto, a spectral-sample based sax and trumpet synthesizer. Another important application is the Network Editor, a graphical tool for creating CLAM Networks using a graphical boxes-and-connections metaphor ala Max. This application can be used as a rapid prototyping and research tool. But CLAM has also been used in many other internal projects for instance for developing a voice processing VST plugin, a high-quality time-stretching algorithm or content-based analysis applications.

CLAM can be compared to some other frameworks presented in chapter 2. After the discussion in section 3.3, and using the different information that we have gathered in it, we will now build a comparison table as a conclusion for this chapter. For doing so we will use some of the features that we most value about CLAM and we think are essential of any framework. A framework that may aim at substituting CLAM should be :

(1) **Comprehensive**. It should offer a complete infrastructure and tools for audio and music analysis, processing and synthesis. Furthermore, it should offer tools for object passivation, visualization or input/output of any kind of audio and music source.

(2) **Cross-platform**. It should be fully portable at least across major operating systems and platforms.

(3) **Free/Open Source**. It should be at least Open Source and preferably Free in the sense defined by the Free Software Foundation (see [Free Software Foundation,]).

(4) **Active**. In order to be successful a software framework must be active and have a community that offers support and feedback.

(5) **Complete software framework**. It should be usable as a white-box / black-box framework but should also offer a visual builder or similar tool.

(6) **Efficient**. Any usable framework for audio processing should have efficiency as one of its fundamental goals.

(7) **Well-implemented**. The framework should be implemented in C++ as this is the best suited language for our purposes. Furthermore, it should use good coding practices and design principles such as Design Patters and the like.

Taking those features into account we in Table 3.1 we summarize how the environments most closely related to CLAM stand the comparison with our framework.

As a final conclusion it is important to mention that although CLAM has already proven useful, it is well designed and presents many interesting features, it is success is not guaranteed. The success of a framework depends on many internal but also external factors. In [Scaletti, 2002], Carla Scaletti outlines the most important factors that ("apart from the lucky accidents") will make a computer music language successful. These are applicable extensible to not only computer music languages but also any kind of music related environment. We will reproduce the complete list as a conclusion for this chapter:

- Reasons intrinsic to the language. A language is successful...
  - if people are using successfully
    - if it answers a need that is not otherwise satisfied
    - if it is able to express the unanticipated
    - if its underlying data structure can support extensions and multiple interpretations without violating the original model
    - if its authors can strike a balance between providing users with what they say they need and what they do not yet know that they need.
- Reasons extrinsic to the language. A language is successful...
  - if one can learn it and learn from it
  - when it has a community of users
  - when it serves as a nexus for interdisciplinary cross-fertilization
  - when its author uses it regularly
  - when the people behind it are committed to its success.
  - when people are ready for it
  - when people say it is successful
  - sometimes due to pure luck
  - if it has contributed ideas and stimulated new developments in the field.
  - A language cannot be successful unless it first exists

	(1)	(2)	(3)	(4)	(5)	(6)	(7)
CSL	Yes	Yes	Yes	Yes	No	Yes	Fair
SndObj	No. Very few tools	Yes	Yes	No. But updated very often	No	No	Fair
STK	No. Only synthesis	Yes	Yes	Yes	No	Yes	Fair
OSW	No. Synthesis-oriented and limited repository	Yes	Yes	No	Yes	Yes	Very Good
Aura	No. Synthesis-oriented and limited repository	Planned	Not yet published	Not yet published	Yes	Yes	Good
Marsyas	No. Analysis-oriented	Yes	Yes	No	No	No	Fair
Maate	No. Analysis-oriented	No	Yes	No	No	No	Fair

Table 3.1: Comparing Frameworks similar to CLAM

CHAPTER 4

The Digital Signal Processing Object-Oriented Metamodel

The main hypothesis of this work is that any signal processing system can be modelled as a set of interrelated objects. In the previous chapter we presented an object-oriented framework for audio and music processing that can be seen as a prove of concept of this hypothesis. The Digital Signal Processing Object-Oriented Metamodel, or DSPOOM for short, is an abstraction of many of the ideas found during the design of the CLAM framework[1]. Nevertheless most of these abstractions can also be found in other frameworks for music and audio signal processing as those reviewed in chapter 2. Because of this we present DSPOOM as a general metamodel valid for any signal processing system.

§4.1 DSPOOM as a Classification of DSP Objects

As presented in section 1.2.4 a metamodel such as DSPOOM aims at abstracting commonalities between a set of related models and presenting an abstract "model of models". Abstract metaclasses are actually acting as concrete class classifiers and as a matter of fact the relation between a metaclass (abstract class in the metamodel) and a concrete class is the same as the relation existing between a concrete class and its instances.

The infrastructure offered by CLAM and reviewed in the previous chapter can be understood as a set of objects and the way they interact. Most of these objects belong to three classes: Processing, ProcessingData and Network. These classes define the way that most CLAM objects contribute to the configuration of a given application, being this application in itself a CLAM model of a particular system. The three classes are abstract and therefore cannot be instantiated by themselves, but they

[1] Remember that, as already commented in different occasions, we believe that "frameworks generate metamodels"

define the model a CLAM system will comply to. In other words, a CLAM model can be described by the way that particular instances of these classes behave and interact in between them.

In a similar way DSPOOM classifies objects into four categories: objects that process (Processings), objects that hold data necessary for the process (Processing Data), objects that connect or interface (such as ports, data nodes or controls), and application or system-level objects.

Of these categories, the first two are much more important in their scope and the other two can be seen as auxiliary but necessary for completing the metamodel. The basic idea is to separate objects that encapsulate a process called Processing objects and objects that undergo a certain process encapsulating a kind of data object and are called Processing Data objects. This clear separation between data and process objects is not exclusive of CLAM but can be found more or less explicitly in different environments reviewed in chapter 2. See for instance the clear separation between *virtual processors* and *virtual data* in the VDSP framework in section 2.2.3. Each of this categories forms an abstract class of the DSPOOM Metamodel. Thus, a particular model, instance of the metamodel cannot instantiate objects of this class directly but instances of the subclasses related to the model under study.

Therefore, and simplifying the four-category metamodel, a DSPOOM based model can be viewed as a set of Processing objects deployed as an interconnected network. Each Processing can retrieve Processing Data tokens and modify them according to some algorithm. Programmers can keep control over the Processing Data flow between Processing or they can delegate this task to one of the many possible automated Flow Control schedulers. When a set of Processing objects is arranged they form a new processing. Thus the new processing can be used as an abstraction of the whole composition. DSPOOM Processings are compositional and scalable. Processing objects can be grouped on compile-time as Processing Composites, or dynamically on run-time as Networks.

DSPOOM is first, and above all, and object-oriented metamodel. Nevertheless, and as a consequence of applying good object-oriented practices, a graphical model of computation very much related to Process Networks and Actor-Orientation has emerged (see section 1.5 for an overview of this technology).

After this first introduction we will first present the main two metaclasses (i.e. Processing and Processing Data) in more detail. We will then present the composition capabilities of the framework. In the next sections we will analyze the object-oriented features of the metamodel and will relate it to existing graphical models of computation.

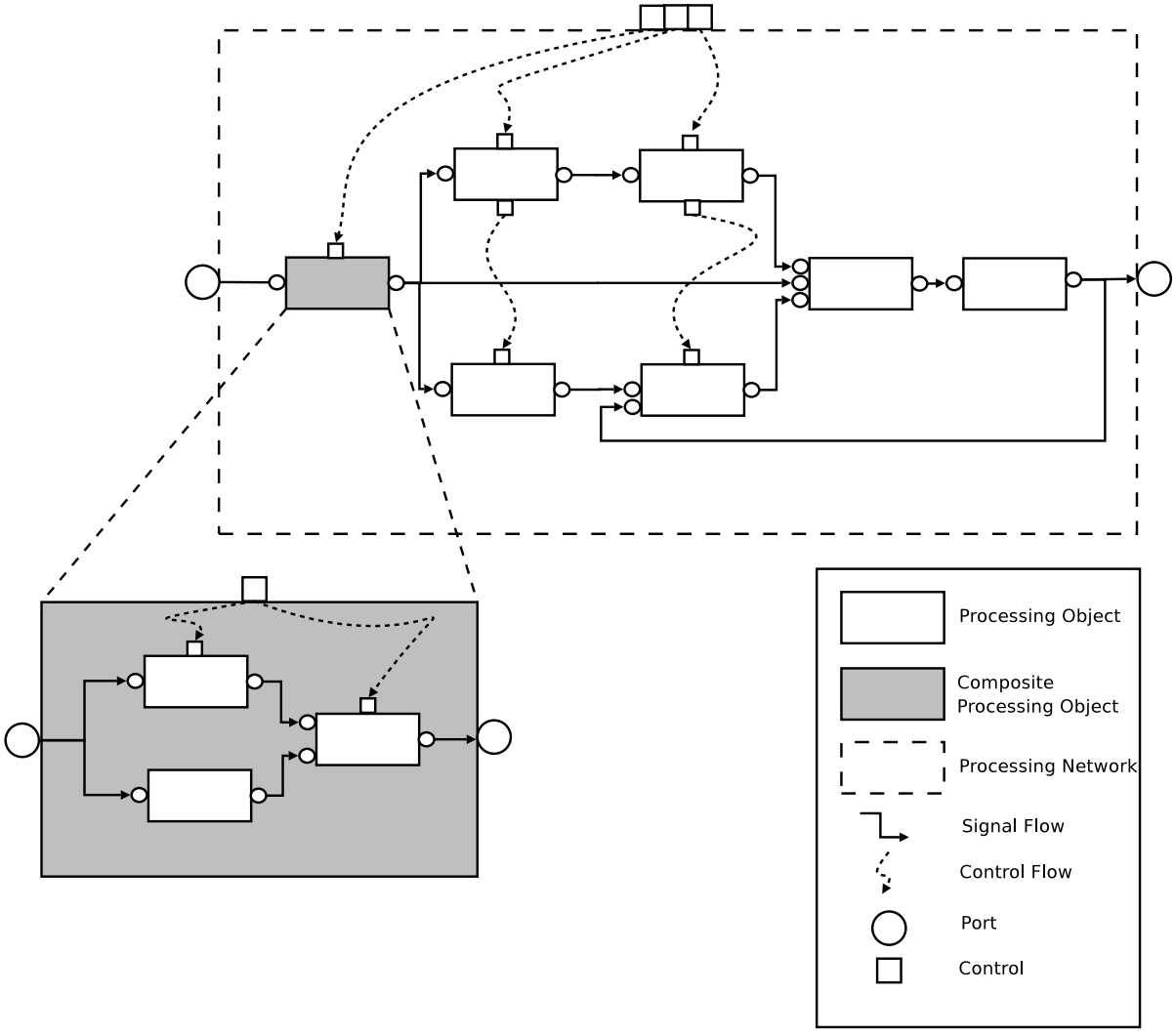

Figure 4.1: Basic elements in DSPOOM

§4.1.1 Processing Objects

We usually understand the verb "to process" as the action of transforming some existing data. This transformation can be a change in the data state or even its essence.

The DSPOOM Processing class is the abstract encapsulation of a process following the object-oriented paradigm. We call any instance of a Processing subclass a Processing object. Just as any object (see section 1.1.1) a Processing object has its own identity, behavior, structure and sequence of valid states. A DSPOOM Processing object is directly related to Processing objects in CLAM (see 3.2.2.1) but also to similar concepts in other environments. Therefore a DSPOOM Processing object can be seen as an abstraction of Marsyas' *transformations* (see 2.3.1.1), OSW's and Kyma's *transforms* (see 2.3.2.2 and 2.5.1), *objects* in Max (see 2.5.2) and *sound objects* in SndObj (see 2.3.3.2), *unit generators* in CSL and Aura (see 2.3.3.1 and 2.3.2.3), STK's *instruments* (see 2.3.2.1), *processes* in FORMES (see 2.3.3.3) or *virtual processors* in VSDP (see 2.2.3).

The Processing objects are the main building blocks of a DSPOOM modeled system. All processing in a DSPOOM model must be performed inside a Processing object. While the Processing object is running it receives and emits two kinds of output: synchronous data and asynchronous controls.

Figure 4.2 illustrates the different concepts that are encapsulated in the Processing abstract class and therefore in any concrete Processing object. Its main components are a *configuration*, incoming and outgoing data *ports*, incoming and outgoing *controls* and any number of internal *algorithms*. We will now introduce all of these concepts and the way they interrelate.

First though it is important to note that in a processing object we define two different flows: from left to right we have the *data flow* and from top to bottom we have the *control flow*. The data flow is synchronous and thus controlled by an external clock. The control flow is asynchronous and event-driven. We will come back to the definition of different flows in section 4.1.1.3, when we are a bit more familiar with the processing internal structure.

When triggering the process, we are asking the Processing object to access some incoming data and, using the encapsulated algorithm(s), transform it to some output data. A Processing object is able to access external data through its connection Ports. Input ports, or Inports for short, access incoming data and Output ports, or Outports, send outgoing data. This same idea is found in other environments apart from CLAM. Ports, for instance are called *inlets* and *outlets* in OSW and Max (see 2.3.2.2 and 2.5.2)

Apart from the obvious interface for accessing encapsulated Processing Data, Ports must offer connection facilities. Pairs of ports can be connected as long as one of them is an Inport and the other

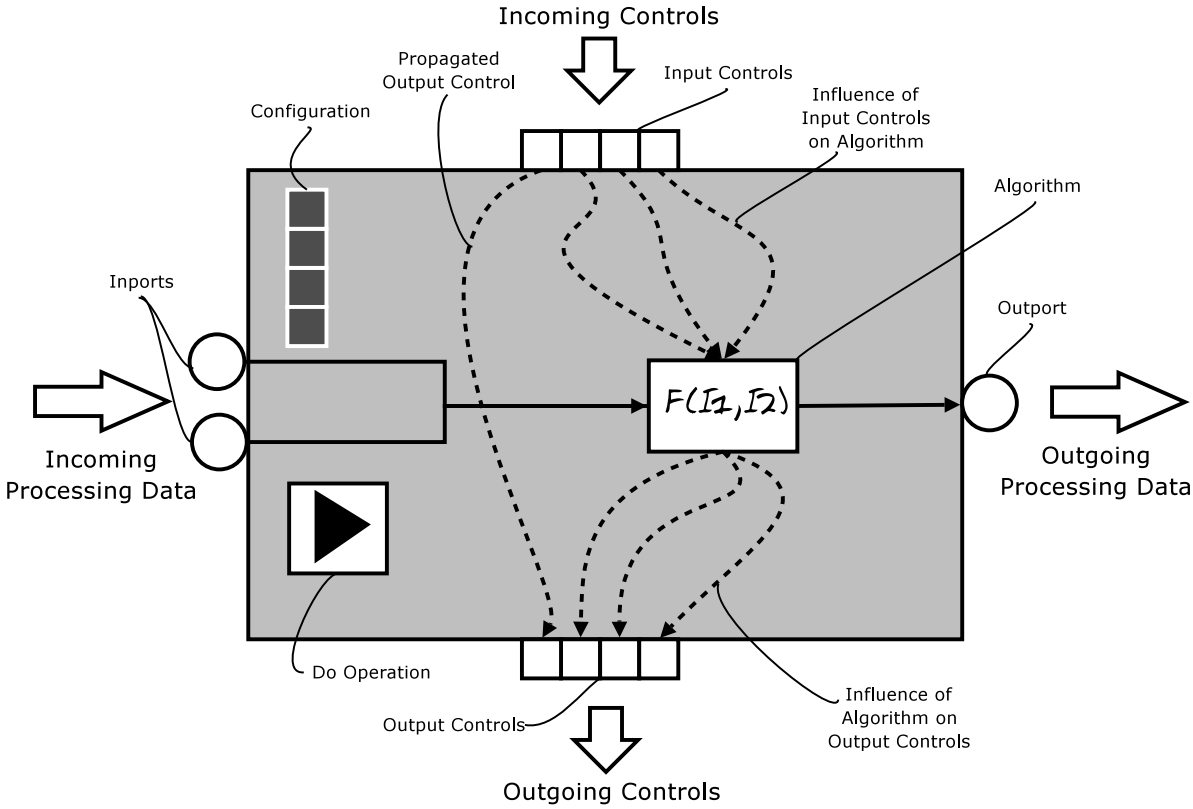

Figure 4.2: DSPOOM Processing class

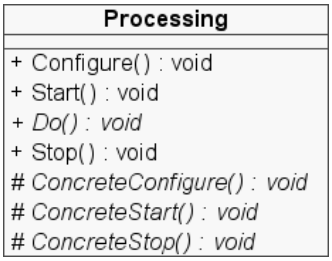

Figure 4.3: Processing Class

and Outport and meet a further condition: they are of the same Processing Data type. Although this condition may be somehow relaxed by allowing connections of polymorphic Processing Data Ports, this is not the recommended nor the usual situation.

Besides receiving and sending data at a fixed rate through its Ports, a Processing object may also receive asynchronous Control events. These events affect the Processing object internal state and therefore are able to influence the result of the process itself. The Processing object can also broadcast these received events or some internally generated ones through its output Controls.

But all this mechanisms can be seen as auxiliary to the processing object main functionality, which is that of encapsulating one or more particular algorithms that work towards a clearly defined purpose. These algorithms are the ones actually in charge of transforming or processing the input data. The selection of one of the maybe alternative algorithms available is usually done upon configuration though some particular processing objects may be able to implement an Strategy pattern[Gamma et al., 1995] for dynamically selecting one algorithm or the other. Whether this policy is possible or not will depend, as in many other situations, on whether a change in the algorithm means a structural change or not.

The execution of the Processing functionality is triggered by sending it a Do message. This should be the only way to access a Processing object functionality and the response to this message must be well defined. On the other hand, dynamic changes (and by dynamic we mean those that can be applied without the object having to transition from one of the main states to the other) will be triggered by the acknowledgement of an input control. The response of a Processing object to a Do message depends on the values of the data in the input ports but also on the internal state, which depends on the input controls received.

§4.1.1.1 The Processing Object Lifecycle

As already stated in section 1.1.1, the state of an object is defined by the value of its attributes at a moment in time. The sequence of states on an object define its lifecycle. Nevertheless, not all variations on attribute values produce an important transition in the lifecycle. That is why we talk

about main states or supra states and sub-states.

A Processing object lifecycle is made up of the following main states: *Unconfigured*, *Ready*, and *Running* (see figure 4.4). While in the *Unconfigured* state the Processing object is waiting to be configured; in the *Ready* state it can be reconfigured or started; in the *Running* state, once the Processing object has been configured and started, the actual process can be executed; finally the Processing object can be stopped in order to start the cycle again.

The messages that can be sent to a Processing object in order to change its state are: *Configure*, *Start*, *Do* and *Stop*. Any of these operations has a generic part and a concrete part that is coupled to the concrete Processing class. The generic part of the operation is in charge of controlling the state transitions and life life cycle invariants. It is reasonable to think that the use of the Template Method design pattern (see [Gamma et al., 1995]) will offer an optimum solution to this situation, implementing the commonalities in the abstract Processing class and delegating any particular issues to the concrete classes. We will now see how all these state transitions affect a general Processing object.

The Processing object can only respond to the Configure message if it is in the Unconfigured or Ready states. In any case if the configuration that is passed is well-formed and valid the Processing object will enter the Ready state, else it will stay or go to the Unconfigured state. In this configuration process different internal operations may take place:

(1) Initialization of the *configuration variables*, member variables that contain parameters that are not supposed to change during the execution phase.

(2) Initialization of internal tables that need to be allocated taking into account some configuration parameter such as the size.

The Start message will only be acknowledged if the object is in the Ready state and will produce a transition to the Running state. In this transition, two activities are to take place:

(1) Asserting that the Configuration process has taken place and left the Processing object in a valid concrete Ready state.

(2) Initialization of *execution variables*. Execution member variables are defined as those that are bound to change during execution and they include items such as internal counters, timers or memories. These execution variables also include input and output *controls*.

A Processing object can only respond to the Do message and therefore process when it is in the Running state. During this phase the Processing object can only receive this message and any other communication must be done through the control mechanism. In this state we can identify two other sub-states:

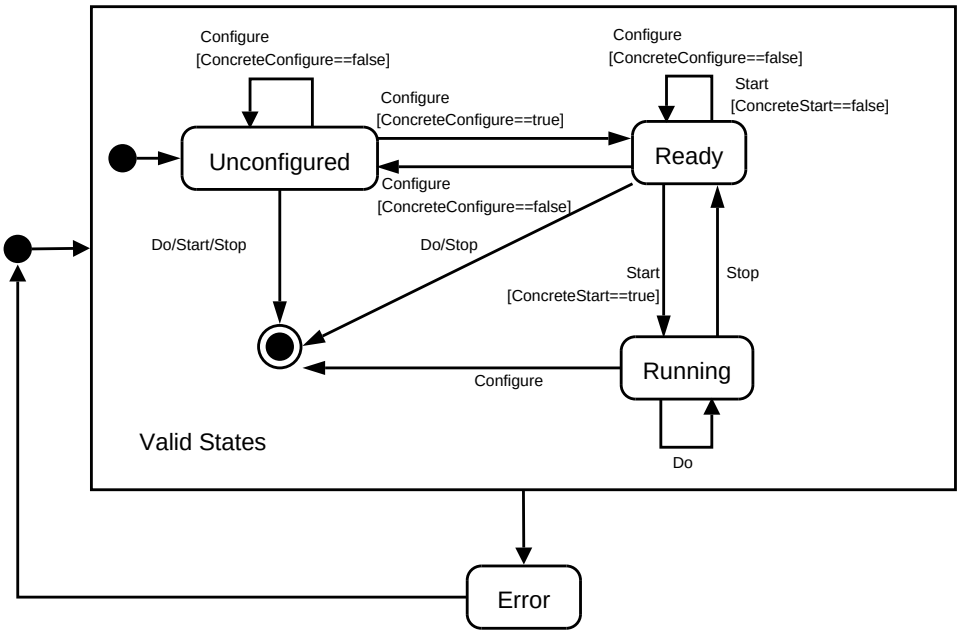

Figure 4.4: DSPOOM Processing state diagram

Idle and *Processing*. In the *Processing* state the object is actually performing the computations as a response to a Do call while in the *Idle* state the object has already handed control to the Flow Control and is waiting to be called again.

Finally, the Stop operation will make a Running processing object enter the Configured state. In this transition, memory can be deallocated and execution variables reset.

It is interesting to note that this lifecycle is a variation/enhancement of the lifecycle of a process in the Simula language. In [Dahl and Nygaard, 1966] it was stated that a process could be in one of four possible states: (1) active, (2) suspended (3) passive and (4) terminated. On the other hand, in section 2.2.1 we commented how Ptolemy, as many other frameworks, distinguishes three execution phases: *set-up*, *run* and *wrap-up*. Note that in our case these three phases are also explicit, with the Configure operation making most of the *set-up* operations, the Start operation finishing up the *set-up* and making the transition to the *run* phase where the Do operation is called, and the Stop operation in charge of the *wrap-up*.

§4.1.1.2 Processing Configuration

A Processing Configuration is an object that contains values for a Processing object's non-execution variables, that is all the variable structural attributes that can only be changed when the Processing object is not in the Running state. The Configuration is used for setting the initial state of the processing object. It may contain attributes related to any structural characteristic like the algorithms or strategy to be selected and values for structural parameters , such as the expected size of the incoming data, that will be used to also initialize the algorithms or internal tables that may be used for convenience. A Configuration though may also have initial values for non-structural attributes such as the controls (see paragraph on "Data and Control Flow").

Processing Configuration classes are in fact very much related to Processing Data classes and need to offer the same mechanisms and services. We will review them in section 4.1.2.

It is important to keep in mind that the configuration mechanism cannot be used to change parameters of a Processing object on execution state, only during an initial configuration stage. If the class needs to offer an interface for changing parameters or values on run-time the Control mechanism must be used.

Configuration related data is known as *parameter arguments* in Ptolemy, in order to make a clear distinction between such data and that related to data and control flow, which is known as *stream arguments* (see 2.2.1).

§4.1.1.3 Data and Control Flow

While a Processing object is in the Running state it will receive, process and produce two different kinds of information:

- Synchronous data: That will be fed from and to the Ports every time a Do method is called. Processing objects consume Processing Data through their input Ports and produce Processing Data to their output Ports.

- Asynchronous data: Fed from and to the Controls whenever a control event happens. They usually change the internal state/mode of the algorithm

These two kinds of data clearly define two different graphs and flow control policies. We will now explain how these kinds of data affect a Processing object and will leave flow control issues to future sections (see section on Networks in 4.1.3).

Processing objects consume Processing Data through their input Ports, process it and leave the result in their output Ports. The Processing Data is consumed and produced in response to a call to the Do operation therefore when receiving such a message the Processing object must have valid data in its inputs and must have a valid location where to write the result.

It is important to note that in DSPOOM a data token is the atomic partition of any processing data that refers to an instant in time. Therefore a whole spectrum, whatever its size, is a data token. But a chunk of audio is not considered a token as its data spreads over time, in this case the data token is each of its sample. Processing objects though, do not consume a unique, not even fixed, amount of data tokens. Each processing object may configure the size of the data chunk needed for a single execution. A processing may need n spectrums or n audio samples in order to produce a valid output. In section 4.1.2 we will review the main features and services that these Processing Data objects must offer in the context of a DSPOOM metamodel.

Ports in a processing may be also understood as pointers that point to somewhere where the data to be processed is located (usually a memory location). If both the outport and one of the inputs are pointing to the same location, the processing object is said to be processing *inplace*. Not all the processing objects have the ability of processing inplace as this greatly depends on the algorithm that they encapsulate. Furthermore, input and output ports do not even have to be the same kind: the transformation or process introduced by the processing object on the incoming data can be so structural that even the data kind may change (e.g. a processing object may convert an input "tree" into and output "piece of furniture").

But, as already mentioned, apart from the synchronous data flow, Processing objects can respond to an asynchronous flow of events. This mechanism is encapsulated in the concept of Controls. A Processing object may have any number of input controls and output Controls. An input Control affects the non-structural aspect of the Processing object state. This means that the reception of an input Control does not produce a transition from one of the four main states to another, it rather produces an inter-state transition from one of the possible sub-states to the other. On the other hand, output Controls maybe generated at any time although it is usual for them to be the result of one of the two following cases: (1) a response to a received input Control or (2) a result of a particular algorithm that apart from (or instead of) producing output data also generates a number of asynchronous events.

Controls must also have a clear initial value. In most cases this value is set in the construction of the owner Processing object. In other cases, though it is interesting to associate this initial value to a particular configuration parameter. The initialization of the control value is performed in the Start transition so that every time that the Stop/Start cycle is followed the control is able to return to its initial value.

This distinction between data and control is directly related to the *In-band Out-of-band Partitions* pattern commented in section 1.5.3.

§4.1.1.4 Kinds of Processing Classes: Generators, Sinks and Transforms

In the Kahn Process Network model of computation (see 1.5.1.2) two kinds of subgraphs are of special importance: *data sources* and *data sinks* [Parks, 1995]. In section 1.5.3 we also saw how the *Data Flow Architecture* pattern classified modules into *sink*, *filters*, and *sources*. In a similar way in DSPOOM we identify *Generating* Processing objects, which are data sources, and *Sink* Processing objects[2]. Figure 4.5 illustrates a CLAM system with such components.

We define a *Generating Processing object* as a Processing object that generates data but does not consume it. This means that a Generating Processing objects has no input ports. So, instead of transforming input data, what it does is to respond to the Do operation by generating data that is related to its internal state. In such objects, input controls play a very important role because they are the responsibles for changing the processing internal state and this way modify the data that is generated. Nevertheless this does not mean that if a generating object does not receive an input Control between two consecutive calls it will generate the exact same data. The object internal state may also respond to internally generated control data. An oscillator, for instance, will have an internal oscillating function or table so it will not generate the same samples in two consecutive calls. This behavior highlights the

[2]Note that the concept of "generating" or "sink" can also be applied to Composite Processing objects and Processing Networks (see 4.1.3).

fact that generating objects can be used for converting an asynchronous data flow (usually coming from an external system or interface) into a synchronous one.

One special case of generating objects are the random generators. These generating objects are the exception to the rule in the sense that the output they generate does not respond to the internal state but rather to a randomized variable.

Sink objects are the opposite to generating objects: they consume data without generating any. Thus, they have input but not output ports. The basic responsibility of a sink object is usually taking internal data from our system and translating it to some other format to transmit it to another system or directly to the user. Examples of sink objects are visualizers, audio players or passivators.

On the other hand, regular DSPOOM Processing classes are just coupled to a single data type that is used both as input and output. Processing objects that have a different processing data type as input are called *Transforms*. Examples of such processing objects are an FFT (input is audio and output is a spectrum) or a Sinusoidal Synthesizer (input is an array of spectral peaks and output is a spectrum).

§4.1.2 Processing Data Objects

All data in a DSPOOM model is contained in *Processing Data* objects, a concept that is related to the *Payloads* pattern explained in section 1.5.3.

DSPOOM Processing objects can only process Processing Data objects. To be fully usable in such a context, a Processing Data class must offer a number of services, namely:

(1) Introspection: A Processing Data object must be able to respond when queried about the number of attributes it holds and their type. It must also be able to know its own type or class name.

(2) Homogeneous interface: All Processing Data classes must have a homogeneous interface so as to be queried transparently without any knowledge of the concrete subclass.

(3) Encapsulation: Attributes must be protected and access to them only given through appropriate setters and getters.

(4) Persistence: A Processing Data must have automatically built-in persistence into an appropriate format.

(5) Display facilities: Any Processing Data must be accompanied by display facilities that allow for debugging and visualizing its content at any time.

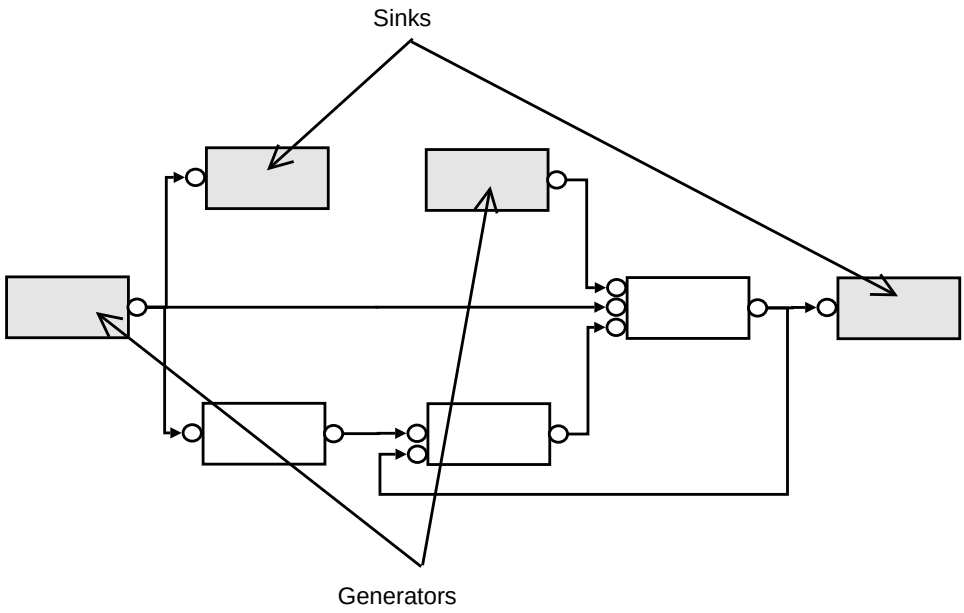

Figure 4.5: Generators and Sinks

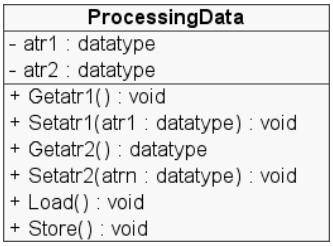

Figure 4.6: Processing Data Class

(6) Composition: An attribute of a Processing Data class may be another processing data.

Note that many of these requirements are related to the services offered by MOF meta-objects (see section 1.4.3).

§4.1.2.1 Data and Value Attributes

Attributes in a Processing Data class can be classified into *data attributes* and *value attributes*. Data attributes basically act as data containers and they are usually of complex types such as arrays.

On the other hand, value attributes act as auxiliary information related to the data attributes. They are always of simple types such as integers or floating point. Value attributes can in turn be divided into *informative value attributes* and *structural value attributes*. Informative value attributes are simple value containers that are used to interpret the Processing Data content. A modification in such attributes does not imply a change in the related data. Conversely structural value attributes are meant to inform but also to modify the internal structure of Processing Data objects. An example of such an attribute is the "size" of a given Processing Data object, its value must be obviously synchronized to the size of the internal buffers and a change in it means a resizing of all the buffers.

But it is not always obvious to identify whether a value attribute is informative or structural and its interpretation is many times a design decision. The same value attribute can be interpreted as informative or structural in different Processing Data classes, depending on what the designer decided. An example of such a dual value attribute could be the "scale" (linear or logarithmic) of a particular data. One may choose to implement it as purely informative in the sense that modifying it would not imply a data conversion. In this situation the responsibility of keeping data and value attributes synchronized is left to the Processing Data class user. But one may also choose to implement it as structural and decide to convert data any time the scale value is modified.

It is generally better to implement ambiguous value attributes as structural therefore removing the synchronization responsibility from the user. But in some cases the cost and logical complexity of adopting this solution may recommend otherwise.

§4.1.2.2 Dynamic and Static Views of Data Flow

Processing objects synchronously receive incoming data through their input ports and output the processed data through their output ports. Though it is many times convenient to keep this metaphor of travelling data that enters the processing, is transformed and is fed to the output, it is also important to sometimes think in terms of its "static" representation. Therefore two different interpretations may be given to the flow of processing data through a DSPOOM network: a dynamic view and a static view. Both of them are valid and their use will depend on the goal of the particular model.

In the dynamic view we think of a processing data as a travelling entity that enters every processing in its path and exits sometimes even in a different form. This interpretation is useful for analyzing dynamic behavior of a given system specially regarding its latency or real-time performance. It is also useful for understanding overall effect of a processing network on the input signal.

When using the static dataflow view we interpret that processing data all "live" in nodes or slots where different processing objects are connected. A processing always reads from the same position in memory and writes to the same position in memory so its input and output ports can be seen as a static (in space) processing data that is constantly changing. This view is interesting in order to study flow related issues and the interaction that occur between neighboring processing objects. We will later see more details about the network structure, including the definition of *data nodes* (see section 4.1.3.2.1), and understand more about this static view.

§4.1.2.3 Processing Data and Controls

At this point it may still not be completely clear how we can decide in a particular model if some data should be modeled as Processing Data or a Control. There are several issues though that can illustrate the existing difference between Processing Data Objects and Controls.

To start with, Controls are simple data types such as integers, floats or boolean. Any structure more complex than that cannot be sent as a Control. This feature may be seen as a limitation of the metamodel but it is in fact a feature. It is important to bear in mind that controls are sent asynchronously whenever they generate or are modified. The control associated to a slider, for instance, can be transmitted several times before actually affecting the result of the process, which will happen next time that the Processing Do is called. It would be a waste of resources to transmit complex structures with this mechanism.

§4.1.2.4 Descriptors

Descriptors are a special kind of Processing Data that are obtained from another Processing

Data or Descriptor through a simple *extraction* process. A descriptor is metadata as it contains data that describes its originating data. Generic descriptors can be obtained from almost any kind of Processing Data, examples of such descriptors are the average or the nth order moment. Nevertheless most useful descriptors carry more semantic information and are in some way coupled to a particular Processing Data, examples of such descriptors are the spectral centroid or the zero crossing rate.

The previous explanation is not enough to clearly distinguish Descriptors from regular Processing Data. Should a spectrum be considered a Descriptor as it can directly "extracted" from a time-domain signal? We know that the spectrum is the result of a *transform* and not an *extraction process* nevertheless in some cases it may be useful to consider and treat it as a Descriptor. Therefore we cannot draw a clear line separating regular Processing Data from Descriptors. Nevertheless two hints that may be used are the following: first a Descriptor should be easily computable using a limited set of simple statistical operations; and second a Descriptor cannot be easily converted back to its originating Processing Data. Although both these rules of thumb clearly discard the spectrum from being a Descriptor, some cases are not easy to decide and a given model dealing with descriptors must establish what it considers as "limited operations" and as "easily converted back". In the general case we can conclude that any Processing Data could be understood as a Descriptor under certain circumstances.

An *Extractor* is a special kind of Processing object that is input a Processing Data or a Descriptor and outputs a Descriptor after having applied the extraction process. This extraction process is the result of applying simple statistic tools to the input signal. Therefore an Extractor must have an accessible repository of basic statistical tools.

Expressions can be built by composing with extractors. For instance, it is usual to compute Descriptors on a frame basis, accumulate these values and then perform a time wise average.

We will come again to different kinds of descriptors and their usage in chapter 5 in the context of the Object-Oriented Content Transmission Metamodel.

§4.1.3 Composing with DSPOOM Objects: Networks and Processing Composites

The DSPOOM metamodel offers different mechanisms for composing with Processing objects. In this section we will show their features and intent. Composition in DSPOOM is not an absolute need as any model can be fully specified by a number of independent though coordinated Processing objects. Nevertheless, building self-contained sub-models (thus abstract subsystems) offers a number of advantages, namely:

(1) Complexity and detail hiding. Submodels can act as intermediate layers that hide complexity from the user. This way we can choose what details will be promoted from one level to another and what formal view we want to offer the user of each layer.

(2) Flow control automation. By building coherent and homogeneous compositions we are able to apply standardized approaches to automatic flow control in which an "intelligent" entity is able to manage data flow and process execution.

(3) Efficiency and optimization. A composition can be a specialized grouping of Processing objects with a specific purpose and goal. By using information about the context in which each individual and possibly generic Processing object is used, we can manage to optimize the overall execution.

The two mechanisms for composing with Processing objects in a DSPOOM model are *Networks* and *Processing Composites*. Networks are dynamic compositions that can be modified at run time in any way by adding new Processing objects or modifying connections. Processing Composites are static compositions that are built on compile time and cannot be modified thereafter. These DSPOOM constructs were already graphically illustrated in figure 4.1.

Note that both mechanisms address the first of the three benefits previously enumerated. But while Networks promote the ability to use automatic flow control (benefit number 2), Processing Composites strive for efficiency (benefit number 3). As R. Dannenberg points out in [Dannenberg, 2004] it may seem at first sight that static composition is not useful in the general case as, although it can be more efficient, it is far less flexible. But experience indicates that static composition is preferred in many occasions due to a number of different advantages: it avoids dynamic patching and therefore becomes more efficient, it enables inlining and other performance tricks, it allows better debugging and finally and most important, users can reason better about their code when it is static.

In general the dynamic composition offered by Networks follows the Dataflow Architecture pattern while the static composition in Processing Composites follows the Adaptive Pipeline pattern (see 1.5.3). Networks are also somehow related to Max's, CSL's and OSW's *patches* (see 2.5.2, 2.3.3.1, and 2.3.2.2) while Processing Composites are similar to Aura's *instruments* (see 2.3.2.3).

§4.1.3.1 Processing Composites

A Processing Composite is a static composition of DSPOOM Processing objects. A Processing Composite is manually implemented by a developer and might be fine-tuned for efficiency reasons. But the list of included Processing objects and their connections may not be modified at run-

time[3]. It is a direct implementation of the Composite pattern as defined in the Gang of Four catalog [Gamma et al., 1995].

A Processing Composite object has one parent that acts as the *composite* and any number of children that act as *components*. Any child can in turn be the parent of other Processing objects recursively defining different levels of composition. A Processing Composite object is seen as a regular Processing object from the outside, even in a Network-like context. Therefore, and as it happens in the classical composite pattern, performing an operation on the Processing Composite means performing it in all its children recursively to the lowest level. Particularly:

- Constructing a Processing Composite means constructing all its children.
- Configuring a Processing Composite means configuring all its children.
- Starting or stopping a Processing Composite means starting or stopping all its children.
- Executing a Processing Composite means executing all its children.

The "construction condition" implies that the most usual way to construct a Processing Composite is by having its children as regular member attributes. These attributes will be constructed in their parent construction, they will be *attached* to the children list, and they will be notified of whom the parent is at that time.

The "configuration condition" means that a Processing Composite has to offer a configuration that at least contains a subset of all the configuration parameters in the children. The process of creating a composite configuration cannot be automated and depends on design decisions. The Processing Composite class developer has to choose what parameters will be promoted from the children to the parent layer of composition. This list of *published* configuration parameters will define the subset of all parameters that will be available for the user. It is important to note that in many situations, children Processing objects share common configuration parameters or have parameters with coupled semantics. For instance, all children Processing objects in a Processing Composite may have a *SamplingRate* attribute that needs to be consistent. Even in that same Processing Composite we may have a spectral domain Processing object that has a *SpectralRange* parameter that, although not equivalent, is directly coupled to the *SamplingRate*. Because of all of this it is not a good idea to simply build the composed configuration by aggregating individual children configurations.

The "start/stop condition" does not imply any complex control issues and can be easily automated.

[3]As a matter of fact we may use the control mechanism in order to change internal connections or bypass particular Processing objects in a Composite. But this is rather a side effect of the flexibility of the metamodel rather than an important inherent feature.

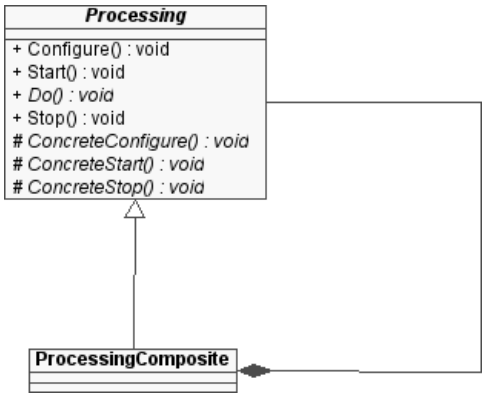

Figure 4.7: Processing Composite

And finally the "execution condition" is where all the flow control issues have to be solved. No automation can be offered at this level and it is the designer's responsibility to choose in what order the children will be executed or how intermediate data will be handled. On the other hand, in reward for this complexity the Processing Composite designer is in the position to optimize and fine-tune all execution and control issues that could not be addressed in a generic environment such as a Network (see below).

§4.1.3.2 Networks

A DSPOOM Network is a set of interconnected Processing objects that collaborate for a common goal and can be modified on run-time. As illustrated in figure 4.8, it can be seen as a set of Processing objects with connected input and output ports and input and output controls.

Nevertheless, if we take a closer look the network entity is made up of the following elements [4]:

- A list of processing objects
- A list of pairs of connected output port / input port.
- A list of reading and writing data types and window sizes (i.e. firing rules) for all ports in the network
- A list of pairs of connected output control / input controls
- A flow control policy that will possibly yield an associated schedule

[4] A particular implementation of such a scheme was already given in the context of the CLAM framework in section 3.2.2.3.

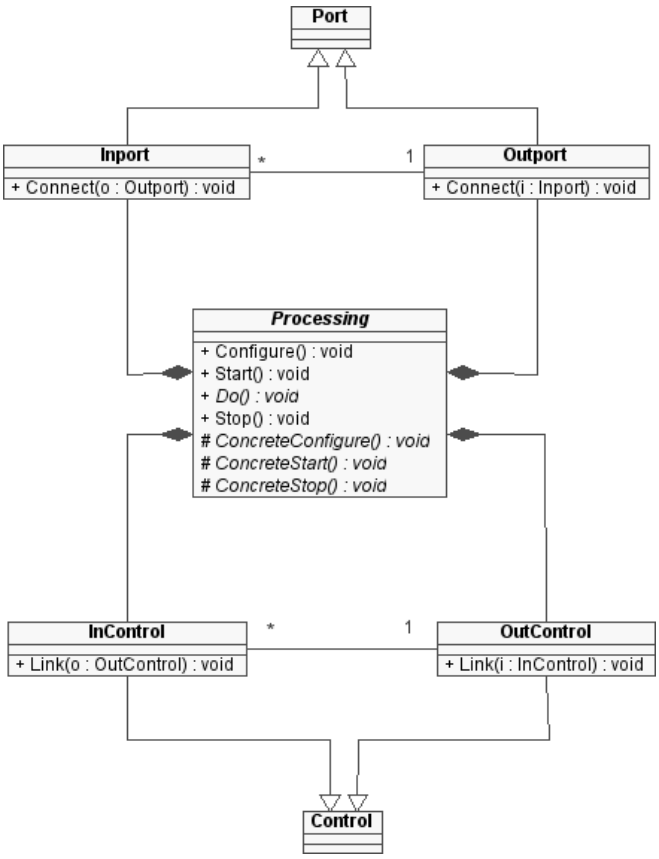

Figure 4.8: DSPOOM Network Class Diagram

A Network has a dynamic list of Processing objects that can be updated at run-time. Therefore, the Network must offer facilities for inserting or deleting new Processing objects. This operation could be done while the network is executing. As a matter of fact the addition or suppression of a Processing object from the network does not necessarily affect the result of the process. The existence of a Processing object is not relevant as long as it is not connected to the process graph.

In order to define this process graph, the network must keep track of the list of connected pairs of input and output Ports. An output Port may have any number of input Ports connected to it while an input port can only be connected to a single output Port.

All interconnected Ports should be strongly typed and expect the same Processing Data type. But the amount of data tokens needed by every Port is not predetermined, does not have to be the same for all ports and does not even need to be fixed on run-time. The amount of data tokens that a given Port needs to consume or produce in every execution is determined by its associated *region* or *window size*. In principle, and in a generic application, there is no forced relation between the region sizes of connected ports. That is, an Outport many have a very small region while the connected Inports have larger reading regions. In this case the producing Processing object will have to be triggered several times before the reading Processing objects can proceed with an execution.

Apart from the main process graph defined by interconnected Ports, a Network has a secondary graph defined by Controls and their connections. As already commented in previous sections, while data flow is handled in a synchronous manner, the control mechanism is event-driven. A control event is transmitted from the output control to the connected input controls as soon as it generates. The new control value overwrites the previously existing one, even if it has still not been read.

But while the Control mechanism is very simple and does not need any supervision, the data flow does need of some sort of external control. The Network itself is in charge of managing writing and reading of data. For doing so the Network must respond to a flow control policy that has been pre-determined and fire the execution of the Processing objects accordingly and by calling their Do operation.

If, for instance, a static schedule is decided, window sizes for all ports must remain fixed during execution. Then the network will determine a static schedule complete cycle consisting of $\mathcal{F}$ (\)firings for each nth Processing object. The execution of a network will stop whenever a stop control is received at the network level (this control may be generated, for instance, by the user or by the absence of data at the network input).

Different dynamic flow control policies can be used depending on the particular problem or system under study. Using a *pull* or *lazy* policy the execution thread starts by the outermost Processing

object in the Network (the one whose output Port corresponds to the output of the Network). If this Processing object checks that it does not have enough input data it hands the control to the Processing object whose output Port is connected to its inputs and so on. In the *push* or *eager* version, the process starts with the Processing object that acts as the input to the chain and generating data until all Processing objects connected to its output Port can process. Then it hands control to them and they repeat the process.

A Network has compositional properties so it can be made of interconnected Networks that in turn have other internal Networks, etc. Therefore, when looking at a Network from the distance, it behaves also like a Processing object and has input and output data as well as controls.

A Network is very much related to Processing Composites. The main difference is that Processing Composites are decided at compile time. Defining a new Processing Composite requires programming a new class and combining existing Processing classes or Processing Composite classes. A Processing Composite is seen at run-time as a regular Processing object and therefore must have all flow control behavior coded into the class. This behavior cannot be parameterized or changed and is usually very much case-dependant. A Processing Composite is less flexible than a corresponding Network but can be more efficient as it is designed knowing the particular characteristics of the components and the overall process being implemented.

On the other hand, a Network is much more flexible, can be decided at run-time without any programming. A Network is more flexible but can sometimes result less efficient than a tuned processing composite. In general terms, we will prefer a Network to a processing composite and will only use Processing Composite after having used the corresponding Network and having decided on optimization that would be necessary in order to yield a more efficient result.

As a Network can be seen as a Processing object when looked from a distance, it can also be classified into the same categories than a Processing object. A Network is a *generating* Network if it has no inputs and a Network is a *sink* Network if it has no outputs.

§4.1.3.2.1 Data Nodes

A *Data Node* is defined from the logical grouping of all arcs (or FIFO queues) in the graph sharing a common source Port (see Figure 4.9). The interesting feature that leads us to this logical grouping is that arcs with a common source may in fact share the same FIFO queue, avoiding many unnecessary data copying and moving at the only price of making flow control issues a bit more complex. But then again, these issues should be automatic and transparent to anyone not wishing to understand

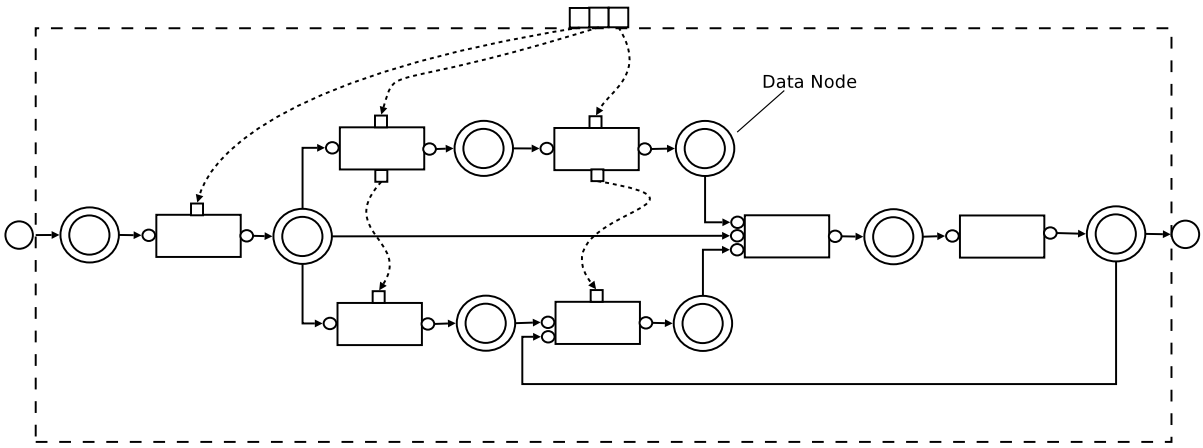

Figure 4.9: DSPOOM Network and Data Nodes

the details of this low-level layer.

A DSPOOM Data Node acts as a connection slot where several Processing objects can be connected. Out of all the connected Processing objects, only one can be a *producer* while the rest will act as *consumers*. In other words, all connected Ports except one must be Inports. As already commented, connected Ports may have different and varying sizes. The amount of data tokens that an Inport needs to process is its only firing rule and these firing rules may be interpreted as sliding windows or *regions*.

But a Data Node is more than a simple connection slot. First it must be interpreted as a data container as it is here where the physical FIFO queues are actually implemented. The implementation of these FIFO queues in the Data Node admits different solutions but the most immediate is based on a circular buffer with several reading and only one writing region.

The Data Node is also in charge of keeping track of the different regions connected to it. It is the Data Node responsibility to avoid inconsistent situations and notify the flow control entity of any exceptional state. Reading regions may be of any size and advance at any rate. They may also overlap. Therefore, the only unwanted conditions that the Data Node has to avoid are:

(1) A reading region reads non-written data.

(2) The writing region advances a reading region and overwrites non-read data

Although the Data Node can be understood as an implementation issue that in most cases should be transparent to the user there may be special cases in which the user would like to access a Data Node contents. Some possible reasons for having to do so are:

- A Processing object has to be directly connected to an external data source and this data has to

be directly fed to the Data Node. This is a common situation when developing plug-ins for APIs such as VST.

- The contents of a Data Node need to be visualized. In most cases it will be sufficient to offer special Visualizer sink Processing objects that are connected to the Data Node as just another regular Processing object. But in other cases it may impossible to avoid accessing the Data Node directly.

- The state of a Network may have to be passivated. The contents of the Data Node are an important part of the Network state. They may have to be passivated when passivating the Network.

We can therefore conclude as for what the Data Node issue concerns that although it is an implementation detail located in a lower layer than the rest of the Network facilities, it needs to offer a clear and accessible interface as sometimes it will be interesting not to completely hide it from the user.

§4.2 Is DSPOOM "truly" Object Oriented?

The Digital Signal Processing Object-Oriented Metamodel has been presented as a completely object-oriented metamodel. The metamodel is an abstraction of the CLAM framework presented in the previous chapter, and for developing CLAM extensive object-oriented techniques and methodologies were applied. Furthermore, DSPOOM applies the object-oriented maxima that states that "everything is and object" (see [Kay, 1993]).

Nevertheless, DSPOOM advocates for a clear separation between Process and Data, which in some sense is similar to advocating for a separation between data and operations. This idea does not fit well into traditional object-oriented thinking.

In this section we will discuss and justify why, regardless of this separation, or even more "because" of this separation between processes and data, DSPOOM is perfectly object-oriented.

§4.2.1 Why almost-degenerated objects are sometimes good objects

The "classical" way to interpret an object-oriented system is as a set of entity objects that hold data inside and respond to messages that can modify the internal data and state. According to some design methodologies (see [Abbot, 1983]) each noun in the domain's terminology must become an object and each verb must become a method on an object. The Expert pattern included in the

GRASP catalogue [Larman, 2002], which is dedicated to illustrate basic and fundamental concepts in object-oriented analysis and design, recommends to assign the operations to the class who owns the data needed by the operation. According to some OO purists objects that only hold data inside or only have methods but no attributes should be considered *degenerated* objects.

Nevertheless, there are many signs that this conception of a class as an extension of an abstract data type is too constrained and not always recommendable. First it is interesting to note that modern object-oriented design has not followed this model. Design patterns and magical objects such as containers, iterators and traits[Alexandrescu, 2001] are good examples of objects that would be considered "degenerated".

It would be fairly simple to implement this classical object-oriented model (a class is a set of attributes plus a collection of methods that can be applied to these data) if we were just to model simple time-domain processes. An `Audio` class might have, for example, a `DoIIRFilter()` method. The `Audio` class would respond to this message by invoking the corresponding Infinite Input Response filtering algorithm. Even in this case the class would grow to the infinite and would not be maintainable or extensible.

Furthermore such a model does not stand true if more complex processes are to be applied, specially if a particular process is meant to change the representation of the data. As an example imagine how you would model an FFT. We could still have an `Audio` class with a `DoFFT()` method. This method, though, would change the representation of the signal in such a way that it would be very difficult to maintain this same representation as data of the same `Audio` class.

In many cases, it is interesting to keep a conceptual distinction between operations and data. This idea has been used in many different situations. In [Halbert and O'Brien, 1987], the authors say that there are times when operations are so complex that it is better they become objects in themselves. The authors list the following reasons for doing so:

(1) The operation is a useful abstraction.

(2) The operation is likely to be shared by different classes.

(3) The operation is complex.

(4) The operation makes little use of the representation of its operands.

(5) Relatively few users of the class will want to use the operation.

Some other reasons that could be added to this list are: parts of the operation are shared between different operations, the operation can be applied to different kinds of data, the operation has memory

or many different operations can be applied to the same class of data.

Our domain is digital signal processing and we need to find a metaphor that is close to the experts understanding of their domain. In digital signal processing, users expect to find a clear distinction between a process and its input and output data.

At this point it is important to stress that our point of view on object-oriented design is very close to that of Nygaard's, already described in 1.2.3. An object-oriented model is always modeling a *system*. And a system, as explained in section 1.2.1, is a set of interacting objects, which are usually representing processes in the system. Our definition of a Processing object is in line with this idea of object-orientation and would be even be admitted by more traditional definitions. Booch [Booch, 1994b], for instance, after citing more restrictive definitions ends up defining an object as "anything with a crisply defined boundary" and he gives the example of a chemical process in a manufacturing plant.

And although the objects in a system are usually representing processes, we believe that we should also model as an object all the data involved in them. As a matter of fact, a model can be only classified as good or bad if it fits its purpose (see section 1.2.2). Other authors have also agreed on this same idea. In [Graham, 1991], Ian Graham describes several methods for deciding what should and should not be an object but he also emphasizes the fact that "purpose is the chief determinant of what is to be a class".

Finally, another hint that promote the idea that DSPOOM is "truly" object-oriented is that it provides many of the advantages of object-orientation such as:

- Encapsulation and Information Hiding
- Inheritance and Polymorphism
- Modularity and Scalability
- Composition
- Re-use

§4.3 DSPOOM as a Graphical Model of Computation

As we have seen DSPOOM offers a completely object-oriented metamodel for digital signal processing. Nevertheless, the consequence of applying object-oriented modeling techniques to the signal domain not surprisingly yields a Graphical Model of Computation.

We will now compare DSPOOM graphical model to those introduced in section 1.5. In order to do so we will first summarize the main properties of its graphical model:

(1) In the DSPOOM graphical MoC the main nodes in the graph correspond to Processing objects.

(2) By connecting Ports from different Processing objects we are defining the arcs in the graphical model.

(3) These arcs in the graph are interpreted as unbounded FIFO queues where data tokens are written and read.

(4) Processing objects produce/consume from the FIFO queues in a synchronous manner, when told to do so by the flow control entity.

(5) The data consumption/production rate of Processing objects is not fixed

(6) This consumption rate may vary at run time.

(7) By connecting controls from different Processing objects we are also defining a secondary set of arcs.

(8) Controls are transmitted using an event-driven mechanism as soon as they are generated.

(9) Controls are not enqueued and they overwrite past values even if they have not been read.

The first four properties in the list simply confirm the fact that DSPOOM's graphical MoC is appropriate for signal processing as those properties are shared by most existing graphical models that have been used in this domain.

The fact that Processing objects can produce/consume different quantities of data tokens leads us to observe that our model can be more easily assimilated to Dataflow Networks (see 1.5.1.3). The "firing rules" of Dataflow Networks are translated into region sizes in the DSPOOM models. The quantity of data tokens to be produced or consumed by a Processing object Port is specified by giving its region a particular size. This is precisely what Dataflow Networks firing rules usually specify.

But common Dataflow Networks such as Synchronous Dataflow Networks have firing rules that are statically defined (see 1.5.1.4). In order to have regions resizeable at run-time like those found in DSPOOM we have to turn to Dynamic Dataflow Networks (see section 1.5.1.6).

Note though, that the DSPOOM metamodel does not require all modeled applications to have dynamically resizeable regions. That is why a DSPOOM compliant model can be also assimilated to a Synchronous Dataflow Network. Therefore, whether a DSPOOM model can be statically scheduled or

not is not a fundamental issue. As for the general case we will ensure that scheduling can be dynamically performed. But in many applications, and under some usually acceptable restrictions, static scheduling can be performed.

As a first conclusion, by looking at the first six properties and taking into account the previous discussion we may conclude that DSPOOM's graphical MoC is a:

"(possibly Dynamic) Dataflow Network"

But the control mechanism in DSPOOM introduces some differences. Its event-driven syntax introduces an extension to the basic Process Network and Dataflow language. This extension though is almost identical to that in Context-aware Process Networks (see section 1.5.1.8). The asynchronous coordination introduced in that model in order to be able to use context information to control KPN execution has the same requirements and features than DSPOOM's control mechanism. The interpretation given to controls in DSPOOM is slightly different as controls are seen as events directly travelling from Processing object to Processing object, without the need of an intermediate register such as the one deployed in Context-aware Process Networks. But then again, these are only implementation details as the basic and fundamental behavior, illustrated by our properties 7 to 9, are fully compatible.

Because of all this, we can say that DSPOOM's graphical model of computation is equivalent to a:

"Context-aware (and possibly Dynamic) Dataflow Network"

Note that until now we have consciously not mentioned the Data Node issue exposed in section 4.1.3. We consider that issue as an implementation detail as far as DSPOOM's graphical MoC is concerned. Anyway, it is interesting to discuss how this affects the basic graphical model.

DSPOOM's Data Node introduces a variant on the basic graphical MoC adding a secondary node type. In this layer, nodes in the graph can be either Processing Nodes or Data Nodes.

Because the benefits and applications of making this issue explicit in DSPOOM's graphical MoC are not sufficiently clear we have chosen not to base the model on its existence. Nevertheless it is interesting to note, as a pointer for future work, that its inclusion makes our graphical model resemble a Petri Net and particularly an *Object-Oriented Petri Net* (see section 1.5.1.1), where Data Nodes are *places* and Processing objects are *transitions*.

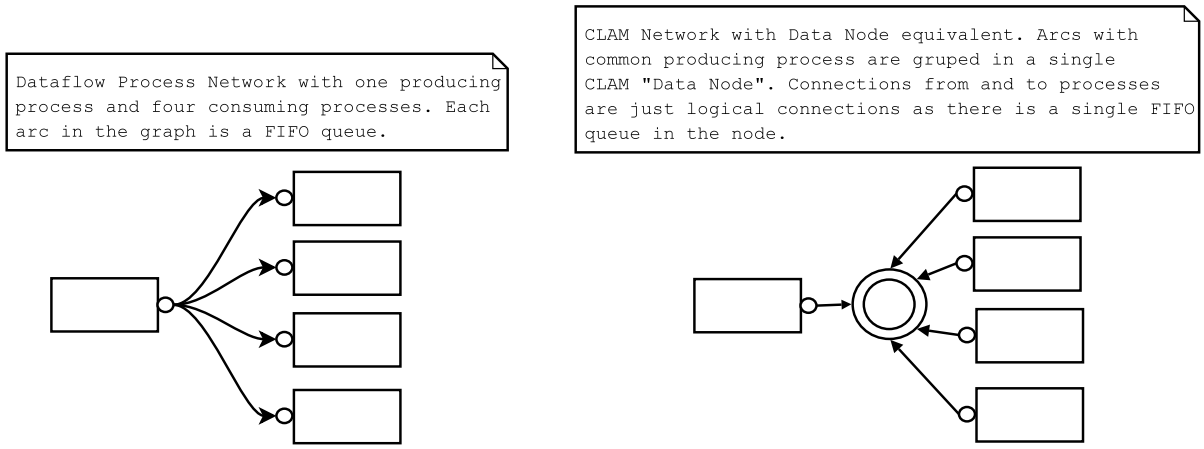

Figure 4.10: DSPOOM Data Node

§4.4 Summary and Conclusions

In this chapter we have presented the Digital Signal Processing Object-Oriented Metamodel (DSPOOM). This metamodel may be considered the main contribution of this thesis and is basically a result of abstracting the conceptual results found in the CLAM framework.

DSPOOM combines the advantages of the object-oriented paradigm with system engineering techniques and particularly with graphical Models of Computation in order to offer a generic metamodel that can be instantiated to model any kind of signal processing related system.

To do so the metamodel presents a classification of signal processing objects into two basic categories: objects that process or *Processing* objects and objects that hold data or *Processing Data* objects. Processing objects represent the object-oriented encapsulation of a process or algorithm. They include support for synchronous data processing and asynchronous event-driven control processing as well as a configuration mechanism and a explicit life cycle. Data input and output to Processing objects is done through *Ports* and control data is handled through the *Control* mechanism. On the other hand Processing Data objects must offer a homogeneous getter/setter interface and support for meta object facilities such as reflection and automatic serialization services.

The metamodel also presents mechanisms for composing statically and dynamically with basic DSPOOM objects. Static compositions are called *Processing Composites* and dynamic compositions are called *Networks*.

Finally the DSPOOM metamodel can also be considered as an object-oriented implementation of a graphical Model of Computation, particularly the *Context-aware Dataflow Networks*.

We may therefore conclude that the metamodel here described presents a comprehensive conceptual framework to model signal processing systems and applications. The metamodel has already had its practical validation with the CLAM framework presented in the previous chapter. Therefore, and although we believe in the metamodel generality, it has only been validated in the audio and music domain.

CHAPTER 5

The Object-Oriented Content Transmission Metamodel

The basic DSPOOM metamodel has proven to be useful for modeling DSP processes and applications. The present chapter is concerned with the applicability of the metamodel to more abstract processes that are related to the DSP domain but also to higher-level semantic domains. It turns out that this new metamodel, instance of the basic DSPOOM, fits well to the idea of Content Transmission. We introduced the idea of Content Transmission in [Amatriain and Herrera, 2001a] and later developed it in [Amatriain and Herrera, 2001b], where the Object-oriented Content Transmission Metamodel (OOCTM) was first presented as such. In the present chapter we will highlight its main features.

As audio and music processing applications tend to increase their level of abstraction and to approach the end-user level it seems clear that one of the focuses is to step up from the signal processing realm and directly address the content level of an audio source. The term content-processing is therefore becoming commonly accepted [Camurri, 1999, Chiariglione, 2000, Karjalainen, 1999]. Content processing is a general term that includes applications such as content analysis, content-based transformations or content-based retrieval.

While manual annotation has been used for many years in different applications, the focus now is on finding automatic content extraction and content processing tools. An increasing number of projects focus on the extraction of meaningful features from an audio signal. Meanwhile, standards like MPEG7 [Martínez, 2002, Manjunath et al., 2002] are trying to find a convenient way of describing audiovisual content. Nevertheless, content description is usually thought of as an additional information stream attached to the actual content and the only envisioned scenario is that of a search and retrieval framework.

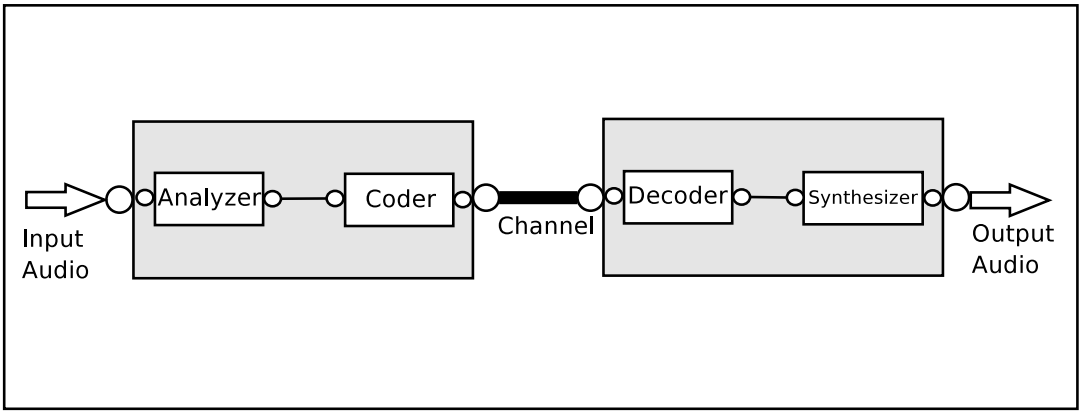

Figure 5.1: The Object Oriented Content Transmission Metamodel

The basic idea when implementing a content processing scheme is to have a previous analysis step in which the content of the signal is identified and described. Then this description can be classified, transmitted, or transformed. Content description is usually thought of as an additional stream of information to be attached to the actual content. However, if we are able to find a thorough and reliable description we can think of forgetting about the signal and concentrate on processing only its description. And, as it will later be discussed, the goal of finding an appropriate content description is very much related to the task of identifying and describing the so-called *Sound Objects*.

Bearing these previous ideas in mind, a metamodel of content transmission (see Figure 5.1) is proposed as a general framework for content-based applications. As we will later see any content-based application can be modeled as a subset of this metamodel.

The metamodel is based on an analysis-synthesis process. Therefore, the only data involved in the transmission step is the content description taking the form of metadata. A multilevel content description tree is used as an efficient representation of the identified Sound Object hierarchy. Several technologies are available for representing content description, but, taking into account our experience in MPEG-7s standardization process [Petters et al., 1999], we would in a general situation recommend an XML-based metadata language such as MPEG-7s DDL.

A property derived from our OOCTM is that if there is a suitable content description, the actual content itself may no longer be needed and we can concentrate on transmitting only its description. Thus, the receiver should be able to interpret the information that, in the form of metadata, is available at its inputs, and synthesize new content relying only on this description. It is possibly in the music field where this last step has been further developed, and that fact allows us to think of such a transmission scheme being available on the near future.

The OOCTM does not worry about encoding fidelity or signal distortion in the classical sense.

A signal has been correctly encoded and transmitted if its meaning has not changed substantially. And what "substantially" means depends on the particular application: in some applications we may need to keep signal-level fidelity while in others we may just be interested in the approximate content in abstract terms.

Audio representations can be classified according to the following properties [Vercoe et al., 1998]: encodability, synthesizability, generecity, meaningfulness, accuracy, efficiency and compactness. A representation is said to be *encodable* if it can be directly derived from the waveform. On the other hand it is said to be *synthesizable* if an "appropriate" sound can be obtained from the representation. A description that is both encodable and synthesizable is said to be *invertible*[1].

The more *general* a representation is the more kind of sounds it will be applicable to. Sound representations that are highly semantic (or *meaningful*) use parameters with clear high-level meaning and are easier to manipulate. *Accuracy* is a measure of how perceptually similar the synthesized sound is to the original one while *efficiency* is a measure of how far redundancy can be exploited in a given representation. Finally, *compactness* is just the ratio between accuracy and redundancy and it is a measure of how redundancy can be exploited while maintaining a certain level of accuracy.

Throughout this chapter we will use the word "descriptor" very often. Although this concept has already been brought up in different parts of the Thesis it is interesting to reproduce again its definition. We will use the one given in the MPEG-7 standard where a *descriptor* is defined as "a representation of a *feature* that defines its syntax and semantics". And a feature is defined as a "a distinctive characteristic of the data which signifies something to somebody".

The OOCTM is related to some existing models and metamodels. It is interesting to note that such a transmission model can be seen as a step beyond Shannon and Weavers traditional communication model [Shannon and Weaver, 1949]. We will further discuss this issue in section 5.3.1. The metamodel is also quite related to Structured Audio, a metamodel for audio and music transmission included in the MPEG-4 standard [Scheirer, 1999c]. We will comment and exploit this relation in section 5.3.2.

In the next sections, we will particularize this metamodel to the case of audio and music content transmission and will give some details on each of the components functionality.

[1] Non-invertible descriptions result in special applications that only instantiate a subset of the complete OOCTM metamodel. On the one hand encodable but not synthesizable representations fall in the category of *signal analysis and understanding*. On the other hand, synthesizable but not encodable formats are simply called *synthesis languages*.

§5.1 Motivation

First, we will take a look at two issues that have already been mentioned and are fundamental for understanding the Object-Oriented Content Transmission Metamodel: what is *Content* and what is a *Sound Object*.

§5.1.1 What is Content?

The term content processing has already been around for a few years [Karjalainen, 1999, Chiariglione, 2000, Camurri, 1999] but its meaning is still unclear and a matter of controversy. When we talk about content analysis, content browsing, content indexing, content processing or content transformation we are usually addressing the higher-level information that a signal produced by an audiovisual source carries within.

Even though the previous pseudo-definition is conservative in its scope, it already includes a crucial and sometimes polemical term: higher-level. It is true that this label assumes that it is being compared to something else, and this something else is usually the signal processing level. Even so, what about semantic features that can (more or less directly) be extracted from the actual signal? Is color a low-level descriptor or a high-level semantic feature of a visual object? Should we consider pitch as a higher-level feature as opposite to its signal-processing counterpart, fundamental frequency? How can we distinguish between the abstraction level implied by some perceptual feature like loudness and some other with more semantic load such as genre?

It is our opinion that for deciding whether something is content or not we must address its functional value. Therefore we will use the word content for any piece of information related to the audio source that is in any way meaningful (that it carries semantic information) to the targeted user. Thus, the description of that content can be thought of as a content hierarchy with different levels of abstraction, any of them potentially useful for some users. In that sense, think of how different would a content description of a song be if the targeted user was a naive listener or an expert musicologist. Even a low-level descriptor such as the spectral envelope of a signal can be thought of as a particular level of content description targeted for the signal processing engineer. We subscribe the already commented definition of a descriptor in MPEG7 as "a distinctive characteristic of the data which signifies something to somebody".

In the next sections we will also see how our idea of content fits well into an object-oriented metamodel.

§5.1.2 On Sound Objects

As already mentioned in the introduction to this chapter, the main goal of the content transmission metamodel is to analyze the signal, identify sound objects and describe them in an appropriate way. But, before getting any deeper into the different modules that make up the metamodel, it is necessary to have a clear idea of what we mean when talking about Sound Objects.

Maybe the most commonly accepted definition of a Sound Object is that related to Pierre Schaeffer's theories [Schaeffer, 1966]. In [Chion, 1983], a Sound Object is defined as "any sound phenomenon or event perceived as a coherent whole (...) regardless its source or meaning". Although this definition might be useful from a psycho-acoustical or perceptual point of view, it is not so from an implementation or engineering point of view.

Other explanations of an object from a multimedia point of view result in definitions with a narrower scope (see [Tolonen, 2000] , as an example of the use of objects from a physical models perspective). In MPEG-7s Multimedia Description Scheme an object is defined as follows: "A perceivable object is an entity that exists, i.e. has temporal and spatial extent, in a narrative world (e.g. Tom's piano). An abstract object is the result of applying abstraction to a perceivable object (e.g. any piano)".

In this section we intend to give a clear definition of what is meant when talking about this idea. For doing so, we rely on definitions given to similar concepts in other areas. Especially we refer to the Object-Oriented paradigm as described in section 1.1. It is interesting to note, though, the strong relation there has traditionally been between OO technologies and computer music or sound signal processing [Pope, 1991a]. As a matter of fact, the definition we will later introduce can be seen as a superset and conceptual enhancement of other previously introduced concepts (see [Scaletti and Hebel, 1991] , for example).

As already commented in section 1.2.3 Alan Kay includes in his definition of OO the maxima that "everything is an object"[Kay, 1993] . Following this same idea, when dealing with Object Oriented Sound Processing, everything must be thought of as an object: a sound stream is an object, a track is an object, a musical note is an object, and an instrument is an object. These objects have different properties and relate between them in different ways.

Let us see a basic example. In a sound stream we have a number of tracks one of which contains a trumpet performance. In this track, there may be different and identical notes (same pitch, same loudness, same attack type). Thus, at first sight, we might distinguish four different kinds of objects:

(1) The whole sound stream

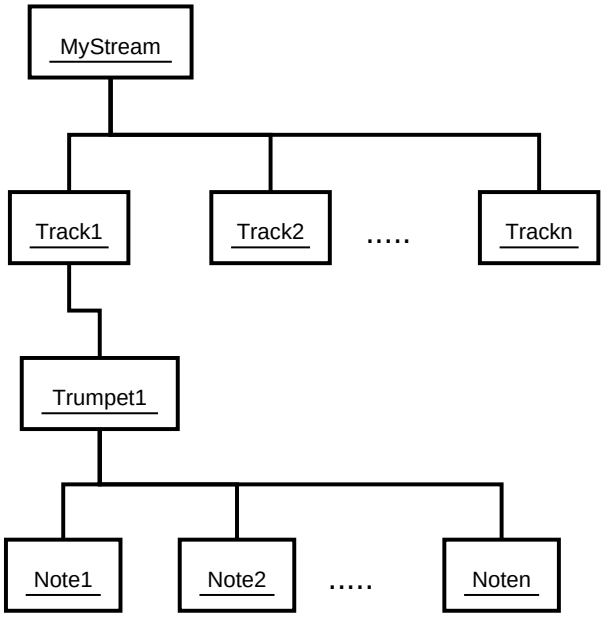

Figure 5.2: UML object diagram of a simple audio stream

(2) Our set of tracks (out of which we concentrate on the one with the trumpet performance)

(3) The instrument in that track (trumpet)

(4) Any number of notes in the track

As a first and basic interpretation, Figure 5.2 illustrates a possible UML object diagram of the system.

On the other hand, in section 1.1.1 we defined a class as an abstract representation of a set of objects that comply with an identical behavior. Following the previous example, we could define what the class Sound_Stream, Audio_Track, Instrument and so on should behave like. The UML class diagram of the previous example would become the one depicted in Figure 5.3, which should be read: "a Sound Stream is made up of any number of Tracks (a Track can only belong to a single Stream); an Audio Track is related to a single Instrument and an Instrument can be recorded into different Tracks; an Audio Track is also made up of any number of notes which have an association relation with the instrument that produced them; Trumpet is a particular case of an Instrument (behaves like an Instrument but may add specific behavior) and Mono Audio Track and Stereo Audio Track are particular cases of Audio Tracks in which a Stereo Audio Track must contain two Mono Audio Tracks and a Mono Audio Track can be contained in at most one Stereo Audio Track."[2]

When declaring a class, we must ask ourselves what should be its *behavior* declaring methods

[2]Note that in no way this example is trying to build a generic model of a sound stream and it is just being used to illustrate the methodology. As a matter of fact, as already highlighted in section 1.2, a single concept or system may be modeled differently according to the purpose of the model itself.

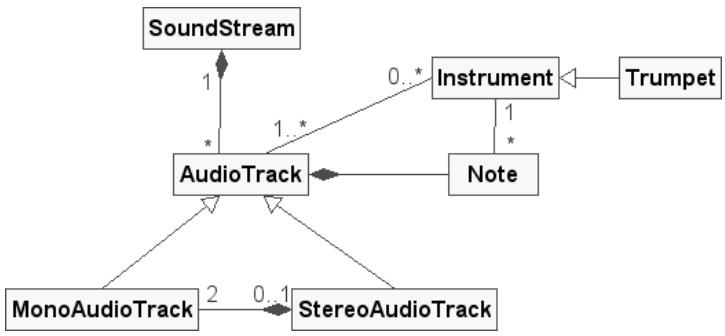

Figure 5.3: UML simplified class diagram representing an audio stream

for that purpose. The class `SoundStream`, for example, might have methods such as `AddAudioTrack()`, `FindInstrument()`. On the other hand we should also identify the attributes that will be used to distinguish the state of two objects belonging to a same class. In that sense, for example, we should identify the attributes that may allow us to distinguish two different instruments (trumpet and piano). We may end up having a diagram similar to the one depicted in Figure 5.4.

The previous diagram, though, does not explicitly show our first hypothesis of everything being an "Sound Object". That is, all the elements are objects but there is no explicit relation in between them to show that they share some commonalities . For doing so, the only missing link we should add is the fact that every class in our model should be a subclass of the SoundObject superclass. The diagram would then become (previously introduced methods and attributes are not shown for simplicity) the one illustrated in Figure 5.5.

Note that following this model we will treat every bit of audio content as an object on its own. For that reason, as we will later see, "content description" is very much related to "object identification".

It is also important to note that Sound Objects in the Object-Oriented Content Transmission Metamodel are in fact Processing Data objects instances of the DSPOOM metamodel (see section 4.1.2).

§5.2 General Building Blocks

The general block diagram of the Object-Oriented Content Transmission Metamodel has already been introduced and is illustrated in figure 5.1. In this section we will zoom into the different components and explain their structure and relation with the required processes. As a first general introduction to the different blocks in figure 5.6 we present the same block diagram but now adding contextual information for each of the components. Note also that the diagram now also reflects the possibility that the Decoder and the Synthesizer communicate either through Data or through Controls.

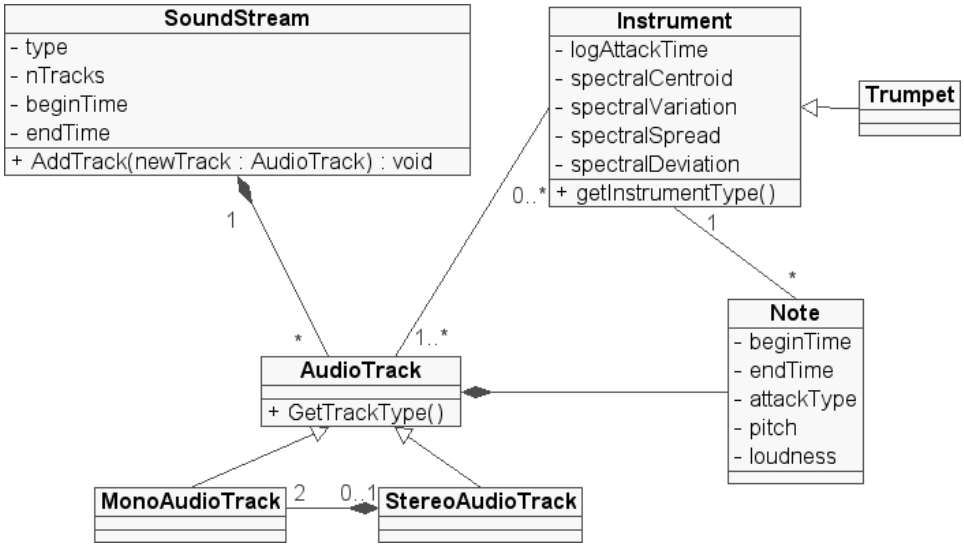

Figure 5.4: UML class diagram representing an audio stream

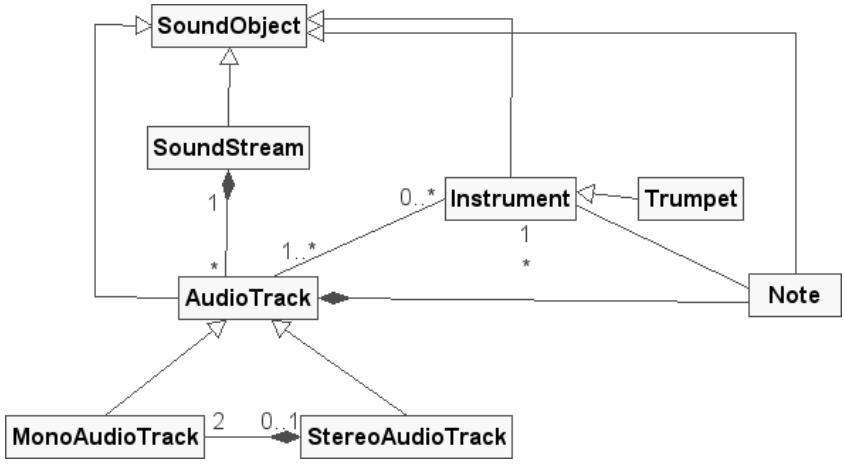

Figure 5.5: The "everything is a sound object" UML class diagram

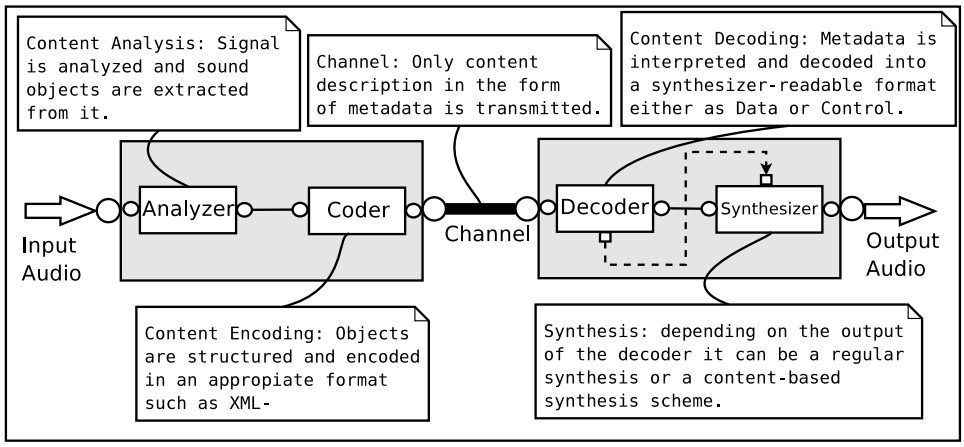

Figure 5.6: Explaining the OOCTM Block Diagram

It is interesting to point out that although all the components have relations and dependencies, an instance of the OOCTM does not need to implement or use all of them. In some cases, for instance, some blocks may be substituted by human intervention. We will give some examples of such schemes in sections 5.3.3 and 5.3.4 and some specific applications in section 5.4.

§5.2.1 The Semantic Transmitter

The transmitter is in charge of taking an input signal and transmitting its content-related metadata. It is what we call the *encoding process*, which can be decomposed into two important subprocesses: analysis and coding.

§5.2.1.1 The Analysis Step (Content Identification)

The Analysis step has two major goals: extract content from the input signal and identify objects.

The easiest way to add content description to an audiovisual chunk of information is by means of textual or oral annotation. The extraction process is in that case performed by an expert that can interpret the content, extract some useful information and classify each sound object, provided there is an appropriate taxonomy available.

When thinking in terms of automatic content-extraction [Scheirer, 2000], two levels of descriptors are usually distinguished: low-level and high-level content descriptors. As a first approach, and in a broad sense, low-level descriptors are those related to the signal itself and have little or no meaning to the end-user. In other words, and thinking in terms of the audio domain, these descriptors cannot be heard. On the other hand, high-level descriptors are meaningful and might be related to semantic or syntactic features of the sound. They will be used to classify sound objects into the class they belong.

The borderline between these categories is thin and not always clear. As previously mentioned the question of "what is" and "what is not" meaningful is not an objective property of a descriptor but rather a property of the whole process. Therefore some descriptors can be viewed as either low or high-level depending on the characteristics of the extraction process and the targeted use. When using these classification we might better think in terms of a multilevel analysis scheme as the one depicted in figure 5.7.

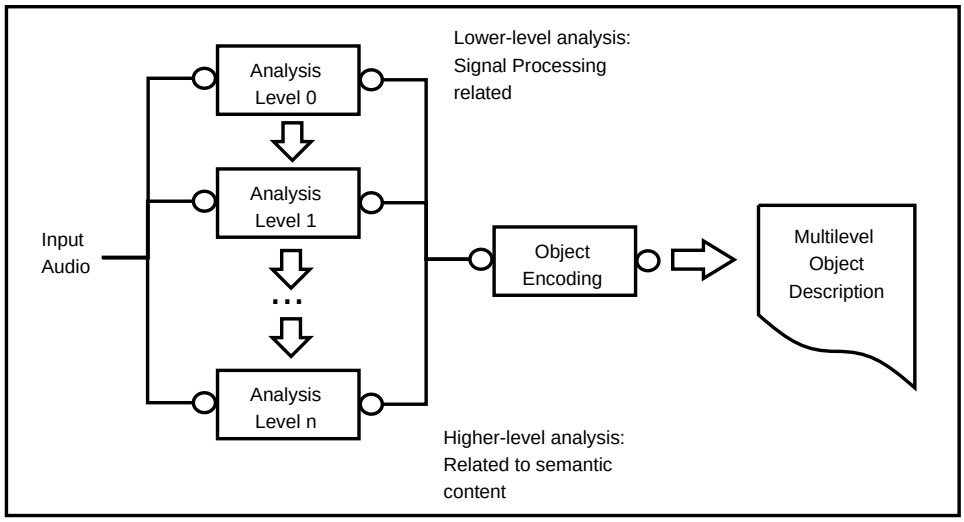

Figure 5.7: Multilevel analysis

§5.2.1.1.1 Low-level Descriptors

As mentioned before, low level descriptors are closely related to the signal itself. We can distinguish at least three levels of extraction granularity from the signal: at any point of the signal, in small arbitrary regions (i.e. frames) and in longer pre-segmented regions.

The set of features that can be extracted at any point in the signal are called instantaneous descriptors. In the case of a time domain representation, most of the useful instantaneous values that can be computed are related to the amplitude or energy of the signal.

If we are dealing with a frequency-domain representation many spectrum-related instantaneous features, such as the spectral centroid or the spectral tilt, can be computed on a given point. To be more precise, one should consider these descriptors as nearly instantaneous as they are not associated to a point in time of the signal but rather to a small region or *frame*.

An important step towards a musically useful parameterization is the segmentation of a sound into regions that are homogeneous in terms of a set of sound attributes. The goal is to identify regions that, using the signal properties, can then be classified in terms of their content. This way we can identify and extract region attributes that will give higher-level control over the sound.

A useful segmentation process applied to a monophonic source divides a melody into notes and silences and then each note into an attack, a steady state and a release regions. Attack and release regions are identified by the way the instantaneous attributes change in time and the steady state regions are detected by the stability of these same attributes. Global attributes that can characterize attacks and releases refer to the average variation of each of the instantaneous attributes, such as average fundamental frequency variation, average amplitude variation, or average spectral shape change. In the steady state regions, it is meaningful to extract the average of each of the instantaneous attributes and measure other global attributes such as time-varying rate and depth of vibrato [Gómez et al., 2003a].

Once a given sound has been segmented into regions we can study and extract the attributes that describe each one. Most of the interesting attributes are simply the mean and variance of each of the frame attributes for the whole region. For example, we can compute the mean and variance for the amplitude of sinusoidal and residual components, the fundamental frequency, the spectral shape of sinusoidal and residual components, or the spectral tilt.

Region attributes can be extracted from the frame attributes in the same way that the frame attributes are extracted from the frame data. The result of the extraction of the frame and region attributes is a hierarchical multi-level data structure where each level represents a different sound abstraction.

From several sound representations it is possible to extract the type of attributes mentioned above. The critical issue is how to extract them in order to minimize interferences, thus obtaining, as much as possible, meaningful high-level attributes free of correlations.

Up until this point, we have extracted and computed features that are directly related to the signal-domain characteristics of the sound and, although they may have a perceptual meaning, they are not taking into account the importance of the listeners perceptual filter.

In that sense, for example, it is well known that the amplitude of the sound is not directly related to the sensation of loudness produced on the listener, not even in a logarithmic scale (see [Moore et al., 1997]). [Fletcher and Munson, 1933] established a set of equal-loudness curves called isophones. The main characteristic of these curves is that the relation between a logarithmic physical measure and its psychoacoustical counterpart is a frequency dependant function. Although this curves have proven only valid for the stable part of pure sinus (more than 500 ms), they have been used as a quite robust approximation for measuring loudness of complex mixtures [Pfeiffer, 1999].

Any audio signal can be represented as a time-domain signal or as its spectral transform, and following this same idea we can separate low-level descriptors into two categories: temporal and spectral descriptors.

Temporal descriptors can be immediately computed from the actual signal or may require a previous adaptation stage in order to extract the amplitude or energy envelope of the signal, thus only taking into account the overall behavior of the signal and not its short-time variations. Examples of temporal descriptors are attack time, temporal centroid, zero-crossing rate, etc...

Many other useful descriptors can be extracted from the spectrum of an audio signal. These descriptors can be mapped to higher-level attributes. As a matter of fact, of the basic dimensions of a sound, two of them (pitch and brightness) are more easily mapped to frequency domain descriptors and a third one (timbre) is also very closely related to the spectral characteristics of a sound. A previous analysis step needs to be accomplished in order to extract the main spectral features. Descriptors directly derived from the spectrum are, for example: spectral envelope, power spectrum, spectral amplitude, spectral centroid, spectral tilt, spectral irregularity, spectral shape, spectral spread; derived from the spectral peaks: number of peaks, peak frequencies, peak magnitudes, peak phases, sinusoidality; derived from a fundamental detection: fundamental frequency, harmonic deviation; etc...

§5.2.1.1.2 High-level Descriptors

While descriptors presented until now are purely morphologic (i.e. they do not carry any

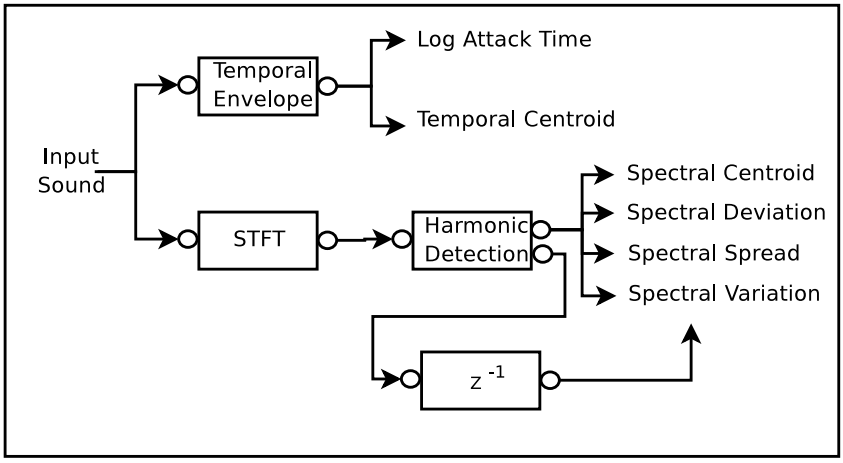

Figure 5.8: Combining low-level descriptors for creating higher-level descriptors: MPEG-7's Timbre Descriptor Scheme

information on the actual meaning of the source and just refer to its inner structural elements), high-level descriptors can carry either semantic or syntactic meaning.

Syntactic high-level descriptors can be sometimes computed as a combination of low-level descriptors. They refer to features that can be understood by an end-user without previous signal processing knowledge but do not carry semantic meaning. In other words, syntactic descriptors cannot be used to label a piece of sound according to what actually is but rather to describe how it is distributed or what is made of (i.e. its structure). Thus, syntactic descriptors can be seen as attributes of our sound classes but, by themselves, cannot be used to identify objects and classify them. For that reason, the computation of syntactic descriptors (either low or high-leveled) is not dependent on any kind of musical knowledge, symbolic or real-world knowledge. In [Petters et al., 1999], for example, we presented a way of describing timbre of isolated monophonic instrument notes (the scheme for computing the descriptors of a harmonic timbre is depicted in Figure 5.8). In the case of our timbre descriptor, for example, the resulting descriptor is not sufficient to label a note as being violin or piano but rather to compute relative perceptual distances between different instrument samples.

On the other hand, semantic descriptors refer to meaningful features of the sound and are "understandable" for the end-user. We therefore need to apply more high-level or real world knowledge. The degree of abstraction of a semantic descriptor though has a wide range, labels such as "scary" or more concrete such as "violin sound" can be considered semantic descriptors.

The main purpose of a semantic descriptor is to label the piece of sound to which it refers using a commonly accepted concept or term that corresponds to a given sound class (e.g. instrument, string instrument, violin...). Descriptors used as the result of a classification process are called *classifying*

descriptors. It is interesting to note that, in this case, the classification process is performed in a top-down manner. Using low-level or high-level syntactic descriptors we might be more or less immediately be able to identify our piece of sound in as belonging to an abstract class (in the worst case we are always able to classify it as a Sound Object). Applying both real-world knowledge and signal processing knowledge we may be able to get our problem to a more concrete ground and start down-casting our description to something like "string instrument" or "violin". Note that for this classification process different techniques may be used being the most obvious a basic decision tree.

Other semantic descriptors, though, do not aim at classifying the sound but rather at describing some important feature or attribute. These descriptors can label a sound as being "loud", "bright" or "scary". As a matter of fact these descriptors are not binary and can indeed quantify how much a sound belongs to a given category (i.e. How bright or scary a sound is). We call such descriptors *quantifying descriptors*. In this case, conversely to what happened with the previous classifiers, the more concrete a feature is the easier it will be to derive it from our previously computed low-level or high-level syntactic descriptors. For example, a label like "bright" might be directly derived from the spectral centroid low-level descriptor. Much more real-world knowledge must be applied to be able to classify a sound as "sad" or "frightening".

Different proposals have been made in order to create a semantic map or multi-level structure for describing an audio scene, one of them being the ten-level map presented in the MPEG Geneva meeting (May, 2000) [Jaimes et al., 2000]. This proposal includes four syntactic levels and six semantic levels: Type/Technique, Global Distribution, Local Structure, Global Composition, Generic Objects, Generic Scene, Specific Objects, Specific Scene, Abstract Objects, and Abstract Scene[3]. Another example of multilevel hierarchies in the context of MPEG-7 is Michael Casey's multilevel content hierarchy for sound effects (see [Casey, 2001]).

While that proposal is quite theoretical and simple, and comes from a generalization of a similar structure proposed for video description, other proposals come from years of studies on the specific characteristics of an audio scene and have even had practical applications. One of the most renowned techniques that can fit into this category is CASA (Computer Auditory Scene Analysis) [Bregman, 1990]. It is far beyond the scope of this section to go deep into any of these proposals, but it is interesting to note that CASA has addressed the issue of describing complex sound mixtures that include music, speech and sound effects, also providing techniques for separating these different kinds of streams into sound objects (see [Nakatani and Okuno, 1998], for example).

It is now important to relate the different kinds of descriptors to the generic DSPOOM classes

[3]Note that the word "object" is here used in the sense defined in MPEG-7 and already commented in the introduction to this chapter.

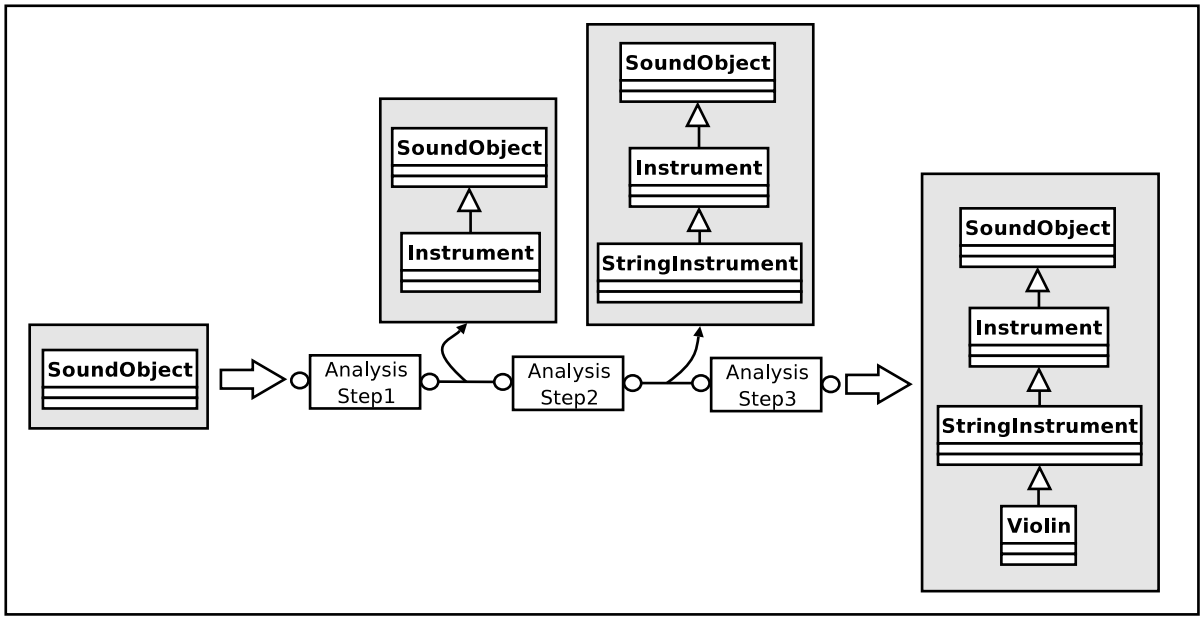

Figure 5.9: Multilevel semantic analysis/classification and polymorphic objects

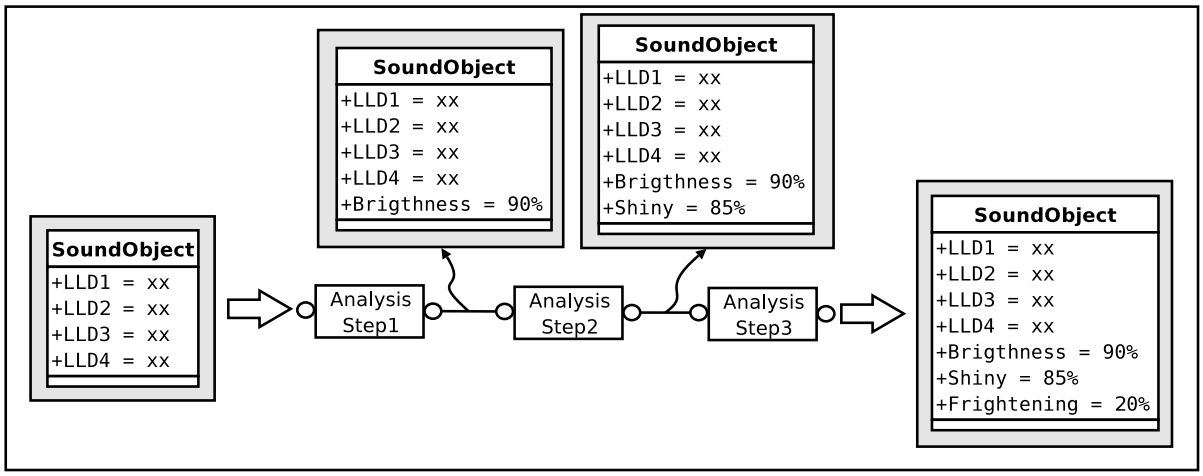

Figure 5.10: Multilevel semantic analysis for adding higher-level abstract features

presented in the previous chapter. At first sight it may seem that all of them are subclasses of the Descriptor class presented in section 3.2.2.2. And that is true for all of them except for one category: the "high-level classifying descriptors". These descriptors carry the description in their own class label and not as attached or independent information. Classifying descriptors are therefore DSPOOM Processing Datas.

§5.2.1.2 The Coding Step (Content Description)

In the coding step, all the content information extracted and organized in the form of objects in previous analysis step needs to be encoded in an appropriate format. Ideally binary and textual based versions of the format should be provided in order to provide both coding and transmission efficiency and readability. It is also important for the coding scheme used to offer support to the way that the output of our analysis block is organized. In that sense, it is necessary to use a highly structured language that enables the description of a tree-like data structure giving also support to object-oriented concepts.

Maybe the first idea that comes to mind is using UML as a way of describing our content. UML is indeed a highly structured language and supports all OO concepts. It would be an excellent choice for describing our Sound Classes. But it is not so appropriate if what we want is to describe the state of our objects/instances or, in other words, make our objects persistent. On the other hand, and as described in section 1.4.3, a quite immediate relation can be established between in-memory objects declared in any object-oriented language and their persistent representation or metadata.

There are many examples of coding schemes used for encoding metadata or, more precisely, audiovisual content description, perhaps the most ambitious being MPEG7 (see 1.4.2). Although MPEG7 is focused on search and retrieval issues, the actual encoding of the audiovisual content description is flexible enough as for being used in applications as the ones envisioned in this thesis [Ebrahimi and Christopoulos, 1998, Lindsay and Kriechbaum, 1999].

Our Content Transmission metamodel does not enforce a particular description scheme such as MPEG-7 but does recommend XML as an appropriate encoding format. Besides, our Sound Objects are in fact DSPOOM Processing Data's and their attached features are DSPOOM Descriptors. If CLAM is used as an appropriate implementation of the metamodel all such object have automatic built-in XML persistence (see section 3.2.2.4). Note that, as a matter of fact, XML-Schema will be the language used for structuring our content or defining Sound Classes, but the actual output of the analysis or content of the identified objects will be a standard XML document.

On the other hand, the encoding step must also be in charge of deciding the degree of abstraction to be applied to the output of the content extraction step. This decision must be taken on the basis

of the application and the user's requirements although it will obviously affect the data transmission rate. The encoder must decide what level of the content tree should indeed be encoded depending on the degree of concreteness demanded to the transmission process, degree that will usually be fixed by the particularities of the receiver. If only high-level semantic information is encoded, the receiver will be forced to use more of its 'artificial imagination' (see next section). The more low-leveled the information encoded is, the more 'real world knowledge' the receiver should have.

Another subject, which will not be dealt with, is how this textual information could be compressed and transformed into a more efficient binary format suitable for transmission. Suffice it to say that different solutions, such as MPEG7's Binary Format[Manjunath et al., 2002], have been found to this issue.

§5.2.2 The Semantic Receiver

The receiver takes incoming content-description in the form of metadata and has to reconstruct the sound. Also, it is a two-step process decomposable into the *decoding* step and the *synthesis* step. Nevertheless in this case, and as we will see in 5.2.2.3, both blocks can be combined into a single functional component.

§5.2.2.1 The Decoding Step (Content Interpretation)

The main task of the decoder is to interpret the information received through the channel in order to be able to feed the Synthesizer with the correct parameters. Encoded sound objects must be interpreted and prepared for the next module requirements. A very common of such requirements is that the Synthesizer does not expect to receive synchronous Processing Data but rather asynchronous Controls. Because of this, and as already illustrated in Figure 5.6, the Decoder may output either Processing Data or Controls depending on the particular needs of the Synthesizer. In case of the Synthesizer needing Controls the conversion from Processing Data to Controls is performed by basically reading the incoming encoded objects and sequencing them as events when the associated time tags correspond to the current synthesis time. From here on we will assume that the Decoder will by default work this way, outputting Controls that are sequenced from Processing Data Sound Objects received through the channel.

The transmitted stream may have a varying degree of abstraction that will affect the way the receiver will respond. The stream may contain from signal related processing data with associated low-level descriptors to high-level processing data representing high-level classifying descriptors. The

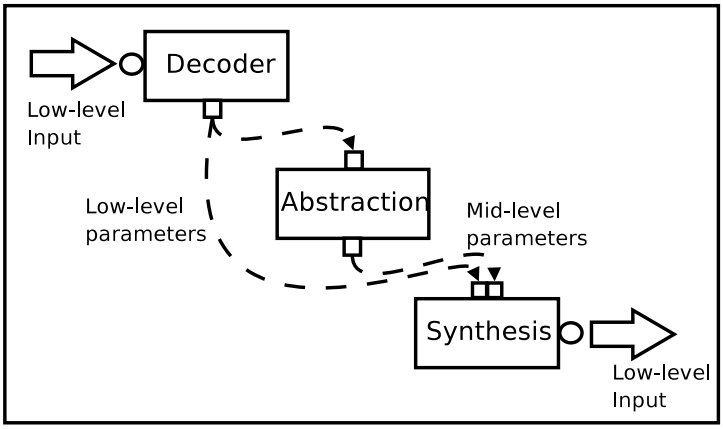

Figure 5.11: Low-level input to the Decoder: the Abstraction process

way the receiver has to process the input stream depends on how high or low-leveled the content description received is. Two main processes are involved in bringing the description into the optimal *level: abstraction* and *inference*. We will now detail their main characteristics.

If the decoder is input low-leveled descriptions, there are two options, depending on the application requirements. The low-level descriptors can be directly fed into the Synthesis engine or there can be an intermediate *abstraction process*. At first sight it may seem that the first approach is more sensible and obviously more economical. But it has a inherent problem that is difficult to solve: if the description is very low-level it also has to be exhaustive and this, in many situations, is not easy to accomplish or is not worth it in terms of channel bandwidth (we may end-up having a description that is a few times the original size). An example of a situation where a non-exhaustive low-leveled description is received would be an input like "sound object, centroid=120Hz". It is obvious that many sound objects comply with this low-level description, the decoder would be in charge of adjusting other necessary parameters.

For all those reasons it may be usually interesting to include the intermediate abstraction process. In this process the decoder has to use 'real world' knowledge in order to convert low-level information into mid-level information, more understandable from the synthesizer point of view. If the abstraction process is omitted and the synthesizer receives low-level information but this description is not exhaustive, those parameters not specified should be taken as default. Thus, paradoxically, the synthesizer is granted some degrees of freedom and the result may loose concretion.

On the other hand, if the input to the decoder consists only of high-level semantic information, an intermediate *inference process* is always needed in order to make the content description understandable by the synthesis engine. This process, contrary of the abstraction process earlier mentioned, might

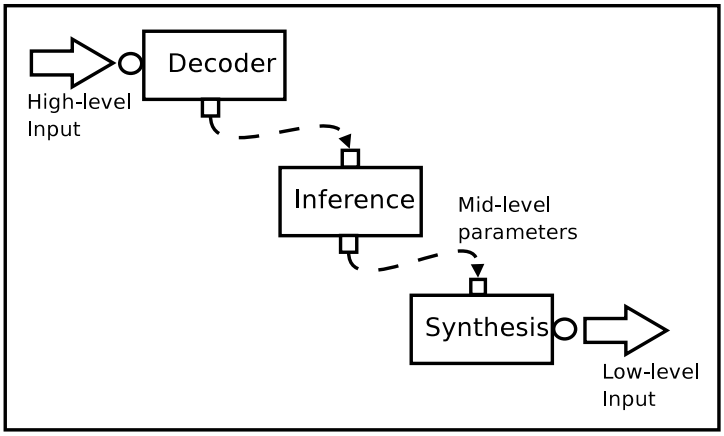

Figure 5.12: High-level input to the Decoder: the Inference process

be better understood by using an example. Imagine the decoder's input is 'violin.note'. The synthesizer will be unable to interpret that content description because of its degree of abstraction. The decoder is therefore forced to lower the level of abstraction by suppressing degrees of freedom. The output of the decoder should be something like 'violin note, pitch: C4, loudness: mf'.

Both abstraction and inference are indeed one-to-many process, that is, the same input should yield a finite set of different outputs. The way the decoding process gets rid of the degrees of freedom should rely on user or application preferences as well as on random processes or context awareness. In the previous example, the decision on the note and loudness to be played could be based on knowledge on the author, the style, the user's likes, previous or future notes, harmony and a final random process to choose one of the best alternatives.

In any case, the decoder must translate the input metadata into some sort of synthesis language that can be easily interpreted by the synthesizer. Therefore the key point of the language used for expressing synthesis parameters is that it must not only meet the requirements of the synthesizer's input but also the needs of the decoder's output. Note that this translation means in most cases a translation from Processing Data into asynchronous control events as most synthesis languages handle simple events, not complex synchronous data.

§5.2.2.2 The Synthesis Step

As commented in the previous paragraphs the decoder is in charge of translating the input metadata into a synthesis language. Once the transmitted metadata has been translated into some language/format understandable by the synthesizer, the synthesis step is reduced to a traditional synthesis process. We therefore only need the synthesizer to be prepared to respond to the different decoded

parameters.

Therefore, in the synthesis step the key issue is the language of choice. Many languages have been developed for the purpose of controlling a synthesizer. Among them, the most extended one is MIDI [MMA, 1998] although its limitations make it clearly not sufficient for the system proposed in this paper. Another synthesis language that deserves consideration at this point is MPEG4's SAOL (Structured Audio Orchestra Language) [Scheirer, 1999b]. See section 5.3.2 for a more in depth explanation on Structured Audio and its relation to the OOCTM.

But, as it will be explained in section 5.3.2, Structured Audio and SAOL present several limitations and are not well suited for our purposes. In next chapter, an object-oriented synthesis language is presented. The MetriXML language is proposed as a link between analysis, encoding, and synthesis specification and it presents a model of music that is a particular instance of DSPOOM and very much related to OOCTM. Note that a further advantage of the MetriXML is that it is an XML-based language, just as the one proposed as result of the encoding process. A transformation from one XML document containing analysis results and another XML document containing synthesis parameters can be as simple as defining an XSLT[4].

§5.2.2.3 A Combined Receiver Scheme (Content-based Synthesis)

Although sometimes it may be useful to conceptually separate the receiver into a decoder and a synthesizer, many other times, a combined scheme that treats the receiver as a whole will be more feasible.

In that case, the resulting receiver scheme is what we call a Content-based Synthesizer, or Object-based Synthesizer which, at first sight, does not differ much from that of a traditional synthesizer. As illustrated in Figure 5.13 the input metadata is converted to control events and mapped to synthesizer paramatersparameters.

In a general situation, a simple mapping strategy may be sufficient. But if the level of abstraction of the input metadata is higher, the gap between the information transmitted and the parameters that are to be fed to the synthesis engine might be impossible to fill using conventional techniques. Imagine for example a situation where the transmitted metadata included a content description such as: (genre: jazz, mood: sad, user_profile: musician).

The latter example leads to the fact that we are facing a problem of search and retrieval more than one of finding an appropriate mapping strategy. We could have a database made up of sound files with an attached content description in the form of metadata. The goal of the system is then to find

[4]XSLT is a language for transforming XML documents into other XML documents (see [W3C, 1999])

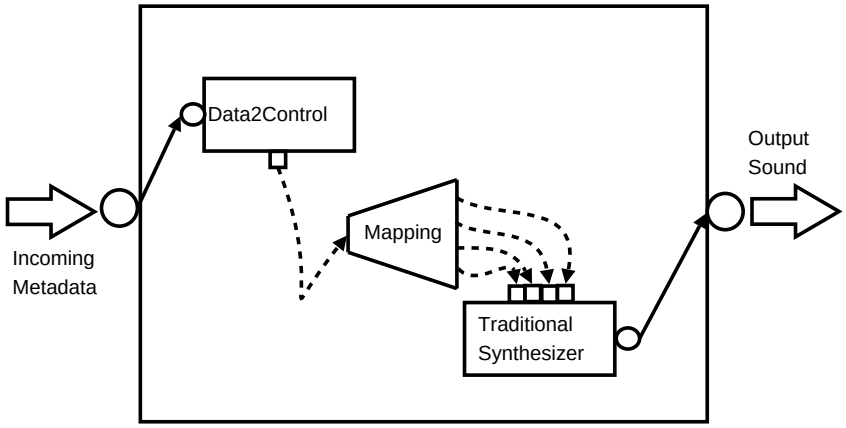

Figure 5.13: Combined scheme for modeling the receiver in a content-transmission system

what object in the database fulfils the requirements of the input metadata.

A problem we still have to face with such a model is the difficulty to automatically extract parameters with such a level of abstraction from the signal itself. We can find examples of existing applications that implement the system depicted in Figure 5.14 but they always need a previous step of manually annotating the content of the whole database.

A possible solution to this inconvenience is the use of machine learning techniques. It is recently becoming usual in this sort of frameworks to implement, for example, collaborative filtering engines (classification based on the analysis of users preferences: if most of our users classify item X as being Y, we label it that way). In that case though, the classification and identification is performed without taking into account any inner property of the sounds. On the other hand, if what we intend to have is a system capable of learning from the sound features, we may favor a Case-Based Reasoning (CBR) engine as the one used in [Arcos et al., 1998].

Anyhow, a first precondition for deciding on the system's viability would be to reduce the size of the resulting database. We observe though that there is no need to store sounds that could be easily obtained from other already existing in the database. In the case that no sound exactly matched the content description at the input we could then just find the most similar one and adapt it in the desired direction. This adaptation step is basically a content-based transformation (see section 5.3.4)[5].

One of the problems that still remains is what similarity measure the system has to deal with. Similarity in sound and music is obviously a many-dimensional measure that can be highly dependent on a particular application. Furthermore, it may turn out that our database has more than one case that is similar to the content description received. All of them may need a further adaptation (transformation) but the problem is how to decide on what transformation is more immediate and effective. In that sense,

[5]Different examples of how to accomplish this with a transformation or an interpolation will be seen in the next chapter.

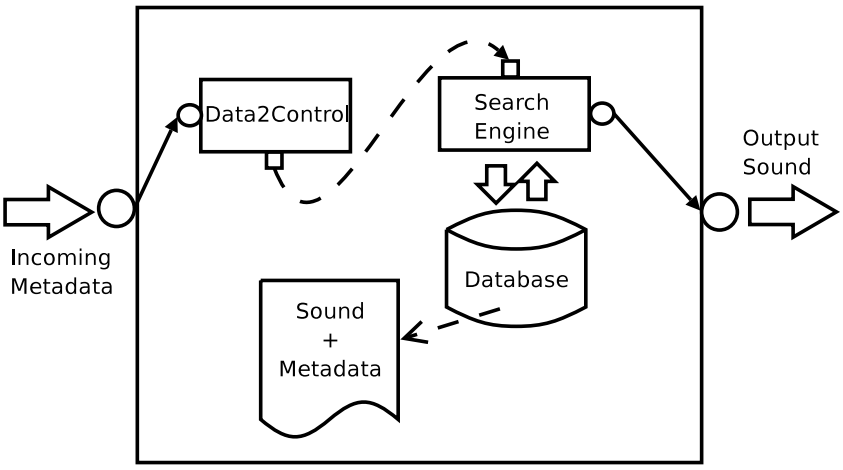

Figure 5.14: Search and retrieval as a means for synthesizing

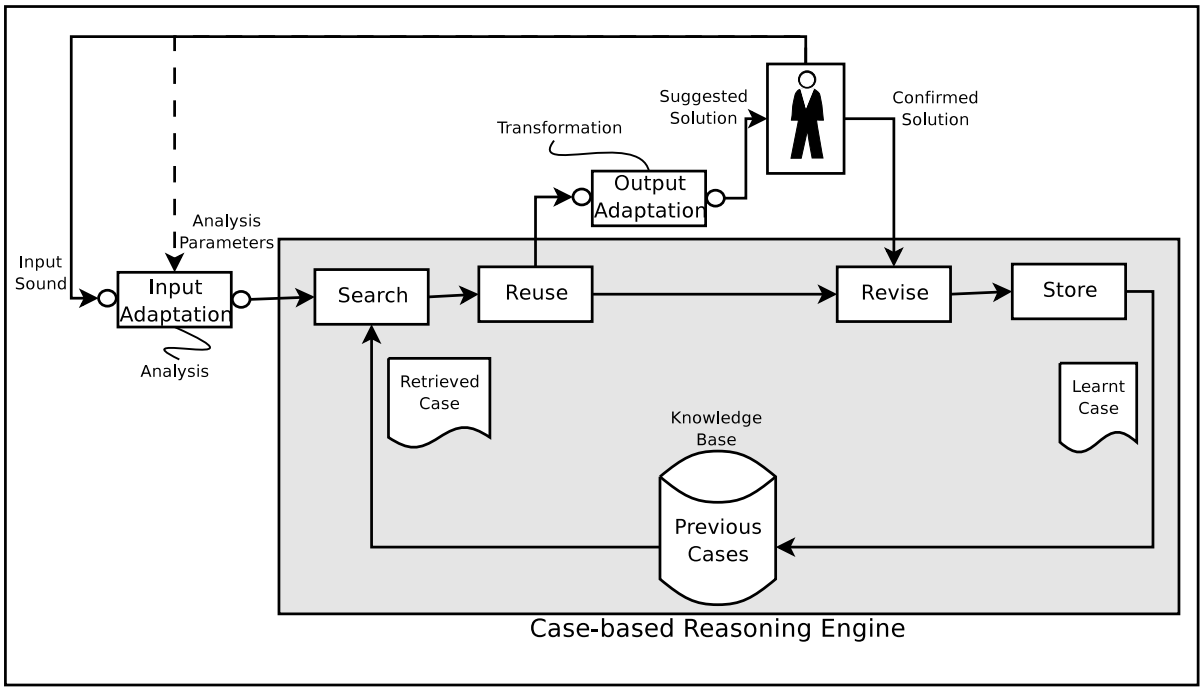

Figure 5.15: A Case-based Reasoning Receiver

it may be interesting to identify and classify items for the database not only for what they actually are but for what they may become. A sound can thus be classified as bright-able, piano-able, fast-able . If a solution is confirmed as accepted by the user we may not only add the resulting sound and its content description to the database but also the knowledge derived from the adaptation process.

§5.3 OOCTM and related Models

Once we have understood the motivations and the structure of the Object-Oriented Content Transmission Metamodel it is well worth it to compare it to related models or metamodels.

§5.3.1 Beyond Shannon&Weaver's Model of Information Transmission

The Content Transmission metamodel that we have just introduced implies a redefinition of the schemes commonly accepted for the communication act itself (see [Darnell, 1972] or [Griffin, 1997], for instance, for a comprehensive listing of such models) as it can be seen as a step beyond Shannon and Weaver's traditional communication model [Shannon and Weaver, 1949].

§5.3.1.1 The Shannon&Weaver Model

The most commonly accepted and spread communication model is that proposed by electrical engineer Claude Shannon in 1949 and then interpreted by Warren Weaver. It must be noted that particularly Shannon's model was intended to be an information transmission model, mostly applicable for engineering purposes. Nevertheless, Weaver's interpretation, as well as further later uses converted it in a general usage communication model.

Shannon and Weaver introduced the linear model represented in figure 5.16. The goal of any communication system is to transmit a *data source* to a particular *destination*. The transmission is carried out through a physical *channel* that is bound to influence the message, a non-ideal transmission channel can indeed be considered a *noise source*. The engine in charge of preparing the data source and sending it through the channel is called the *transmitter*. This transmitter is in charge of different process but the most important is that of *encoding* the message so it can be transmitted more efficiently and it can become more robust to the effect of the channel. The *receiver* is in charge of obtaining the message from the channel, decoding it, removing noise as much as possible and deliver it to the final destination.

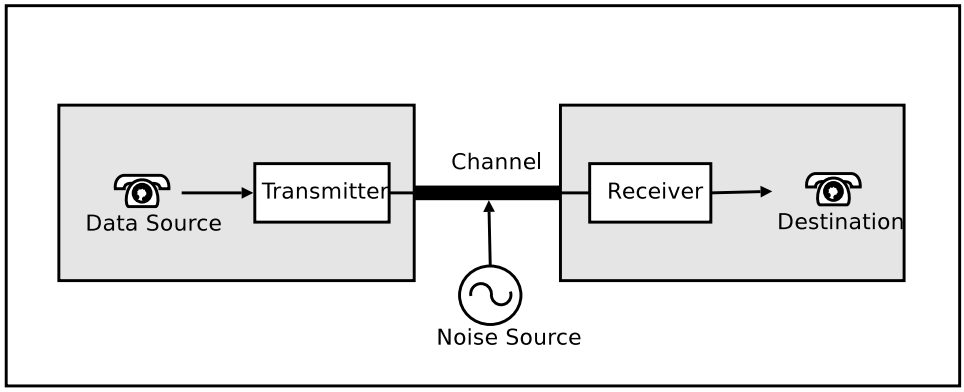

Figure 5.16: Shannon & Weaver's classical information transmission metamodel

In this metamodel, *information* is thought of as the opposite to *entropy*. Meaning is unimportant from that mathematical point of view. According to Shannon and Weaver "the semantic aspects of communication are irrelevant to the engineering problem" [Shannon and Weaver, 1949]. As Eric Scheirer points out this is not an engineering assumption but rather a philosophical one [Scheirer, 2001].

In a similar way, in S&W's model *noise* is anything added by the channel and not intended by the source. The way to overcome the unwanted effects of noise is by adding redundancy. As a matter of fact the S&W theory considers that communication is the science of maintaining and optimal balance between predictability and uncertainty by adding or removing redundancy.

§5.3.1.2 The Object-Oriented Content Transmission Metamodel as a Metamodel for Information Transmission

In Figure 5.1 we illustrated the main components of our Object-Oriented Content Transmission Metamodel, which were furthered develop in the following sections. By comparing this metamodel to the one just described by Shannon and Weaver we can highlight the following differences:

In our model, the stream to be transmitted is no longer seen as a stream of bits with no abstract meaning, information is an abstraction of the actual content, in other words, a stream of meaning. We are therefore not interested in removing data redundancy as such but rather on transmitting the appropriate meaning that the source carries within. And what is appropriate is, as in any other kind of model, related to the context, application or what the receiver is expecting to receive and able to interpret. In a general situation, we do not worry about fidelity to the original data but rather to the original meaning.

In this sense, noise is thought of as anything added to the original piece of information that is likely to change its meaning or make it difficult to understand. Thus, the traditional definition for noise

as a change in the bitstream being transmitted, would only fit our definition if the change is substantial and can produce a change of meaning.

Our channel does not transmit encoded data as such but rather extracted content description or meaning. Because of this Shannon's laws of channel capacity do not hold true and we are able to well exceed its theoretical limits.

On the other hand, and as Cockburn points out [Cockburn, 2002], S&W's metamodel considers a constrained channel while in human communication the channel is unconstrained and the success of the information transmission process depends on the *shared experience* between the transmitter and the receiver, which enables them to use a common vocabulary. In our metamodel, the success of the process is also based on the shared experiences or knowledge between the transmitter and the receiver. As a matter of fact this is only constrain.

In order to accomplish all this the traditional S&W metamodel has to enhanced and particularly two new blocks have to be added.

On one hand, in the transmitter we need to analyze and extract content or meaning from the original data source. This process was explained in section 5.2.1. On the other hand, the transmitted message has to be interpreted and rendered into new data in the receiver. This process was also explained in section 5.2.2.

§5.3.1.3 Other Models of Communication that care about Meaning

Although the Shannon and Weaver theory has deeply influenced all research areas related with communication and we have presented our metamodel as a step beyond that observes the importance of meaning, it is important to note that some models of human communication do care about meaning and make a central issue of it. It is well beyond our intention to give a thorough overview of such models, mostly related to psychological or sociological arenas but it is interesting to include a brief outline in which we will find some relation with the metamodel we have just presented.

Charles Osgood is the creator of the *Meditacional Theory of Meaning [Oswood, 1976]*, related to his well known *semantic differential* technique [Snyder and Oswood, 1967]. According to the results of his studies the meaning of any piece of information can be described in terms of only three dimensions: its evaluation (whether it is good or bad), potency (how strong it is) and activity (how fast it moves).

Pierce and Cronen developed a model of communication that strongly relies on meaning. This model is known as the Coordinated Management of Meaning (CMM) [Philipsen, 1995] and its main feature is that instead of focusing on the message, the stress is put on the receiver. People try to understand the world assigning meaning to an event, the problem is that individual interpretations may

not coincide. The CMM represents meaning in a six level map that goes from the actual *content* to the highest level of *cultural patterns*.

In a similar way the General Semantics Theory[Kodish, 1993] states that you cannot equate a word with the concept that it's supposed to represent. Hayakawa proposes an 'Abstraction Ladder', the actual content can be decomposed into different representations according on the level of abstraction intended. The problem is that as we go higher on the ladder, there is a point where 'speakers' lose the picture of the actual content. He gives the example of an Abstraction Ladder for a picture of a cow. This ladder includes the following levels: (1) Cow as known to science; (2) the actual cow we perceive; (3) "Bessie" (the cow's name); (4) Cow; (5) Livestock; (6) Farm assets; (7) Asset; (8) Wealth. This abstraction ladder is directly related to the multilevel analysis and content description scheme we presented in section 5.2.1.

I.A. Richard developed a theory on the "Meaning of Meaning" in which he suggested that context is key to meaning [Ogden and Richards, 1946]. Meaning is personal and in his view of the communication act he extended Shannon and Weaver's traditional model to show the necessity of common experience for the effective transmission of meaning. This idea is also related to the different strategies presented for our Semantic Receiver, outlined in section 5.2.2.

Symbolic Interactionists, represented by George Herbert Mead, state that the extent of knowing is the extent of naming: intelligence is the ability to symbolically label everything we encounter [Mead, 1910].

Harold Lasswell [Cherry, 1957] suggested a simple but useful model, which captures the essence of message transmission and is made of the following elements: Who, says What, to Whom, in Which medium, with what Effect.

Finally it is also important to acknowledge that also some other researchers in the audio processing field noted that S&W's disregard for meaning imposed too many restrictions on our domain. In particular, the developers of the Structured Audio metamodel, which will be presented in next section, also presented their model as a step beyond S&W's.

§5.3.2 Beyond Structured Audio

The Structured Audio metamodel is very much related to the one being presented in this chapter. This new paradigm was formalized in the MIT's Machine Listening Group with the work of Professor Barry Vercoe and some of his students (especially and mostly Eric D. Scheirer).

What follows is a brief description of this metamodel and its relation with our Object-Oriented

Content Transmission Metamodel.

§5.3.2.1 The Structured Audio metamodel

Structured audio is a way of representing audio information in which semantic information and high-level algorithmic models are used. Examples of existing and well-known structured audio formats include MIDI, any music synthesis language and the linear-prediction coding (LPC) model for speech signals. The term Structured Audio was introduced by Vercoe as a way of interrelating research on sound synthesis, audio coding, and sound recognition [Vercoe et al., 1998].

A structured media representation encodes a signal according to a model making assumptions about the input signal and deriving a parameter space. This property, though, does not suffice to make a representation structured. As a matter of fact any media representation has an implicit or explicit model but the less dimensions its related parameter space has and the more meaning these parameters have, the more structured the representation. Semantic parameters that represent high-level attributes give control over perceptual and structural aspects of the sound and provide for more interesting manipulations. Structured audio representations are parametric in the sense that they are based on a model by which two sounds can be distinguished according to some parameter values.

Audio compression or encoding technology mostly relies on two kind of coders: entropic or lossless coders and perceptual or lossy coders. Entropic coders exploit information-theoretic or Shannon's redundancy and perceptual coders exploit perceptual redundancy or irrelevancy (if sound X and sound Y are perceptually indistinguishable, it does not matter which one is transmitted).

Structured audio coders exploit yet another kind of redundancy: structural redundancy. Most sound signals contain structural redundancy in different ways. Many notes, for example, in a musical track sound the same or nearly the same. Not only a middle C note may be substituted by its model but neighboring notes such as C# may be obtained by transforming the original model algorithmically [Scheirer, 2001]. Another example of structural redundancy is that many sounds are more simply represented as processes than as waveform. A reverberated speech signal, for instance, may be better transmitted by encoding separately the flat speech and the description of the reverberation algorithm)[Vercoe et al., 1998].

Although structured audio is not perceptual encoding, a fundamental issue in structured audio is how listeners perceive the sound and thus how structural parameters affect perception. Structured audio is not interested in usual engineering properties such as perfect reconstruction. A minimum squares measure of the error is not useful because humans do not perceive sound this way.

So, in a structured audio application sound is coded not based on perceptual or information-

theory related compression but rather representing its structure. This structural description of sound is then transmitted to a receiver which reconstructs the sound by executing real-time synthesis algorithms. All audio signals are more or less structured and neither entropy nor perceptual encoders exploit this feature. It is interesting to note, as Scheirer suggests in [Scheirer, 2001], that using structured audio it is possible to transmit data at a lower rate than that suggested by the Shannon rate distortion theory, which only becomes an unsurpassable limit if really random signals were to be transmitted and that is in practice seldom the case.

A coding format consists on two parts: a bitstream description that specifies the syntax and semantics of data sequences and a decoding process, which is an algorithm that describes how to turn the bitstream into a sound. A bitstream is a sequence of data that follows some particular coding format and represents a compressed sound when its length is shorter than the sound it represents.

What makes structured audio coding different is that the model is not fixed but rather dynamically described as part of the transmission stream. As a matter of fact, structured audio can be considered a framework or metamodel that can be used to implement all other coding techniques [Scheirer and Kim, 1999]. It can be proven that the best-known performance of a fixed audio coding method serves as the worst-case estimate of structured audio coding [Scheirer, 2001].

Structured Audio allows ultra-low bitrate transmission of audio signals. It also provides perceptually driven access to audio data. Bearing in mind these two main benefits, many applications may be envisioned: low bandwidth transmission; sound generation from process models (e.g. as in video games or virtual reality applications); flexible music synthesis by allowing the composer to create and transmit synthesis algorithms along with the event list; interactive music applications; content-based transformations and manipulations; and content-based retrieval.

Arguably, out of these applications the most important one is the transmission of ultra-low bitrate audio signals. Structured audio provides excellent compression when models that can be controlled with few parameters are available. This may be so, for example, if the space of sounds to encode is reduced. For example, if only plucked strings are to be transmitted, a plucked string model can be transmitted first and then send only the parameters that control this model. The more structured sounds are the more they can be compressed[6]. Compression rates of up to 10000:1 can be accomplished with structured audio on some particular signals [Scheirer, 2001].

The structured audio metamodel made its way into the MPEG4 standard mainly thanks to the work of Eric D. Scheirer from the MIT's Machine Listening Group. Now seen in some perspective

[6]Although the term "noisy" is sometimes used by the authors as a synonym of "unstructured" we think this term is very misleading. As a matter of fact, a white noise signal is very structured , according to the Kolmogorov complexity theory it can be sent as structured audio with no perceptual loss in a very compact synthesis algorithm with almost no control parameters.

it seems to us that Structured Audio was not mature enough to make it into an standard. It has hardly found any practical application and it has become outdated very soon after. Furthermore, its limitations and too strict specifications make it hard to adapt to future needs.

It is far beyond the scope of this document to give a thorough overview of the standard and we refer the reader interested in structured audio in MPEG-4 to [Scheirer et al., 1998, Scheirer, 1999a, Scheirer and Kim, 1999, Scheirer, 1998b, Scheirer, 1999b, Scheirer, 1998a, Scheirer et al., 2000]. Nevertheless, it is important to highlight that there are five major elements in the MPEG-4 Structured Audio (SA for short) toolset: a Structured Audio Orchestra Language (SAOL), a Structured Audio Score Language (SASL), a Structured Audio Sound Bank Format (SASBF), a normative scheduler, and a normative reference to several MIDI standards that can be used in addition or instead of SASL. Note that the two most important components, SAOL and SASL, were already introduced in section 2.6.1.3.

§5.3.2.2 Structured Audio and the Object-Oriented Content Transmission Metamodel

After the previous description it must have become clear that Structured Audio is very much related to our Object-Oriented Content Transmission Metamodel. We will now highlight the main differences of both approaches.

Structural audio's focus in on *structure*: it is designed to exploit structural redundancy. The focus of our OOCTM is on *content* and its *meaning*: we aim at *understanding* the signal and representing it accordingly. Although the final results in some particular applications might not differ much, the difference in focus is clear: SA is a syntactic metamodel while OOCTM is a semantic metamodel.

According to the authors Structured Audio can surpass the Shannon&Weaver theoretical channel limit except if the encoded signal was completely random. Our metamodel can surpass the limit even if the signal is completely random. As a matter of fact, OOCTM performs extremely well in such situations. If the data source is completely random then it means that it has no meaning. Therefore, it can be transmitted as a simple "play random signals" sound object that will be rendered at the receiver. The result will obviously not resemble the original signal but we insist that we are not interested in mathematical nor perceptual accuracy, only in semantic accuracy. SA performs well on highly structured signals, the OOCTM performs well on both highly meaningful and meaningless signals just as long as a meaningful signal is not classified as meaningless.

As already commented at the beginning of this chapter the idea of *synthesizability* in SA is different from ours. While in SA a given representation is said to be synthesizable if the original sound can be obtained from it in OOCTM a representation is synthesizable simply if it can render a sound. Whether the result is meaningful or not relates to other measures of the description such as

meaningfulness, but not to its *synthesizability*.

SA encodes data sources parametrically, based on signal models. Nevertheless, it does not impose any limitation or conceptual metamodel on these models. The OOCTM encodes data sources as Sound Objects and forces these object-oriented data models to comply with the DSPOOM metamodel. Object-oriented data models can be interpreted as a subset of Parametric models, therefore OOCTM is more restrictive in that sense. But it is only because of this restriction that we can ensure that all models that might be instantiated from the metamodel will carry some semantic information.

In SA the message always includes the transmission of a particular model, to be used by the receiver. Although model transmission can also be provided and used in OOCTM it is not compulsory. In many situations the model may be known beforehand as all components share the same metamodel and may be able to deduce it. On other situations, the degree of abstraction might be so high that no model is necessary at the receiver except from "real-world" knowledge.

Structured Audio needs to standardize language with precise semantics such as SAOL or SASL. In the Object-Oriented Content Transmission Metamodel no languages need to be standardized. XML is used as a general purpose content-description language but any other similar general purpose language could be used. Different particular instances of XML can be used (see MetriXML) but they are not part of the metamodel. In our metamodel even the language description can be transmitted dynamically in a schema.

A key issue in the OOCTM is that the same language is consistently used throughout the metamodel and in any of its components. This is something that cannot be said about SA. SAOL and SASL are basically synthesis languages and they are not suitable for encoding the result of a general signal analysis, even less if this analysis addresses the content level. This fact has been explicitly recognized by the MPEG working group when constructing a completely different standard, MPEG-7 (see section 1.4.2). MPEG-7's content description and MPEG-4's Structured Audio tools are not even compatible and efforts to bring both world together, if ever started, are, in our opinion, not going to succeed.

Our OOCTM is a particular instance of the Digital Signal Processing Object-Oriented Metamodel. And DSPOOM does not stand on any specific language but is rather instantiated in a framework such as CLAM implemented in a general purpose programming language.

It is interesting to note how the Eric Scheirer, main creator of MPEG-4's Structured Audio, points out that a general-purpose computer programming language could be used instead than a language like SAOL, specifically designed for structured audio description (see [Scheirer, 2001]). According to him the approach of having a specific language has a number of advantages, namely:

- MPEG4 structured audio behaves like an audio decoder, it accepts audio blocks, runs real-time, communicates with a DAC, etc...
- General purpose programming languages cannot satisfy system level requirements (no portable way of connecting to a DAC...).
- SA is written with the implementation of costum hardware in mind. The fundamental constructs of SAOL are those that run efficiently on DSP's and thus will run more efficient than C or Java implementations of these algorithms. It is interesting to move the processing to off-side processors.
- It is more convenient to write algorithms in SAOL due to the number of available primitives

A Software framework such as CLAM provides most of the aforementioned advantages. It accepts audio blocks and may run real-time; it can satisfy system level requirements thanks to its operating system abstraction; and it provides even more primitives than Structured Audio. The only point that it does not satisfy is that it is not written with custom hardware in mind, but we see this rather as a disadvantage than as an advantage of SAOL.

He also states that the process of decoding the bitstream header and reconfiguring the synthesizer is similar to parsing and compiling a computer language. According to [Scheirer, 1998b] for implementing a SAOL system, similar skills in software engineering and computer science than those used for implementing a compiler are needed. Then why waste all those resources when there are many general purpose programming language compilers that can do the job?

§5.3.3 Beyond Parametric Encoding: Content Analysis

Any parameterized audio coding format can be considered as a combination of: a sound-understanding algorithm that sets parameter according to some model by analyzing the audio; the transmission of these parameters; and a sound-synthesis algorithm that maps from the transmitted parameters to a new sound [Scheirer, 2001].

The understanding step is usually called *encoding* and the synthesis step *decoding* but if the parameters are obtained directly from a sound, the process may be also termed as analysis/synthesis. If we want to communicate the sound description over a channel we need an encoder and a decoder, both with a priori knowledge of the model being used. But it is impossible to devise an encoding method that always gives the optimal coding of any input sound. In other words, not a single model can fit all kinds of input signals. For this reason we may sum up different specialized models and add some extra bits to the stream in order to indicate what model is being chosen.

Parametric encoding has been used extensively in speech transmission, where a general model of the input signal has existed for a long time. Most commonly used parametric models for speech are simple variations of the classical Linear Predictive Coding (LPC) scheme[Makhoul, 1975], which somehow exploits the knowledge of the vocal production mechanism.

Parametric coding for non-speech signals is much more complex to implement as not many assumptions can be made about the source characteristics or its production mechanisms. MPEG-4 standardized different parametric encoding schemes although the one recommended for non-speech or musical signals is the HILN or Harmonic and Individual Lines plus Noise (see [Purnhagen and Meine, 2000]). This parametric scheme is as a matter of fact a variation of the SMS or Spectral Modelling Synthesis technique (see further explanation in annex B).

The Object-Oriented Content Transmission metamodel represents a step beyond parametric encoding. The basic scheme is the same: the sound is analyzed, some parameters are extracted, transmitted and then synthesized back at the receiver.

But the main difference is on the kind of parametric model used in each case. In traditional parametric encoding, the model makes assumptions about the low-level features of the signal. For instance, the input signal may be supposed to be harmonic and therefore modeled as a set of time varying sinusoids (see Annex B for a more in depth explanation of one of these models).

In the Object-Oriented Content Transmission Metamodel we make assumptions about the semantics of the input data and the way it is organized in the real world. We may, for instance, assume that the input signal will be monophonic musical phrases and then apply a model in which a musical phrase is modelled as a sequence of *notes*.

The OOCTM encodes data as Sound Objects. This is, as already mentioned in the previous section, a particular case of Parametric encoding. Parameters extracted from the signal become attributes of concrete objects. In that sense, all parameters have a particular meaning and contribute to the content description. The final encoded signal also has a clear structure, given by the resulting object-oriented model.

In that sense, the result of an OOCTM encoding is generally understandable while a Parametric encoding is usually not.

Finally, in a Parametric encoding scheme the synthesis capabilities of the receiver are very much limited to a simple signal model. In the OOCTM scheme the synthesizer is able to not only apply different models but also to infer or abstract a sound from an incomplete description (see 5.2.2).

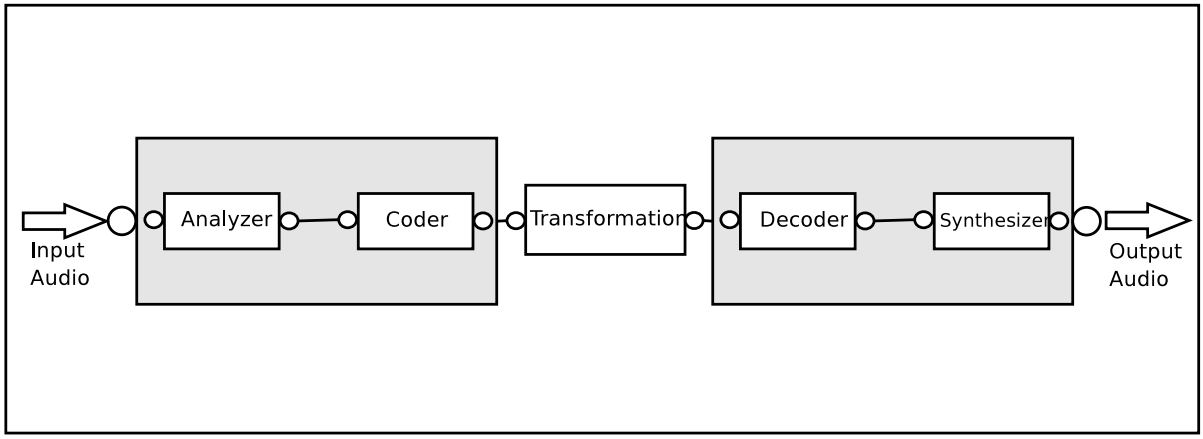

Figure 5.17: Content-based transformations and the OOCTM

§5.3.4 Beyond Sound Effects: Content-based transformations

A particular instance of the Object-Oriented Content Transmission Metamodel is that of Content-based Transformations (see [Amatriain et al., 2003]). As we will see, all the different components in the OOCTM can be involved in such a transformation. The only difference now is a shift in the final goal: instead of transmitting the original object, now we aim at changing it in meaningful ways.

As a matter of fact in a broad sense a content-based transformation only introduces a minor difference in the basic block diagram of the Object-Oriented Content transmission Metamodel introduced in figure 5.1. As illustrated in figure 5.17 the *channel* of the basic metamodel is now replaced by a *Transformation* Processing object. Nevertheless, throughout this section we will present different content-based transformation as particular instances of this metamodel.

When we use the term *transformation*, we use it in a different way from how we would use the word *effect*. When we talk about an effect, we are focusing on the result of changing the sound in a particular way. However, when talking about a transformation, the strength is put on the change that a particular sound undergoes, rather than on the result. Thus, not every sound can undergo a certain transformation, yet an effect can be applied on any source regardless its properties. That is the reason why we use the word transformation when addressing the content level.

Just as in the general OOCTM, in order to be able to apply some kind of content transformation the signal must undergo a previous analysis step. The goal of this step is to compute features that will then be relevant in the transformation step.

The first possible scenario is the one represented in figure 5.18. The output of the analysis is used as a side-chain control to the transformation block. The aim of this analysis is therefore not to

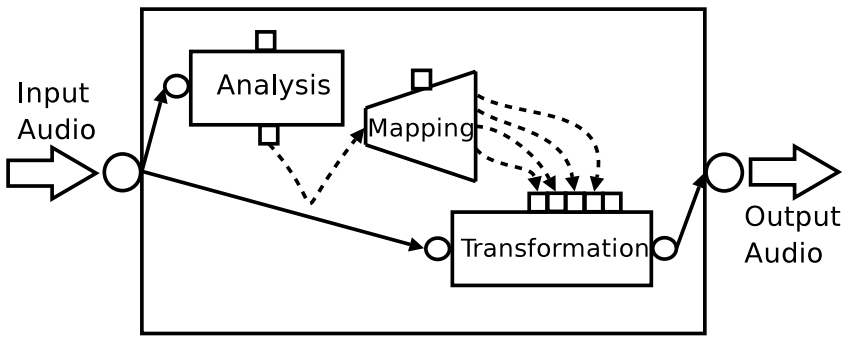

Figure 5.18: Basic content transformation scenario: analysis output is used as a transformation control signal

extract meaning or identify objects but rather to simply extract some partial features that will be used as control. The transformation is then applied to the original sound directly. Note that, in this case, the user input is not used in the transformation chain so the scheme could be labeled as unsupervised. The parameters of the transformation are dynamically adapted to the characteristics of the input signal.

A very basic example of this kind of signal processor is an automatic gain control. Such a system can reduce or increase its gain depending on the relation between the input signal and a given threshold. When the signal exceeds that particular threshold the gain is reduced and the transformation is said to be a compressor (or a limiter if the slope is smaller than 1/10). On the other hand, if the signal is below the threshold, the gain is increased and the transformation is known as an expander. One may argue that this sort of signal analysis is too low-leveled to be included in the category of content-based transformation but we refer again to the definition of content previously introduced. The content description of the signal is being reduced to just a very simple feature: its level. In any event, it is clear that the transformation depends on the analysis of that particular feature applied to the incoming signal.

Most of the transformations implemented in the time-domain can fit quite well into any of the variations of the previous model. The implementation of the processing algorithms is quite straight forward and based on a sample-by-sample process. Examples of transformations that can be effectively implemented using these techniques include those related to effects like delays, chorus, reverbs, or dynamic processors

But sometimes the information that can be immediately gathered from the signal and its time-domain representation may not be enough in order to design a particular meaningful transformation. In such situations, the analysis step must yield more than just a set of features to be used as control signals. Thus, in order to achieve more interesting transformations we need to find a model for the signal in such a way that this intermediate representation is more suitable for applying some particular

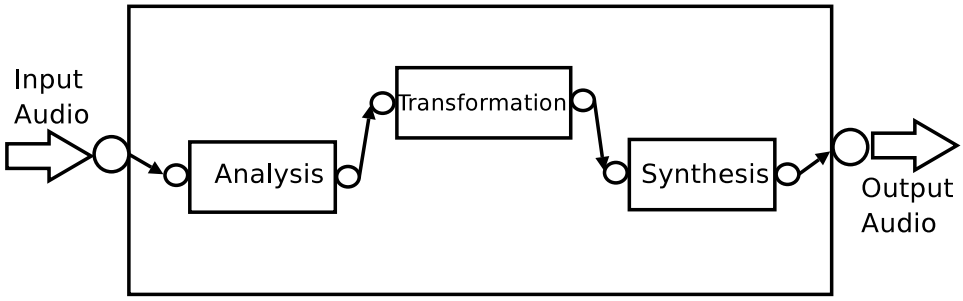

Figure 5.19: Content transformation process based on an analysis/synthesis framework

processes. Therefore, this analysis process is now analogous to the OOCTM semantic analysis step (see 5.2.1).

Figure 5.18 illustrates the new situation in which the signal is analyzed, transformed and then synthesized back (see [Serra and Bonada, 1998, Amatriain et al., 2001]). Note also that this block diagram is, as a matter of fact, the same as the one introduced in figure 5.17 but including the coder in the analysis object and the decoder in the synthesis object.

Sometimes, the analysis step may be skipped because the input stream already contains metadata that can be used for the transformation process. In this case we may not need to instantiate the transmitter side of the OOCTM because this process has been executed somewhere else or is available in the original data (remember that some higher-level such as the title of a piece may have been manually annotated). Figure 5.20 illustrates this situation.

An example of such a transformation would be, for instance, a genre-dependent equalization. By applying some of the existing genre taxonomies we could add metadata defining the genre of a given piece of music. The classification could be performed either manually or by using a combination of previously existing metadata that included, for example, author and title. The transformation block would then implement a basic filtering process that loads different filtering function templates depending on the genre.

Arguably, even another form of content transformation is that based on context awareness. By context awareness we mean the ability of a particular system of becoming aware of its surrounding world. In that sense, a dynamic processor whose threshold depends on the noise-level of the room would be an example of such a scheme, illustrated in figure 5.21.

Furthermore, context awareness is very much related to user profiling. A transformation system can respond differently according to the loaded user model. This user model can include information about user preferences as well as contextual information such as whether the user is happy or not [Chai and Vercoe, 2000].

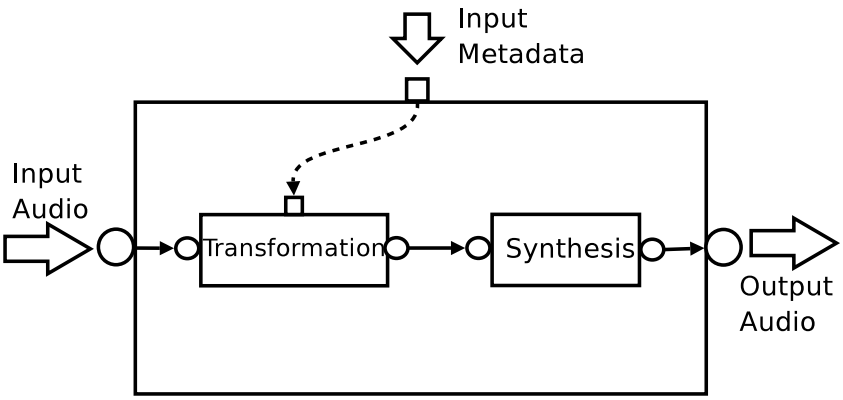

Figure 5.20: Content description in the form of metadata as a secondary input

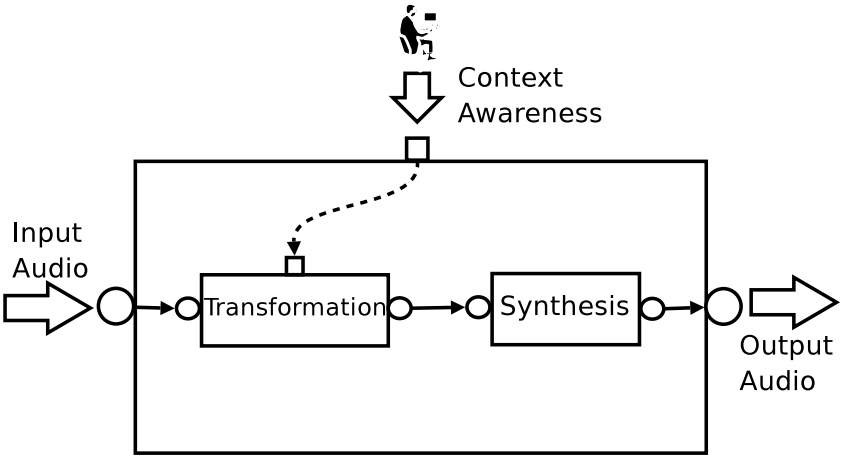

Figure 5.21: Context awareness as a means of control

But even in such a simple example as the one of the automatic gain control, the user input must somehow be taken into account (the threshold and the slope must somehow be set). In that sense, the previous scheme must be modified in order to include this new input. Figure 5.22 illustrates all possible inputs to the transformation chain.

A first version of the new scenario feeds this information directly into the analysis process so the user can control the settings of this particular step. The influence of the users actions is directly on the features extracted from the signal.

Furthermore, the user may be able to directly interact with the output of the analysis process and so change the characteristics of the sound features before using them as a control of the actual transformation. Now, the influence of the users actions is on the mapping function between the features extracted from the signal and the transformation control parameters. For example, we can take into account N features to control M parameters of the transformation, or more simply (using some sort of linear combination) take into account N features to control a single parameter of the transformation process. This way, the behavior that a given transformation will have on a particular sound is much more predictable and coherent to the characteristics of the sound itself. Yet another example of the interaction of the user in the transformation process is at the previously introduced stage of linear mapping between features and transformation control. Non-linearities, such as smoothing to avoid rapid transitions or truncation of the feature curve in order to select only the part of interest, may be introduced and directly controlled by the users input.

Obviously, the user input can be directly fed to the transformation block in order to change the parameters of the actual transformation process. The influence of the users action is now on the transformation controls (which will be generally different from those controlled by the extracted features). The following diagram illustrates the different possible user-inputs to the transformation thread.

But, as we already mentioned, when we talk about content processing, our focus is somehow shifted towards the final user of the system. The scenarios and examples of user input seen up until now suppose the user is still interacting with the transformation at a low-level. Thus, the user is seen more as an algorithm tweaking signal engineer than as a musician or artist.

But, in most cases, when we talk about content-based transformations, we imply that some sort of mapping between low-level parameters and higher-level ones is being performed. The aim of such a mapping is to group and relate features in such a way that they become meaningful for the targeted user. Still, the level of abstraction of the final controls has a lot to do with the profile of that targeted user. An expert user may require low-level, fine-tuning while a naive user will prefer high-level, easy

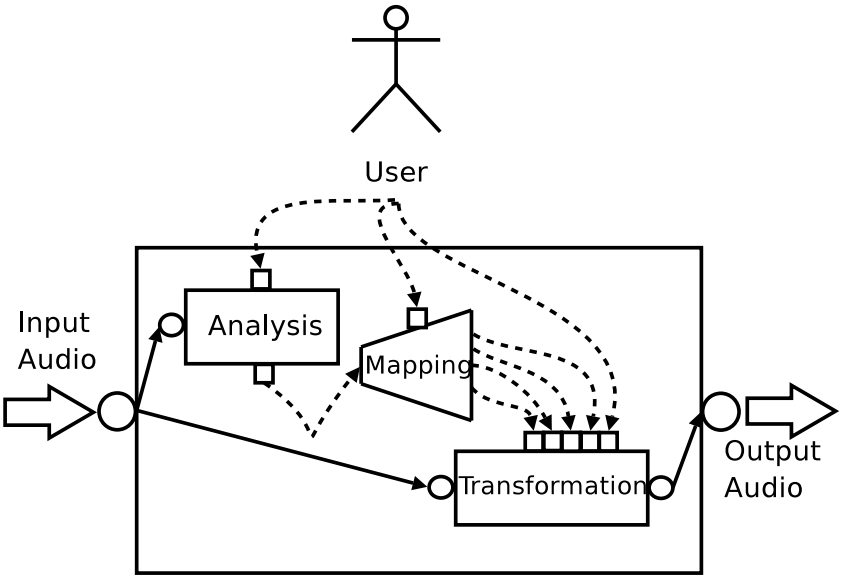

Figure 5.22: User inputs to a content-based transformation system

to grasp parameters. It is interesting to note that this mapping is the inverse process to that of the multilevel analysis process illustrated in figure 5.7.

In the simplest case, the mapping between low and high-level parameters is done at the control level. The user input is processed and mapped to the low-level parameters affected by that particular control (see figure 5.23).

But this mapping can already be performed at the analysis stage. Thus, these higher-level features are analyzed and extracted from the sound in such a way that the user can interact with them in a meaningful way (see Figure 5.24).

It is clear that the choice of a good mapping strategy is absolutely necessary if we aim at providing a user-oriented content transformation. Many studies have focused on mapping human gestures to low-level synthesis parameters (see [Rovan et al., 1997, Schoner et al., 1998, Todoroff, 2002, Wanderley and Battier, 2000], for example). Our focus here may seem different (because we are not dealing with physical gestures) but it is not so. The intention of a sound designer or musician using a transformation from a high-level approach can in many ways be seen as a musical gesture. Indeed, it is also a so-called haptic function, that is a low-frequency (compared to the frequencies in the sound signal itself) change in the control values.

The main perceptual axes in a sound are (arguably): timbre, pitch, loudness, duration, position and quality. Ideally, we are looking for transformations that can change the sound in one of its dimensions without affecting any other or combining them in a meaningful way. In [Amatriain et al., 2003], we give several examples of content-based transformations applied to all of these axes as well as transformations addressing the musical and therefore high-level content of an audio signal.

§5.4 Sample application

In the previous sections we have presented examples of applications that represent the implementation of a particular subset of the Object-Oriented Content Transmission Metamodel or some of its components. We have also a number of content-based transformations that in fact represent a minor variation over the basic scheme. But in [Amatriain and Herrera, 2001b] we presented a system to illustrate the whole OOCTM transmission chain.

The system was based on two already existing CLAM applications: SMSTools and Salto (see section 3.2.3). The basic idea of the system is to transmit monophonic musical phrases by simply transmitting an XML melody description. Figure 5.25 illustrates the main components of the system and the basic data flow.

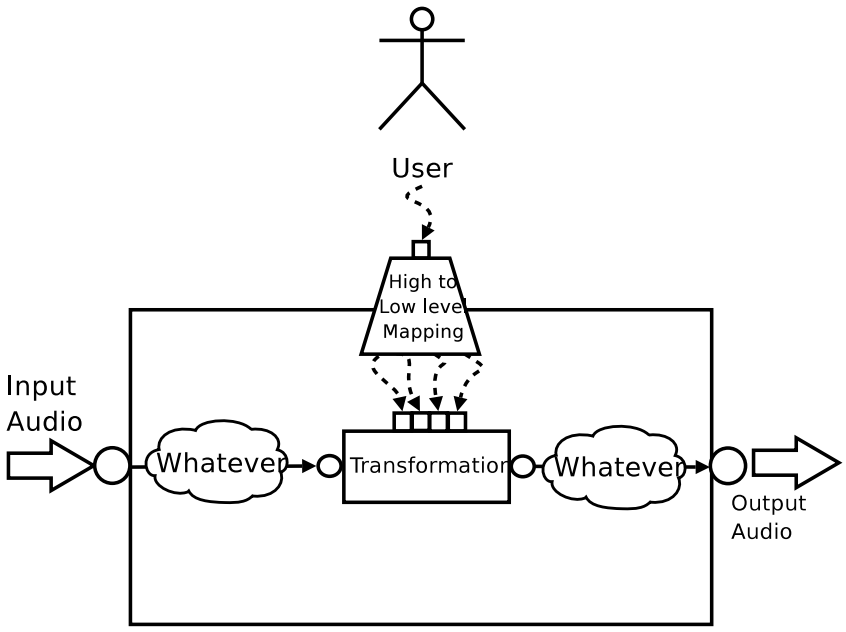

Figure 5.23: High to low-level mapping at the control level

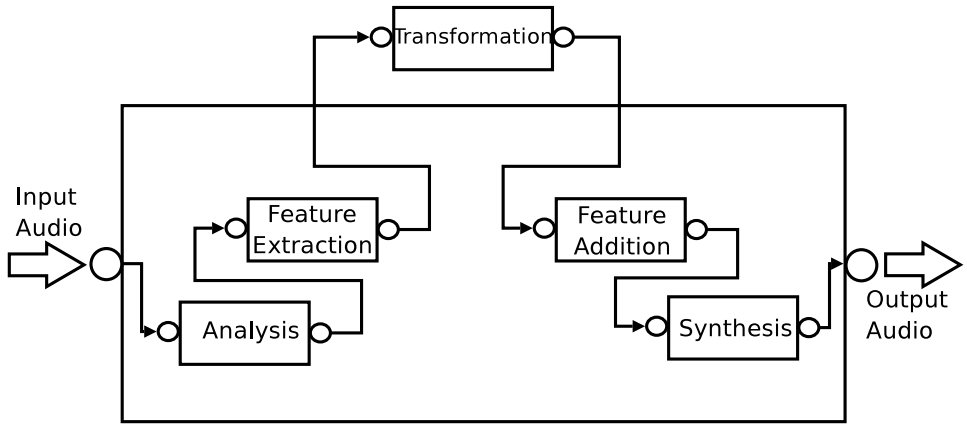

Figure 5.24: High to low-level mapping at the analysis step

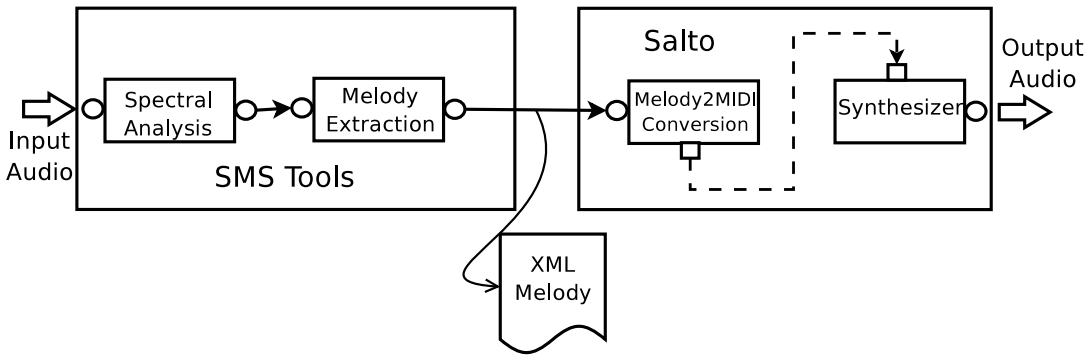

Figure 5.25: OOCTM Sample Application

Note how this block diagram represents the complete OOCTM chain. In our demo system the transmitter is basically the SMSTools application with added melody description capabilities. The receiver is the Salto saxophone synthesizer which was until then a real-time MIDI-controlled synthesizer but to which we added the capability of understanding XML melodies.

Just as it usually happens in these kinds of applications in our system we can distinguish two different phases: initialization phase and transmission phase.

In the **initialization** phase we have to construct and set up the receiver. For doing so we have to take audio samples of isolated notes, analyze them at the receiver and send them to the transmitter as SMS spectral samples. The receiver will then organize them in a local database and finally become ready to listen to incoming messages. Note that in this sample system this initialization phase is rather limited because the synthesizer that we have at the receiver is not general enough. In this case we aimed at having a reasonable quality and fidelity to the original encoded sound and that is still very difficult to achieve with a generic synthesizer. Nevertheless, and as a future line of work, it would in theory be possible to actually transmit the complete description of the instrument using an instrument description language such as MetriX's MIDL (see section 6.4.2). Then the idea would be to transmit both the spectral (or whatever other kind) samples together with an XML description of how they should be organized at the receiver.

In the **transmission** phase the transmitter receives incoming monophonic phrases and extracts the melodic description. For doing so it uses a basic segmentation algorithm for identifying note boundaries and assigns the analyzed energy and fundamental frequency to that particular region (see [Gómez et al., 2003b] or [Gómez et al., 2003a] for more details on the algorithm being used and its implementation). The melodic description is then encoded into XML, in an MPEG-7 like manner and transmitted. The receiver then reads the XML melodic description, decodes it and translates it into internal synthesis control data. It finally synthesizes the output sound.

§5.4.1 Limitations and Opportunities

The system just presented should be understood as a proof of concept. It has several limitations that make it unpractical. Just to name a few:

As already mentioned the receiver holds a particular model of a sax/trumpet synthesizer. In its current implementation the model cannot be modified or configured, just controlled. Nevertheless, as we will show in the next chapter, it is easy to foresee a solution to this issue by allowing the model to transmit complete instrument descriptions such as those allowed by the MetriX Instrument Definition

Language.

On the other hand, the encoded and transmitted melody has severe limitations. To start with, the instrument that has produced it is not encoded into the stream. Encoding the exact instrument that has produced the melody would mean having a robust and precise instrument classification algorithm. Although many people, including some from our own group (see [Herrera et al., 2000]), are working on it but this is right now far from solved.

Another limitation of the encoded melody is its limited information. As a matter of fact the only data being transmitted is the note boundaries, pitch and energy. This is in fact the same information contained in a MIDI stream. In order to have more realistic performances other information such as the energy envelope, ADSR curve or vibrato characteristics should be encoded. Most of these descriptors though can be easily computed.

Finally, and also on the transmitter side, there is a great limitation on the content analysis component. The algorithms used can only work on monophonic melodic phrases. Other more complex signals should be analyzed using other techniques such as PCA or multipitch, which are still not robust and precise enough.

Nevertheless, and regardless of all these limitations, we believe that the presented sample system already highlights many of the opportunities of the Object-Oriented Content Transmission Metamodel.

To start with, any of the components of the system, though maybe limited, is already useful by itself. We have a content analyzer, a content description XML encoding tool, a decoder that interprets XML and translates it into MIDI or information understandable by the user and a content-based synthesizer.

But the most important novelty is the use of coherent and compatible formats all through the chain. XML, or any other compatible metadata format, can be used in the analysis, transmission and synthesis step, opening up a whole range of possible future applications of the metamodel.

§5.5 Summary and Conclusions

In this chapter we have presented an object-oriented metamodel for content processing and transmission (OOCTM). This metamodel may be seen both as an extension and a particularization of the Digital Signal Processing Object-Oriented Metamodel presented in chapter 4 and presents a way of modeling signal processing applications that deal with all aspects of content-based processing such as content analysis or content-based transformations.

The metamodel is based on two conceptual foundations: on one hand we call *content* to any semantic information that is meaningful for the target user; on the other hand, and applying one of the object-oriented paradigm maximas, we state that all semantic information contained in a given signal can be modelled as a collection of related objects.

Following the traditional Shannon&Weaver model for information transmission our metamodel is divided into three main components: a *semantic transmitter*, a *channel*, and a *semantic receiver*. The semantic transmitter is in charge of performing a multilevel analysis on the signal, identifying objects and finally building a multilevel object-based content description and encoding it in an appropriate format such as XML. The channel transports this metadata description and any added noise will not be considered as such unless the original meaning is modified. Finally the semantic receiver receives the multilevel content description, decodes it and translates it into a synthesizer-readable format. The synthesizer included in the receiver then synthesizes the output signal.

It is important to note that we are not so much interested in the fidelity of the final synthesized signal to the original but rather on whether the original "meaning" is preserved and is useful for the final user.

The Object-Oriented Content Transmission Metamodel can be seen as an extension of the classical Shannon&Weaver model for information transmission. The metamodel should be understood as a conceptual framework that can be instantiated to build working applications. In this sense it demonstrates how new technological advances have brought up the opportunity of redefining a communication model that is more than 50 years old. The metamodel is very much related to the Structured Audio metamodel and can also be seen as a step beyond parametric encoding. Finally if we add a transformation function to the channel we end-up having a general scheme for content-based transformations.

Anyway one may question the benefits of content-based audio applications. It is our opinion that by concentrating on the transmission of content description we are actually favoring the distinction between content and its realization. And, by doing so we favor a higher level approach, encapsulation, concept reuse, the upcoming of new applications (i.e. content-based transformations), data reduction, and robustness enhancement, to name a few. In this sense in the current chapter we have also given several examples of applications that represent particular instances of the metamodel or subparts and one in particular that instantiates the whole metamodel in order to transmit and synthesize a previously analyzed and extracted musical melody.

As a conclusion we may state that we have presented a new metamodel for information transmission that does care about meaning and content. The Object-Oriented Content Transmission uses a fully object-oriented view and an interpretation of content as any meaningful information for the user

in order to present a conceptual framework that can give rise to many new and interesting applications. We have shown in practice some sample applications and, although still embrionary, they demonstrate the practicality of the metamodel.

CHAPTER 6

An Object-Oriented Music Model

In chapter 2, and more precisely in sections 2.4 and 2.6, we reviewed different music frameworks and languages. Each of them presented, more or less explicitly, a model of the music domain.

In this chapter we will present an object-oriented music model. Departing from the basic DSPOOM metamodel presented in chapter 4 we will create a particular instance of the metamodel for representing musical information. It is important to note, though, that the scope of our music model is much narrower. We are not trying to build a general metamodel useful in all situations but rather a particular model to prove the usefulness of object-orientation in general and the DSPOOM metamodel in particular in the music domain. Nevertheless, it is important to note that this model shares many features with others presented in chapter 2, particularly the one implemented in the NeXT Music Kit (see section 2.4.1).

Just as in the previous chapter we stated that any entity in an audio system is a *sound object*, we will now consider that any entity in a music system is a *music object*. A music object cannot sound or be heard unless it is somehow converted into a sound object. In order to generate a particular sound object usually several musical objects have to interact and finally undergo a synthesis process. In that sense our idea of music/sound object is somehow similar to that of Kyma's, presented in section 2.5.1.

The music domain is made of many different classes whose objects finally influence the performance. Nevertheless, and as a first approach, our music model distinguishes the following classes: Instrument, Generator, Note and Score.

It is important to note that our music model is object-oriented and not "event-oriented". By event-oriented we are referring to models such as Siren/MODE (see section 2.4.2) where, although objects are used, the key and central modeling concept is the *event*. Usually, music event-oriented models center the whole model around the concept of *Note*, and a Note is modeled as a particular kind of event. An event is in that case defined as an object that has a duration and possibly other properties

(see [Pope, 1998b] or [Pope, 1991c], for example).

In our object-oriented metamodel the word "event" is understood in the systems engineering sense. According to Law and Kelton "An event is an instantaneous occurrence that may change the state of the system" [Law and Kelton, 2000]. Events are directly related to object-oriented messages and messages are sent to objects in order to modify their state. Events have no duration, as far as we are concerned they are instantaneous occurrences. For that reason, it is clear that in our model Notes are objects and not events. An event may be the message sent to a Note in order to change its tuning or to make it stop. Nevertheless, and using a loose variation of the Command pattern (see [Gamma et al., 1995]) we can interpret a message, event or command as an object. We will further develop the model in the next sections.

§6.1 Instruments and Generators

A *musical system* can be seen as a set of musical objects called *Instruments* that collaborate for a common goal or *performance*. Therefore, at the first level, the most important objects in this kind of system are the Instruments. Instrument objects are instances of the last level of a class hierarchy that has the abstract Instrument class as the base class that is then specialized into different families of Instruments such as string, percussion or wind. In the last level of the hierarchy we find particular Instruments such as piano or guitar, see Figure 6.1.

An Instrument responds to incoming events by generating music. These events are generated from the interaction of the performers (system actors) on the Instrument interface.

But if we look closer into an Instrument we discover that Instruments are not the atomic physical objects in a musical system. As a matter of fact, an Instrument is in itself a set of more or less independent elements that produce sound by themselves. A clear and extreme example of this are the drums, where a single Instrument is in fact the result of many different possible combinations of individual Instruments.

When focusing on the properties of the internal mechanism, a first classification could divide Instruments into monophonic and polyphonic. In Instruments belonging to the first category it is clear that no more than a single Note can be sounding at the same time. But even polyphonic Instruments have limited amount of voices (simultaneous sounding notes). For instance, in a guitar no more than six simultaneous notes can be produced.

We define a *Generator* as the atomic entity that can generate a sound by itself. A Generator usually generates sound in response to an incoming control event, therefore Generators must have the

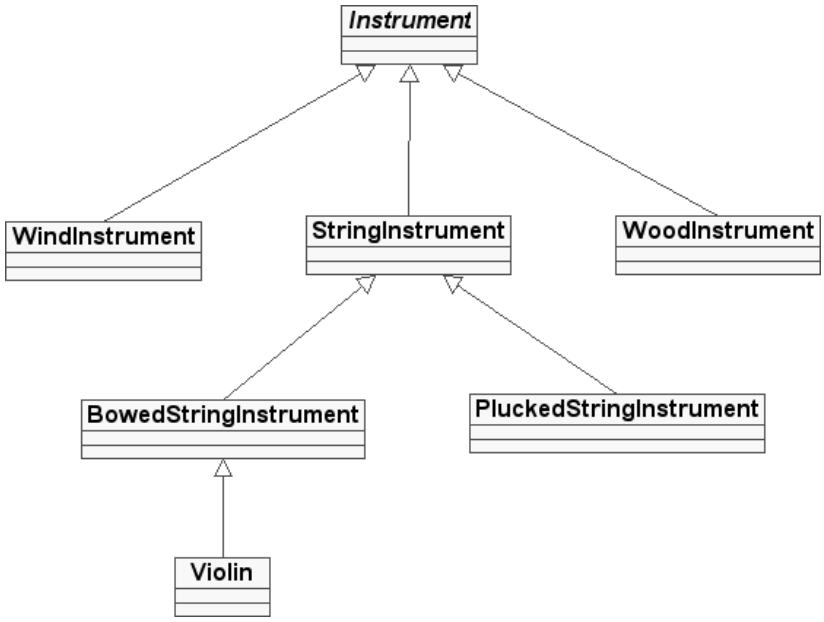

Figure 6.1: An Instrument Class Hierarchy

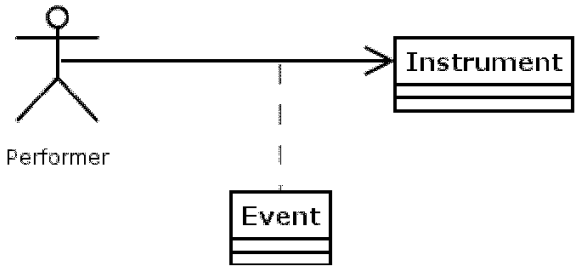

Figure 6.2: Performer, instrument and events

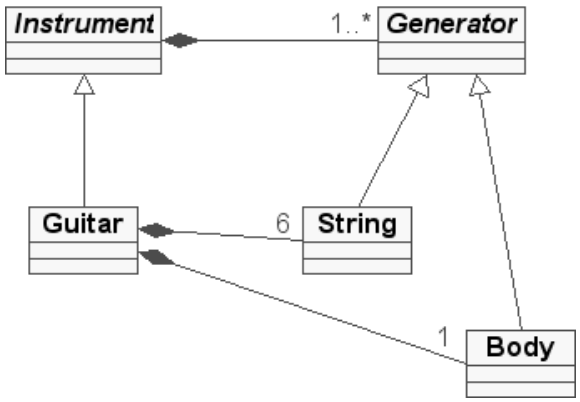

Figure 6.3: Instrument and Generators Class Diagram

ability of receiving and interpreting such events. A Generator is part of an Instrument although there can be Instruments with just a single single Generator (e.g. a flute). An Instrument is therefore a set of Generators that can be controlled independently but always according to the rules dictated by the Instrument definition.

The relation between the Instrument and Generator classes is modeled at an abstract level and must be made concrete when modeling a particular Instrument. If, for example, we have to define a standard acoustic guitar we may describe it as having six Generators, one for each of its six strings.

The guitar Instrument itself is in charge of defining pitch range and other important properties for each of its Generators. On the other hand it may be interesting to define a seventh Generator: the guitar body. The guitar body can indeed be seen as a percussive Generator in itself. This Generator can be controlled independently from the others and responds to completely different behavior rules. Therefore both `String` and `Body` would be subclasses of the abstract `Generator` class.

In a piano it is fairly intuitive to define each key as a different Generator. But one may be tempted to discuss that a piano cannot naturally have all the keys sound at the same time. As a matter of fact the number of simultaneous sounding notes is limited by the number of fingers or hands that are playing. Then, why not say that the actual Generators in a piano are the player's fingers? It is important to remember that according to our object-oriented model Generators are related only to the production mechanism in the system and not to the way controls from the outside actor arrive to them. A piano Generators can be excited by two or four hands but also with a foot or even an automatic mechanism. In other words, all keys in a piano can sound simultaneously and that is enough to prove that each of them is an "independent" Generator.

Therefore, an Instrument can be considered as a system made up of almost autonomous units or objects called Generators. The `Instrument` class is in charge of instantiating each Generator and

assigning it a particular behavior which is in fact a subset of the Instrument behavior in itself. But the Instrument also has supra-Generator behavior that can be used, for example, for describing the way that different Generators will interactuate in between them. For instance an Instrument may have a constraint that if Generator N is active, Generator N+1 cannot become active.

From the description given up to now, it is clear that some sort of relation exists between the music model that we are starting to define and the DSPOOM metamodel presented in chapter 4. Figure 6.4 illustrates the interpretation of an Instrument as a particular instance of the DSPOOM metamodel.

The figure illustrates the DSPOOM modeling of a 4-Generator Instrument. Note that we model an Instrument as a DSPOOM Processing Composite (see 4.1.3) because in a general situation we do not need the Instrument definition to be dynamic, in which case we would need a Network. In most implementations we will build the Instrument in configuration time when reading an Instrument definition file (see 6.4.2) that is in fact a DSPOOM configuration. On the other hand, Generators are regular DSPOOM Processing objects.

The Instrument is a DSPOOM Generating Processing object as it has output ports but no input ports. The Instrument responds to incoming controls by producing an output signal through one of its output ports. These Outports correspond directly to the Outports in each of the Generators. In a similar way, there is a direct mapping between the input controls published by the Instrument and those in the Generators.

Note that the Instrument that we have described so far is the simplest one, made of only Generators. Nevertheless, as in a regular DSPOOM model, we could integrate non-generating Processing objects as secondary Generators. Think for instance on an Instrument that adds a filter for each Generator. This way we are able to use the metamodel for constructing a modular synthesizer or similar applications.

§6.2 Notes

When an Instrument Generator receives an event or message it responds to them by either changing its internal state or by "activating" special objects that also contribute to the Generator state. These objects are called *Notes*. A musical Note is defined as a sound object that has a precise and explicit active lifespan (i.e. the note duration), a certain loudness, an optional pitch and other optional attributes. The pitch, just as the other attributes in a Note, is not constant and can vary in run-time as the Note object responds to an incoming message. Therefore, an incoming message will produce a change in the Note internal state by modifying any of its attributes. A special message, because of its

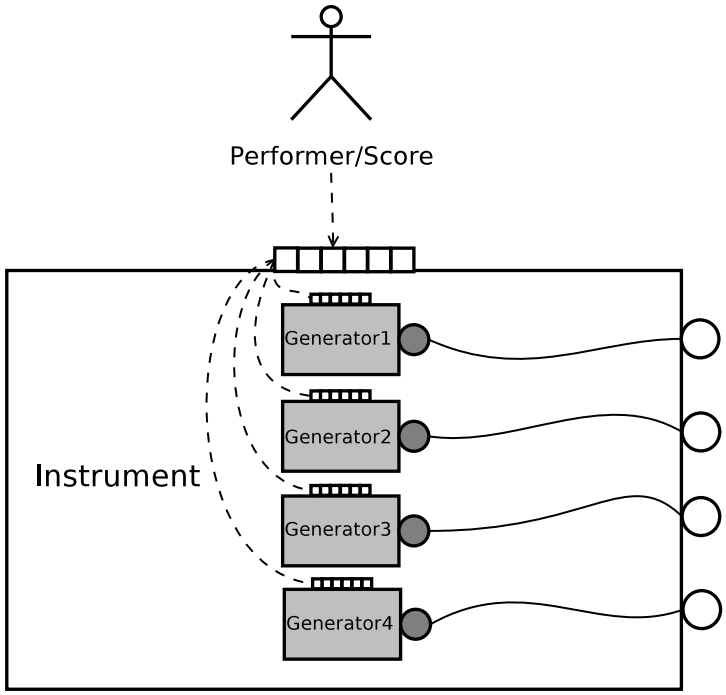

Figure 6.4: Instrument as a DSPOOM Composite

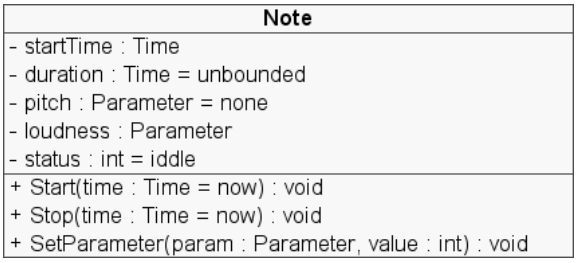

Figure 6.5: The Note class

consequences is the "Stop" message. This will cause the Note object to transition to an inactive state but will generally, and as it will be seen in next paragraphs, not imply the object destruction.

A Note object is associated to its owner Generator and will only respond to messages sent through it. A Generator is by nature monophonic, this meaning that it can only hold a single sounding note at a certain moment. For this reason when a Generator is created it is granted an associated inactive Note. This Note will become active whenever the Generator receives a "play note" message from its Instrument.

It is important to understand, though, that the fact that a Generator has a single Note object through all its lifespan is just for convenience and could conceptually be understood of a set of Notes that are instantiated and de-instantiated every time they start/stop sounding. In this sense the concept of Note that we are here using is perfectly compatible with the Note obtained in the analysis phase of an OOCTM metamodel (see section 5.2.1.

Ranges of valid attributes for a particular Note, such as its minimum and maximum pitch, are handled by its Generator. Clearly the Note object is completely transparent to the rest of the musical system that will only treat with messages sent to Instruments and particular Generators present in it.

Our model of Note is very different from that of event-oriented music models such as Music-N where a Note is considered an event. As a matter of fact our music model also includes the idea of *Note event* that will be explained in next section. But it is important to note that this consideration of a Note as something completely different from an event is not exclusively ours. As Miller Puckette explains in [Puckette, 1991b] already in RTSKED (see [Mathews and Pasquale, 1981]) a note was considered as a *process* and this same idea was pushed forward in FTS and Max. The combination of our Generator/Note structure is also very similar to the note in ZIPI (see section 2.6.2.1).

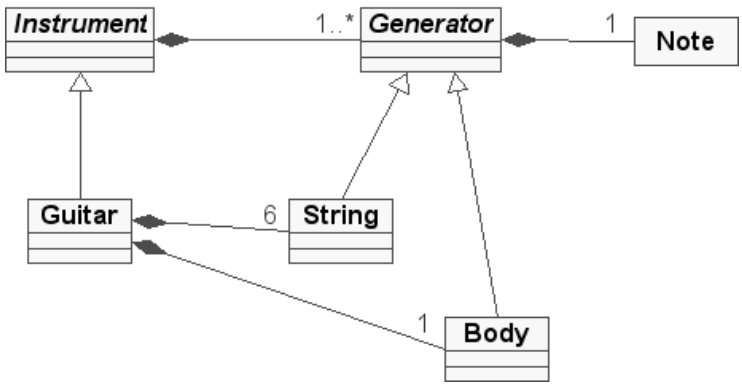

Figure 6.6: Instrument, Generators and Notes Class Diagram

§6.3 Songs and scores

From the previous definition of note it is derived that a Song is defined as the state chart of all Notes present in a musical system (in the case of the conceptually equivalent model of Notes being born and dying on receiving the start/stop message this would rather correspond to the state chart of all the Generators). This definition is very different from its classical equivalent in which a song or musical piece is directly related to its score.

In the object-oriented music model here presented a musical Score must be understood as the sequence of events or messages that will be sent to the different objects present in the musical system. These events, as we already illustrated in figure 6.2, are the result of a supposed interaction between a Performer and an Instrument. As a matter of fact, an event can be sent to a musical Instrument or directly to any of its Generators and as a result it modifies its internal state.

Two different kind of events can be distinguished. Note events are those that through an Instrument or a Generator modify a Note as defined in the previous section. And Global Control Events are those that modify the whole musical system setting parameters such as tempo or key.

A Note Event is made of a *Time Statement* that specifies when the event is to occur, a *Variable* that indicates what object (Instrument, Generator...) will receive the event, and any number of pairs Parameter/Parameter Value, where a *Parameter* specifies what attribute of the object will be modified, and a *Parameter Value* contains the new value for the given parameter.

Note that, as illustrated in figure 6.7, Note Events are in fact also objects. This fact makes our model somehow similar in this sense to the event-oriented models such as Siren (see section 2.4.2) from which we have from the very beginning established a clear difference. We also acknowledge the fact that a purely object-oriented model is not incompatible with event-driven behavior. This same

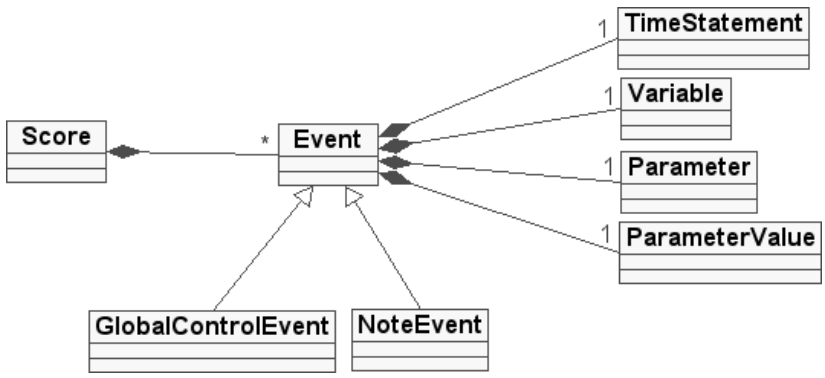

Figure 6.7: Score Class Diagram

conclusion was drawn by the authors of Kyma when they added an event language in version 4.5 (see [Scaletti, 2002]).

An example of a score in pseudo-code would therefore be:

```
At time=0 Note.start at InstrumentX.GeneratorN
At time=1 Note.pitch(A3) at InstrumentX.GeneratorN
At time=2 Tempo(120)
```

The first line is a note event that indicates that Note at GeneratorN in InstrumentX should start sounding. Note that as no other parameters are set these would be taken as default. In the second line we send another Note event modifying the existing pitch for the previously started Note. Finally in the third line we send a global parameter change that will affect the whole score.

It is interesting to note that, as S. T. Pope points out in [Pope, 2004], most software synthesis systems have some sort of note statement consisting of a *keyword*, the *start time* and *duration* of the event, the *Instrument* and the *parameters* of the Instrument. In our model the variable is equivalent to the keyword, the time statement to the start time and the parameter/parameter value pairs to the generic parameters. But there are some differences. A first important difference is that the note event does not necessarily have to address an Instrument as it can address a particular Generator or a whole Instrument section or Instrument. Another important difference is that we do not require a note event to include a duration and this is treated as another regular parameter that may or may not be specified in a note event. This is in fact one of the main limitations of the Music-N model reported by different authors. Miller Puckette, for instance, states in [Puckette, 1991b] that the Music-N model is not well suited to situations in which some aspects of a sound are not defined at its beginning and he adds that those are in fact most of the interesting situations.

§6.4 MetriX

MetriX was designed in order to have a working synthesis system that implemented the model presented in the previous section [Amatriain et al., 1998]. MetriX was at first designed as a textual-based language and then ported to XML (see 6.4.4).

This language, together with a related C++ library conform the MetriX framework. This C++ library was originally written from scratch and just reusing some existing code for the actual synthesis of the sound. Now it has been integrated into the CLAM framework The main goal in MetriX is to offer a powerful and intuitive interface to control all aspects involved in the creation of a synthetic musical piece. For doing so we rely on our object-oriented music model.

The MetriX language is in fact made up of two sub-languages: a MetriX Score Definition Language (MSDL) and a MetriX Instrument Definition Language (MIDL). Therefore, following the tradition of Music-N Languages (see section 2.6.1) such as CSound (see 2.6.1.2), MetriX makes a distinction between the process of defining and creating an Instrument and the process of controlling this Instrument through a music score. The original MetriX language was implemented in textual format and later evolved into XML to become more object-oriented.

Figure 6.8 reproduces the classification map already used in chapters 2 and 3 but now including MetriX in the position that best explains its scope.

MetriX can be considered in some sense a Music-N language (see section 2.6.1): our concept of *Generator* is very much related, although not strictly equivalent, to Music-N's *unit generator* and we also have a clear distinction between the *score* and the *Instrument definition*. Nevertheless, we have already seen in the previous section how our object-oriented music model introduces many differences with classical Music-N models. We will illustrate more differences in the following sections. Because of this, we could label our model as "Enhanced Object-Oriented Music-N model".

§6.4.1 Basics in MetriX

Although, as already commented, MetriX implements the music model presented in the previous sections some specific issues need to be highlighted for understanding the two languages that will be introduced in the following sections.

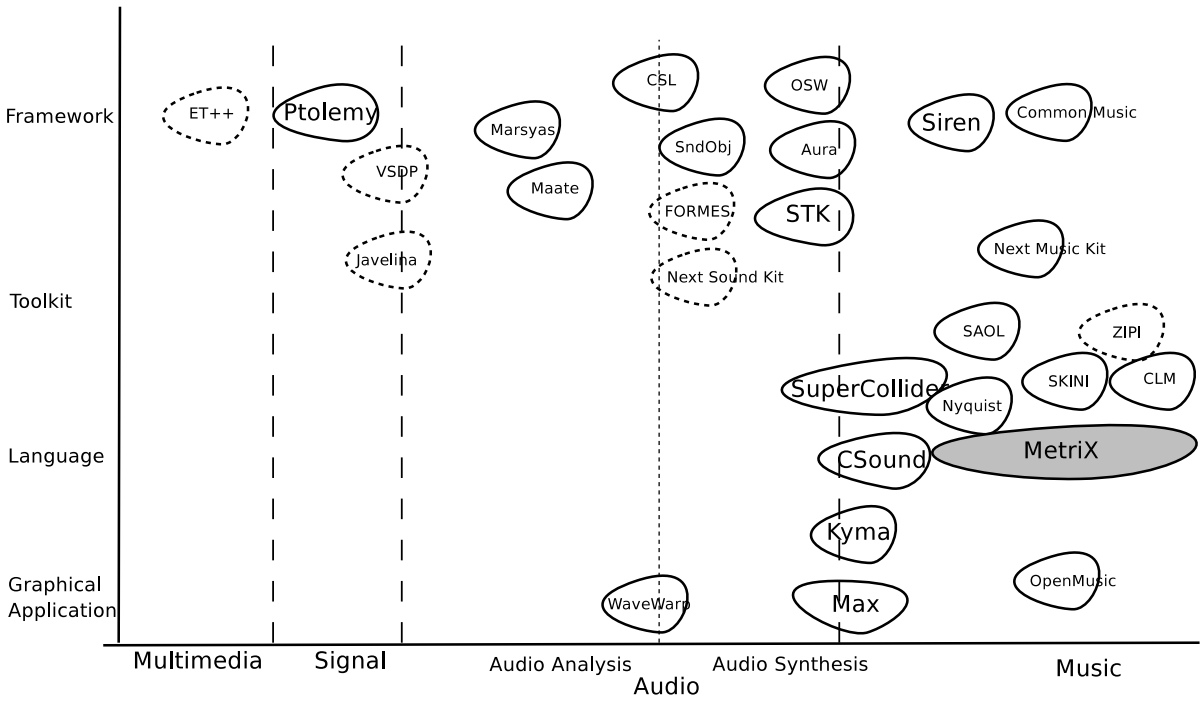

Figure 6.8: MetriX classification in respect to other environments

§6.4.1.1 Instruments and Generators

An Instrument is the class that holds behavior and state for a system made up of a set of Generators and the relations that are established in between them. When we define a new Instrument in MetriX we are actually subclassing the base (and abstract) `Instrument` class in order to implement behavior that is particular to the Instrument being defined. This subclassifying though can be done at the object level by instantiating an object and using some attributes to set its behavior (It is well known that instantiation is as a matter of fact a form of subclassification, see [Graham, 1991] for instance).

Defining an Instrument behavior means specifying how the Instrument will respond to incoming messages. For doing so we must define what Generators make up the Instrument and how each of them responds to input messages.

In order to produce sound an Instrument needs audio generation primitives that can be existing audio waves, analysis results or a set of simple functions such as oscillators or waveform Generators. There is no need to have one of such primitives for every Generator in the Instrument, only those needed to represent the space of possible sound combinations to be produced by the different Generators.

Therefore, the primitives in MetriX are arranged in what we call the *Timbre Space*. The idea of a timbre space is not new (see [Wessel, 1979], for example) but it has specific properties in our framework. A MetriX timbre space is an n-dimensional space constructed by placing the audio generation primitives. Each of the dimensions in the space represents a key feature of the sound such as Pitch or Loudness[1]. The decisions as to how many and which key features are necessary for a given Instrument is dependent on the flexibility and quality the sound designer wants the Instrument to have. As it will be seen later, axes not represented in the timbre space but needed in the synthesis can also be obtained by transformation.

The n-dimensional geometric space is constructed by placing the *samples* or primitives in precise coordinates belonging to this space. Samples located at furthest positions in relation to a given dimension define the limits of the timbre space. Once existing samples are located at definite positions, intermediate samples can be then obtained by interpolating neighboring ones.

A first simple example of a timbre space could be a one dimensional space formed by placing samples on a single axis, sorted by pitch. Notes with intermediate pitch will then be obtained by interpolating the two neighboring samples with the appropriate weight factor.

Adding some complexity to the example we could add a second dimension to our timbre space: loudness. Figure 6.9 illustrates a two-dimensional timbre space. Note that only A's in two different loudness (fortissimo and pianissimo) have been sampled from the original Instrument. Intermediate

[1] This refers to the regular use of the timbre space. More "creative" or exploratory strategies may also be followed.

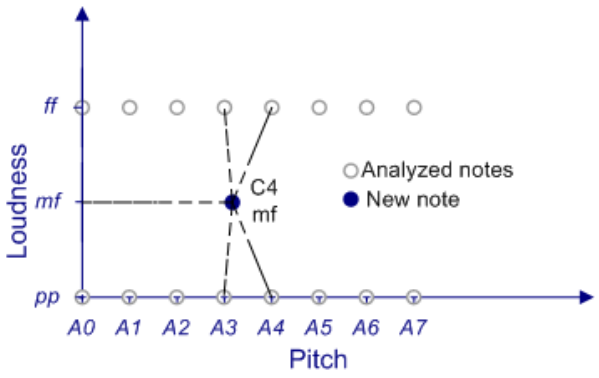

Figure 6.9: Two dimensional timbre space

pitches and loudness will be obtained by interpolating existing samples.

§6.4.1.2 Mappings, Parameters and breakpoint functions

When defining an Instrument, apart from constructing the space of Generators and pre-existing samples, the most important thing is to define what events the Instrument may acknowledge and how it can respond to them. We distinguish three kinds of events or parameters: low-level parameters, timbre space parameters and high-level parameters.

The first kind of parameters address low level features in the sound like its amplitude or fundamental frequency. These parameters represent transformations can be more or less immediately applied to the sounding note using regular signal processing techniques.

The timbre space parameters are indeed a specific kind of low-level parameters that affect one of the dimensions integrated into the defined timbre space. A variation in the parameter just represents a new coordinate in the space and therefore a new interpolation between the existing samples.

Finally, the high-level parameters represent concepts that cannot be obtained by an immediate transformation or that have been assigned to one o the timbre space dimensions. Usually, a variation in a high-level parameter represents a variation in several low-level parameters and even in the timbre space coordinate. For defining those relations we need ways of describing mapping strategies.

In MetriX, for describing mappings as well as transformation there is a simple language tool that allows to define break point functions and refer to things like the number of the Generator, or the value of the parameter. We will comment more details on these matters in next section.

Finally it is interesting to note that this distinction between low-level and high-level parameters in the synthesis process is very similar to the distinction between low-level and high-level analysis parameters or *descriptors* that we explained in the OOCTM (see section 5.2.1).

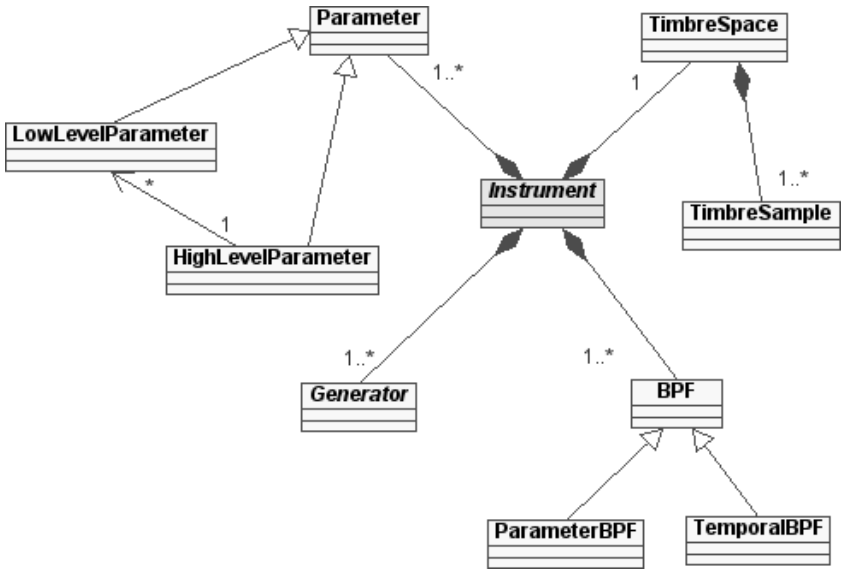

Figure 6.10: MetriX Instrument Class

§6.4.2 MIDL: The MetriX Instrument Definition Language

The MetriX Instrument Definition Language, MIDL for short, is a tagged textual language used for defining an Instrument that will then be used from a MetriX score. Although the language is intended to be general purpose, it has only been tested in the context of SMS-based synthesis (see B). Note also that because of the integration of MetriX into the CLAM framework the MIDL can be easily extended if anything else is required for a different synthesis technique. In this sense the MIDL can be seen as a default implementation of a MetriX Instrument out of which new ones can then be extended.

An Instrument Definition File is divided into four different sections: definition of Instrument Generators, Timbre Space, Temporal and Parameter Break Point Functions, and high and low-level control Parameters.

§6.4.2.1 Generators

In this part of the Instrument File, a unique name and integer identification number must be given to each Generator.

The identification number will be used in the other parts of the Instrument File for applying mathematical operations based on the Generator number. The name will be accessed later from the Score File.

§6.4.2.2 Timbre Space

The definition of a Timbre Space with the MIDL consists on the definition of three different aspects.

First, the number of dimensions to be used. The Instrument designer must decide what features of the Instrument represent a substantial change in the sound that cannot be achieved using the different transformations available. These dimensions such as loudness, pitch, articulation, etc... must have the corresponding SMS data extracted from a previous analysis of those features in the Instrument. A compromise between sound quality and amount of synthesizer memory used must be adopted.

Next, the kind of Interpolation to use between the Data stored. A set of standard interpolation types are available.

Finally, the positioning of each sample in a concrete location in the space must be given. The `Instrument` Class will be capable of solving intermediate positions by the interpolation of the SMS data loaded from the files.

E.g. A good quality piano sound can be obtained by storing the SMS Data from just eight of the piano keys (A0 thru A7) and obtaining the rest by interpolation. Thus, only one dimension is used

(pitch). Other features such as loudness can be obtained by applying different transformations on the data available [Solà, 1997].

§6.4.2.3 Break-point Functions

Two kind of break-point functions are available: Temporal Break-point functions and Parameter Break-point functions. A MetriX break-point function is defined by stating a name, an interpolation type and any number of points from which the others will be computed.

A Temporal Break-point Function is a user defined function that returns a single value according to the relative time elapsed since the beginning of an event.

And unlike Temporal Break-point Functions, Parameter Break-point Functions return a complete envelope that will be applied to the parameter involved according to its definition.

§6.4.2.4 Parameters

In this part of the score, low-level and high-level control parameters are initialized. Only these parameters will be accessible from within the related Score File once this particular Instrument is instantiated.

To initialize a low-level control parameter only its range (maximum and minimum value) and default value must be specified. The specific synthesis engine shall be in charge of interpreting this low-level parameters and apply low-level transformations on the synthesis data accordingly. Although these parameters are not exclusive nor limited to a particular synthesis scheme they have only been implemented, as a proof of concept, to control an SMS-based synthesizer.

A high-level control parameter must also include its range and default value in its definition. But besides that, it must also specify the way it relates to low level parameters or to the timbre space location, a relation that is usually referred to as *mapping function*. A single high-level control parameter may influence any number of low-level parameters. For defining the high-to-low level mapping all standard formulas as well as the previously defined break-point-functions may be used.

MetriX includes a set of standard low-level and high-level control parameters but the list is dynamical and can be enhanced for a particular case.

In [Scaletti and Johnson, 1988] the authors argue that Music-N makes it very difficult to control things at a higher or lower level. With this separation between low and high-level parameters that can then be controlled from the score we are indeed giving a solution to this problem. It is also important to acknowledge that this clear separation between low and high-level parameters was somehow inspired by that in ZIPI (see section 2.6.2.1).

§6.4.2.5 Example

In the following example, a simple SMS based piano synthesizer is implemented. Only one dimension of the timbre space is used and the number of control parameters used have been reduced to the minimum to keep the example simple.

Note: The reserved characters '#' and '@' mean the value of the parameter and the number of the Generator involved respectively; a line starting with '/' is a comment that will therefore not be interpreted; "T" refers to the default duration of an existing sample.

```
/First we define 86 generators named Key0 to Key85
Generators :{"Name(0-85),Key" }
/Now we define the one-dimensional Timbre Space
TimbreSpace :{
        /We define a single dimension
        "nCoord,1"
        /We set the interpolation type to linear
        "IntType,Linear "
        /And we place the eight A's previously analyzed and stored in SDIF format at equidistant
        points
        "c:\Metrix\Piano\A0.sdif,0 "
         "c:\Metrix\Piano\A1.sdif,0.1412 "
         "c:\Metrix\Piano\A2.sdif,0.2824 "
         "c:\Metrix\Piano\A3.sdif,0.4235 "
         "c:\Metrix\Piano\A4.sdif,0.5765 "
         "c:\Metrix\Piano\A5.sdif,0.7176 "
         "c:\Metrix\Piano\A6.sdif,0.8588 "
         "c:\Metrix\Piano\A7.sdif,1 " }
/We now define a Parameter break-point function.  Note that this bpf applies a low-pass filter
/depending on the value of a given parameter.
ParamBPF :{"PianoLPF,Linear,(0,1)(0.5,0.39*# +0.5)(1,0.00006*# ^2)" }
/And now a Temporal break-point function that implements a fade out.
TimeBPF :{"FadeOut,Linear,(0,1)(T,1)(T+0.3,0.1)(T+0.6,0)" }
/We will only use three low-level parameters:  the overall gain, and the gain to be applied to
/the sinusoidal or residual components
LLParams :{
        "Gain,0,0.2,0.1"
        "SineGain,0,1,1"
        "ResGain,0,1,1" }
/With three high-level parameters (Key Velocity, Key Number and Pitch) we control both the
/low-level parameters and the timbre space location
HLParams :{
        "KeyVelocity,0,127,64,
        (Gain,#/127*0.2*TimeBPF(FadeOut))
        (SineGain,ParamBPF(PianoLPF))
        (ResGain,0.0000506*#^2+0.00145*#)"
        "KeyNumber(@),@,@,@,(TimbreSpace,(@/85))"
        "Pitch(@),28.8316*1.0595^@,27.16*1.0595^@,28*1.0595^@ , (TimbreSpace,(@/85))" }
        /Note that KeyNumber and Pitch are alternative ways of controlling the same feature
```

§6.4.3 MSDL: The MetriX Score Definition Language

The MSDL is a text-based synthesis control language which takes part of its features from previously released languages as the NEXT ScoreFile Language (see 2.6.2.3) or SKINI (see 2.6.2.2). No low-level packed messages are involved in defining a Score with the MSDL. With a quick look at an MSDL Score File any musician can get a grasp of what is going on.

Although no exact match with the MIDL syntax is meant, similarity and compatibility is intended.

The Score is made up of two different parts, which will be discussed in the following sections: the Header and the Body.

§6.4.3.1 The Score Header

The Score Header is where all the global variables relative to the score information or to the output sound are defined. Concepts such as Tempo, Beat, Output Sound File or Sample Rate must be included in this part or either will be assumed as default.

Another feature included in the Header is the definition of all the Instruments to be used in the score. A reference to the Instrument File location must therefore be included.

And last, all kind of user variables can be initialized in this part of the Score. The user can define an unlimited set of variables in order to access the Instruments, Generators or even groups of Instruments. Note that if more than one Instrument of the same kind is to be used, its Definition File will only be loaded once and can then be referenced by the use of user defined variables such as piano1, piano2...

§6.4.3.2 The Score Body

The Body is the part of the Score where the actual musical information to control a digital Instrument is included. It is made up of a list of events; sorted by the time they take place in order to keep real-time compatibility.

An event is a group of words that define a message sent to the synthesizer controller. The standard event statement is made up of four statements sorted this way: T V P:PV, where T is Time Statement, V is Variable Statement, P is Parameter Statement, and PV is Parameter Value Statement.

These, together, conform a message that means: Modify Parameter (P) referring to variable (V) according to its new value (PV) at the moment specified (T).

The Time Statement includes features such as the possibility to use standard time, SMPTE timecode or musical Beat notation and relative increments.

Variable statements can refer to user defined variables or directly to Instruments or Generators initialized.

There is a clear relation between the structure of a MetriX statement and an object-oriented message. The Variable is the receiving object, the Parameter is the message selector or operation that the receptor should execute and the Parameter Value is simply the arguments passed in the message. As already mentioned, a musical score can be seen as the set of messages that are sent to musical objects during a performance.

§6.4.3.3 Example

This example shows the main possibilities of using the MSDL for controlling a synthesis process. Note the different kind of Instruments used: the first two have already been defined and included in the synthesizer standard bank, the next two are loaded from an Instrument Definition File during run-time, and the last one is a single SDIF File containing any kind of information accepted by the SDIF File Format [Schwarz and Wright, 2000].

```
/We initialize global information related to the score such as tempo or meter (note that these
/parameters are just initial values as they can be later on changed from within the score body.
Score_Info{
        Tempo:130/2
        Meter:3/4
        Resolution:24 }
/We initialize information related to the kind of output sound we want
Sound_Info{
        SampleRate:44100
        Bits:8 }
/We instantiate 5 instruments:  the first two are already in the standard bank, the next two are
/loaded from their instrument definition file located elsewhere
Instrument_Info{
        Piano
        guitar
        oboe(InsDef:"c:\score\oboe")
        violin(InsDef:"c:\mpegscore\violin")
        clarinet(SDIFDef:"c:\SMS\clarinet.sdif")}
/We now declare some variables to be able to refer to instruments and generators in a much more
/convenient way
Def instrument a=piano
Def instrument c=violin
Def generator nvar=10 c=c.string
Def instrument d=violin
Def generator a1=a.Key0
Def generator a2=a.Key1
Def instrument n=clarinet

/We begin score body
begin

/We send three messages to the a1 object (which happens to be the first key in the piano).
/The message is sent in the second quarter-note bar one
#01:01:02.04 a1 Pitch:C#3 Loudness:mf  Duration:t00:00:01
/We send a low-level control message to clarinet in second 1
t01             clarinet                Gain:(0(1)1(0.5))
/Four seconds after the last message we send a new message to Key2 in the piano
+t04             piano.key2      Pitch:f2
/We now send a global parameter and change the tempo
t04             Score_Info:     Tempo:140
t05             a1                      Loudness:ffff

end
```

§6.4.4 MetriX in XML= MetriXML

The MetriX framework was ported to CLAM. This meant on one hand fitting the MetriX object-oriented music model to the DSPOOM metamodel. As it will be seen in this section, and has already been suggested earlier on in this same chapter, this adaption was completely natural. On the other hand, adapting MetriX to the CLAM framework implied gaining XML representation "for free".

Therefore, the substitution of the textual format for XML was also natural. The overall process gave place to MetriXML.

The first thing to do when adapting a previously existing model to the DSPOOM metamodel is to identify what metaclass each of the model elements belong to. In our case the basic model classes that should find a DSPOOM metaclass are: `Instrument`, `Generator`, `Note`, `Event`, and `Score`.

The relation of the first two classes with the DSPOOM metamodel are as it was already illustrated in figure 6.4: an `Instrument` is a ProcessingComposite made up of Processing objects that are instances of the `Generator` subclass. Then the question is how the Instrument definition expressed in MIDL (see section 6.4.2) is related to the metamodel. The answer is that an Instrument definition is a DSPOOM `Configuration` that will be used to configure the `Instrument` Processing object before its execution. Note that this configuration will also be in charge of "subclassifying" the generic Instrument to a particular one as shown in figure 6.1.

The other subelements present in an Instrument such as the Timbre Space or the Break Point Functions (see figure 6.10) will have no direct interpretation in the DSPOOM metamodel. On execution time they play a secondary role as auxiliary mechanisms of the Instrument and on configuration time they will already be represented by fields in the overall configuration.

In a similar way, the `Note` class is seen as a conceptual shortcut with no direct interpretation in the DSPOOM metamodel. As a matter of fact a Note corresponds to the internal state of each of the Generators. As such we should be able to query a Generator for its Note-state, but no more direct access nor representation is needed.

It is clear that a music `Event` should somehow relate to a DSPOOM Control. But the representation of an Event in a Score, which includes for instance a time tag, is too complex to make it directly compatible with the simple asynchronous control mechanism based on simple data types that is included in the DSPOOM metamodel. The solution is the same that is usually recommended when this situation arises in modeling a particular system in DSPOOM (see section 4.1.1.4). First, a related Processing Data class must be defined. In our case, the Event class will therefore be a Processing Data. Then we need a special sink Processing object class that receives this incoming Processing Data and converts it into DSPOOM control events. In our case we will call this class the `Scheduler`.

The `Scheduler` is the class responsible for receiving input Events as Processing Data, enqueueing them if necessary, and firing DSPOOM control events when it corresponds according to their time tag and the system current time.

Finally, the `Score` class contains Events that have already been defined as Processing Data objects. It naturally derives that the Score is also a Processing Data class. Figure 6.11 illustrates the

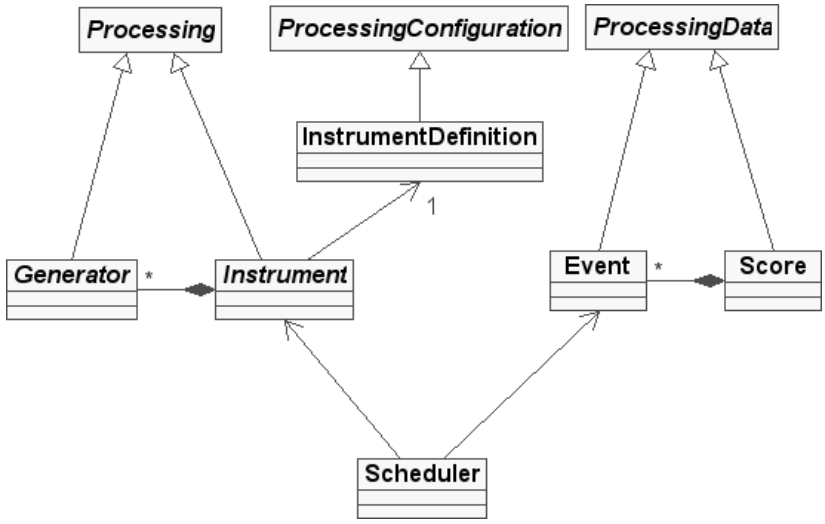

Figure 6.11: MetriXML as a DSPOOM model

class diagram of the MetriXML model in terms of the DSPOOM metamodel.

As explained in section 3.2.2.2 any Configuration implemented in CLAM has automatic XML passivation services.

In the following subsections we will illustrate the process of converting the MetriX textual formats into XML and finding the Object-Oriented Model related to them. For doing so, both for the MetriX Instrument Definition Language and MetriX Score Definition Language we will follow the following steps:

(1) First, we will translate the examples given in previous sections for the textual formats into XML.

(2) Then we will abstract the format model and specify it in XML Schema.

(3) Using the XML Schema abstract specification we will derive the OO model that will be illustrated in an UML class diagram.

§6.4.4.1 MetriXML Instrument Definition

By looking at the MIDL example in section 6.4.2 we can derive the following XML example of a MetriXML Instrument Definition document. Note that because of the structure of the CLAM ProcessingConfig class and some limitations of the automatic XML facilities in the framework we will not use XML *attributes* and declare everything as *elements*. We will also avoid some constructions like *lists* and the XML *inheritance* mechanism.

```
<InstrumentDefinition>
  <Generators>
    <Number>86</Number>
    <Name>Key</Name>
  </Generators>
  <TimbreSpace>
    <!--We define a single dimension>
    <Dimensions>1</Dimensions>
    <!--We set the interpolation type to linear>
    <InterpolationType>Linear</InterpolationType>
    <SampleArray>
      <Sample>
        <File>c:\Metrix\Piano\A0.sdif</File>
        <Position>0</Position>
      </Sample>
      <Sample>
        <File>c:\Metrix\Piano\A1.sdif</File>
        <Position>0.1412</Position>
      </Sample>
      <Sample>
        <File>c:\Metrix\Piano\A2.sdif</File>
        <Position>0.2824</Position>
      </Sample>
      <Sample>
        <File>c:\Metrix\Piano\A3.sdif</File>
        <Position>0.4235</Position>
      </Sample>
      <Sample>
        <File>c:\Metrix\Piano\A4.sdif</File>
        <Position>0.5765</Position>
      </Sample>
      <Sample>
        <File>c:\Metrix\Piano\A5.sdif</File>
        <Position>0.7176</Position>
      </Sample>
      <Sample>
        <File>c:\Metrix\Piano\A6.sdif</File>
        <Position>0.8588</Position>
      </Sample>
      <Sample>
        <File>c:\Metrix\Piano\A7.sdif</File>
        <Position>1</Position>
      </Sample>
    </SampleArray>
  <ParamBPFArray>
    <BPF>
      <Name>PianoLPF</Name>
      <InterpolationType>Linear</InterpolationType>
      <PointArray>
        <Point>
          <X>0</X>
          <Y>1</Y>
        </Point>
        <Point>
          <X>0.5</X>
          <Y>0.39*# +0.5</Y>
        </Point>
        <Point>
          <X>1</X>
          <Y>0.00006*# ^2</Y>
        </Point>
    </BPF>
  </ParamBPFArray>
  <TimeBPFArray>
    <BPF>
      <Name>FadeOut</Name>
      <InterpolationType>Linear</InterpolationType>
      <PointArray>
        <Point>
          <X>0</X>
          <Y>1</Y>
        </Point>
        <Point>
          <X>T</X>
          <Y>1</Y>
        </Point>
        <Point>
          <X>T+0.3</X>
          <Y>0.1</Y>
        </Point>
        <Point>
          <X>T+0.6</X>
          <Y>0</Y>
        </Point>
    </BPF>
```

```
  </TimeBPFArray>
  <LowLevelParamArray>
    <LowLevelParam>
      <Name>Gain</Name>
      <Minimum>0</Minimum>
      <Maximum>0.2</Maximum>
      <Default>0.1</Default>
    </LowLevelParam>
    <LowLevelParam>
      <Name>SineGain</Name>
      <Minimum>0</Minimum>
      <Maximum>1</Maximum>
      <Default>1</Default>
    </LowLevelParam>
    <LowLevelParam>
      <Name>ResGain</Name>
      <Minimum>0</Minimum>
      <Maximum>1</Maximum>
      <Default>1</Default>
    </LowLevelParam>
  </LowLevelParamArray>
  <HighLevelParamArray>
    <HighLevelParam>
      <Name>KeyVelocity</Name>
      <Minimum>0</Minimum>
      <Maximum>127</Maximum>
      <Default>64</Default>
      <LowLevelMapingArray>
        <LowLevelMaping>
          <LowLevelParam>Gain</LowLevelParam>
          <MapingFunction>#/127*0.2*TimeEnvelope(FadeOut)</MapingFunction>
        </LowLevelMaping>
        <LowLevelMaping>
          <LowLevelParam>SineGain</LowLevelParam>
          <MapingFunction>ParamBPF(PianoLPF)</MapingFunction>
        </LowLevelMaping>
        <LowLevelMaping>
          <LowLevelParam>ResGain</LowLevelParam>
          <MapingFunction>ParamBPF(PianoLPF)</MapingFunction>
        </LowLevelMaping>
    </HighLevelParam>
    <HighLevelParam>
      <Name>KeyNumber</Name>
      <Minimum>@</Minimum>
      <Maximum>@</Maximum>
      <Default>@</Default>
      <LowLevelMapingArray>
        <LowLevelMaping>
          <LowLevelParam>TimbreSpace</LowLevelParam>
          <MapingFunction>@/85</MapingFunction>
        </LowLevelMaping>
    </HighLevelParam>
    <HighLevelParam>
      <Name>Pitch</Name>
      <Minimum>27.16*1.0595^@</Minimum>
      <Maximum>28.8316*1.0595^@</Maximum>
      <Default>28*1.0595^@</Default>
      <LowLevelMapingArray>
        <LowLevelMaping>
          <LowLevelParam>TimbreSpace</LowLevelParam>
          <MapingFunction>@/85</MapingFunction>
        </LowLevelMaping>
      </LowLevelMapingArray>
    </HighLevelParam>
  </HighLevelParamArray>
</InstrumentDefinition>
```

And from the previous example we can extract the associated XML-Schema that will then be directly mapeable to the class structure.

```
<xsd:schema xmlns:xsd="http://www.w3.org/1999/XMLSchema">
  <xsd:element name="MetriXInstrumentDefinition" type="MetriXInstrumentDefinitionType"/>
  <xsd:complexType name="MetriXInstrumentDefinitionType">
    <sequence>
      <xsd:element name="Generators" type="GeneratorsType"/>
      <xsd:element name="TimbreSpace" type="TimbreSpaceType"/>
      <xsd:element name="ParamBPFArray" type="BPFArrayType" minOccurs="0"/>
      <xsd:element name="TimeBPFArray" type="BPFArrayType" minOccurs="0"/>
      <xsd:element name="LowLevelParamArray" type="LowLevelParamArrayType"/>
      <xsd:element name="HighLevelParamArray" type="HighLevelParamArrayType" minOccurs="0"/>
    </sequence>
```

```
</xsd:complexType>

<xsd:complexType name="GeneratorsType">
  <xsd:element name="Number" type="xsd:positiveInteger" />
  <xsd:element name="Name" type="xsd:string" />
</xsd:complexType>

<xsd:complexType name="TimbreSpaceType">
  <xsd:element name="Dimensions" type="xsd:positiveInteger"/>
  <xsd:element name="Interpolation">
    <xsd:simpletype type="InterpolationType">
      <xsd:restriction base="xsd:string">
        <xsd:enumeration value="linear">
        <xsd:enumeration value="parabolic">
        <xsd:enumeration value="polynomial">
        <xsd:enumeration value="spline">
      </xsd:restriction>
    </xsd:simpletype>
  </xsd:element>
  <xsd:element name="SampleArray">
    <xsd:complexType name="SampleArrayType">
      <xsd:element name="Sample">
        <xsd:complexType name="SampleType" minOccurs="3">
          <xsd:element name="File" type="xsd:string"/>
          <xsd:element name="Position" type="xsd:decimal"/>
        </xsd:complexType>
      </xsd:element>
    </xsd:complexType>
  </xsd:element>
</xsd:complexType>

<xsd:complexType name="BPFArrayType">
  <xsd:element name="BPF" minOccurs="1" maxOccurs="unbounded">
    <xsd:complexType name="BPFType">
        <xsd:element name="Name" type="xsd:string"/>
        <xsd:element name="Interpolation" type="InterpolationType"/>
        <xsd:element name="PointArray">
          <xsd:complexType name="PointArrayType">
            <xsd:element name="Point" minOccurs="2" maxOccurs="unbounded">
              <xsd:complexType name="PointType">
                <xsd:element name="X" type="xsd:string"/>
                <xsd:element name="Y" type="xsd:string"/>
              </xsd:complexType>
            </xsd:element>
          </xsd:complexType>
        </xsd:element>
    </xsd:complexType>
  </xsd:element>
</xsd:complexType>

<xsd:complexType name="LowLevelParamArrayType">
  <xsd:element name="LowLevelParam" minOccurs="1" maxOccurs="unbounded">
    <xsd:complexType name="LowLevelParamType">
      <xsd:element name="Name" type="xsd:string"/>
      <xsd:element name="Minimum" type="xsd:string"/>
      <xsd:element name="Maximum" type="xsd:string"/>
      <xsd:element name="Default" type="xsd:string"/>
    </xsd:complexType>
  </xsd:element>
</xsd:complexType>

<xsd:complexType name="HighLevelParamArrayType">
  <xsd:element name="HighLevelParam" minOccurs="1" maxOccurs="unbounded">
    <xsd:complexType name="HighLevelParamType">
      <xsd:element name="Name" type="xsd:string"/>
      <xsd:element name="Minimum" type="xsd:string"/>
      <xsd:element name="Maximum" type="xsd:string"/>
      <xsd:element name="Default" type="xsd:string"/>
      <xsd:element name="LowLevelMapingArray">
        <xsd:complexType name="LowLevelMapingArrayType">
          <xsd:element name="LowLevelMaping" minOccurs="1" maxOccurs="unbounded">
            <xsd:complexType name="LowLevelMapingType">
              <xsd:element name="LowLevelParam" type="xsd:string"/>
              <xsd:element name="MapingFunction" type="xsd:string"/>
            </xsd:complexType>
          </xsd:element>
        </xsd:complexType>
      </xsd:element>
    </xsd:complexType>
  </xsd:element>
</xsd:complexType>
```

Finally, and by looking at the previous XML-Schema, it is immediate to derive the Object-Oriented model illustrated in the class diagram in Figure 6.12.

Note that this class diagram turns out to be an extension/refinement of the MetriX model previously illustrated in figure 6.10.

§6.4.4.2 MetriXML Score Definition

We can do the same process transforming the textual MSDL (see 6.4.3) into XML:

```
<MetriXMLScore>
  <Header>
    <Score_Info>
      <Tempo>
        <Num>130</Num>
        <Den>2</Den>
      </Tempo>
      <Meter>
        <Num>3</Num>
        <Den>4</Den>
      </Meter>
      <Resolution>24</Resolution>
    </Score_Info>
    <Sound_Info>
      <SampleRate>44100</SampleRate>
      <Bits>8</Bits>
    <Sound_Info>
    <Instrument_Info>
      <Instrument>
        <Name>Piano</Name>
      </Instrument>
      <Instrument>
        <Name>guitar</Name>
      </Instrument>
      <Instrument>
        <Name>oboe</Name>
        <InstrumentDefinitionFile>\home\xamat\score\oboe.xmidl</InstrumentDefinitionFile>
      </Instrument>
      <Instrument>
        <Name>violin</Name>
        <InstrumentDefinitionFile>\home\xamat\score\violin.xmidl</InstrumentDefinitionFile>
      </Instrument>
      <Instrument>
        <Name>clarinet</Name>
        <SDIFDefinitionFile>\home\xamat\score\clarinet.sdif</SDIFDefinitionFile>
      </Instrument>
    </Instrument_Info>
    <Variables>
      <Variable>
        <Type>Instrument</Type>
        <Name>a</Name>
        <Ref>piano</Ref>
      </Variable>
      <Variable>
        <Type>Instrument</Type>
        <Name>c</Name>
        <Ref>violin</Ref>
      </Variable>
      <Variable>
        <Type>Generator</Type>
        <Name>a1</Name>
        <Ref>a.Key0</Ref>
      </Variable>
    </Variables>
  </Header>

  <Body>
    <Event>
      <Time type="musical">
        <Measure>01</Measure>
        <Beat>01</Beat>
        <NoteNumber>02</NoteNumber>
        <NoteType>04</NoteType>
      <Time>
      <Receiver>a1</Receiver>
      <Message>
        <Parameter>Pitch</Parameter>
```

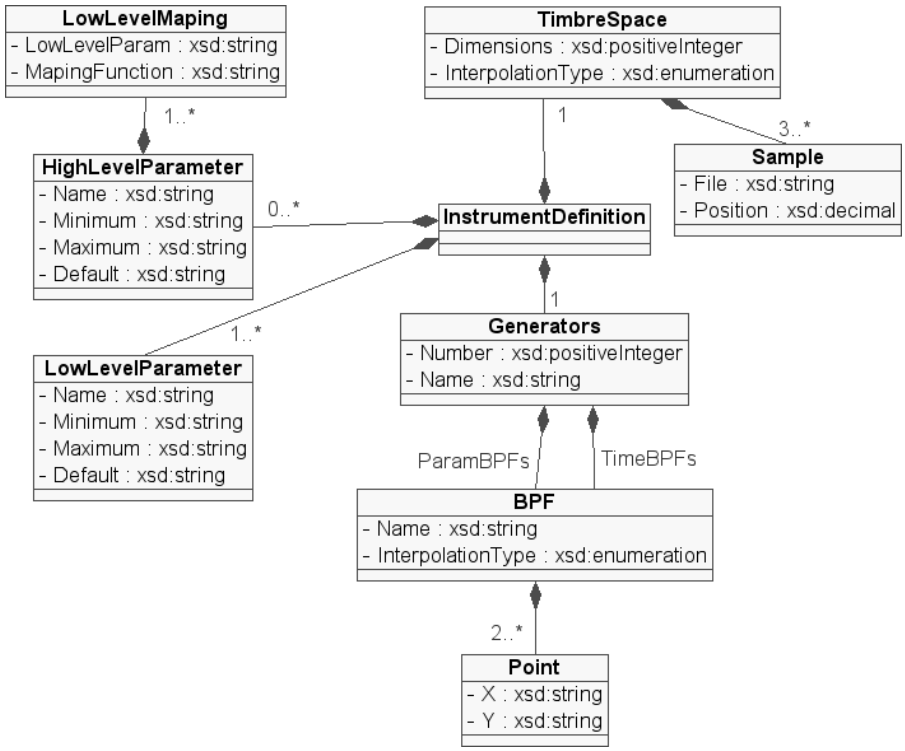

Figure 6.12: MetriXML Instrument Definition class diagram

```
        <Value>C#3</Value>
      </Message>
      <Message>
        <Parameter>Loudness</Parameter>
        <Value>mf</Value>
      </Message>
      <Message>
        <Parameter>Duration</Parameter>
        <Value>t00:00:01</Value>
      </Message>
    </Event>
    <Event>
      <Time type="temporal">
        <Seconds>01</Seconds>
      </Time>
      <Receiver>clarinet</Receiver>
      <Message>
        <Parameter>Gain</Parameter>
        <Value>0,1 1,0.5</Value>
      </Message>
    </Event>
    <Event>
      <Time type="temporal">
        <Seconds>+04</Seconds>
      </Time>
      <Receiver>piano.key2</Receiver>
      <Message>
        <Parameter>Pitch</Parameter>
        <Value>f2</Value>
      </Message>
    </Event>
    <Event>
      <Time type="temporal">
        <Seconds>04</Seconds>
      </Time>
      <Receiver>Score_Info</Receiver>
      <Message>
        <Parameter>Tempo</Parameter>
        <Value>140</Value>
      </Message>
    </Event>
    <!--etc.../>
  </Body>
</MetriXMLScore>
```

And we can now build the associated XML-Schema:

```
<xsd:schema xmlns:xsd="http://www.w3.org/1999/XMLSchema">
  <xsd:element name="MetriXScore" type="MetriXScoreType"/>
  <xsd:complexType name="rational">
    <xsd:element name="Num" type="xsd:integer">
    <xsd:element name="Den" type="xsd:positiveInteger">
  </xsd:complexType>
  <xsd:complexType name="MetriXScoreType">
    <xsd:sequence>
      <xsd:element name="Header">
        <xsd:complexType name="HeaderType">
          <xsd:element name="ScoreInfo">
            <xsd:complexType name="ScoreInfoType">
              <xsd:element name="Tempo" type="rational" minOccurs="0"/>
              <xsd:element name="Meter" type="rational" minOccurs="0"/>
              <xsd:element name="Resolution" type="xsd:positiveInteger" minOccurs="0"/>
            </xsd:complexType>
          </xsd:element>
          <xsd:element name="SoundInfo">
            <xsd:complexType name="SoundInfoType">
              <xsd:element name="SampleRate" type="xsd:positiveInteger" minOccurs="0"/>
              <xsd:element name="Bits" type="xsd:positiveInteger" minOccurs="0"/>
            </xsd:complexType>
          </xsd:element>
          <xsd:element name="InstrumentInfo">
            <xsd:complexType name="InstrumentInfoType">
              <xsd:element name="Instrument" maxOccurs="unbounded">
                <xsd:complexType name="InstrumentType">
                  <xsd:element name="Name" type="xsd:string"/>
                  <xsd:element name="InstrumentDefinitionFile" type="xsd:string" minOccurs="0"/>
                  <xsd:element name="SDIFDefinitionFile" type="xsd:string" minOccurs="0"/>
                </xsd:complexType>
              </xsd:element>
            </xsd:complexType>
          </xsd:element>
          <xsd:element name="Variables">
            <xsd:complexType name="VariablesType">
              <xsd:element name="Variable" maxOccurs="unbounded">
```

```
            <xsd:complexType name="VariableType">
              <xsd:element name="VariableKind">
                <xsd:simpleType=''VariableKindType''>

                  <xsd:restriction base=''xsd:string''>

                    <xsd:enumeration value=''Instrument''/>

                    <xsd:enumeration value=''Generator''/>
                  </xsd:restriction>
                </xsd:simpleType>

              <xsd:element name="Name" type=''xsd:string''/>
              <xsd:element name="Ref" type=''xsd:string''/>
            </xsd:complexType>
          </xsd:element>
        </xsd:complexType>
      </xsd:element>
    </xsd:complexType>
  </xsd:element>

  <xsd:element name="Body">
    <xsd:complexType name="BodyType">
      <xsd:element name=''EventArray''>
        <xsd:complexType name="EventArrayType">
          <xsd::element name=“Event” maxOccurs=''unbounded''>
            <xsd:complexType name=''EventType''>
              <xsd:element name=''Time''>
                <xsd:complexType name=''TimeType''>
                  <xsd:choice>
                    <xsd:element name=''TemporalTimeTag''>
                      <xsd:complexType name=''TemporalTimeTagType''>
                        <xsd:element name=''Hours'' type=''nonNegativeInteger'' maxInclusive=''24''>
                        <xsd:element name=''Minutes'' type=''nonNegativeInteger'' maxInclusive=''60''>
                        <xsd:element name=''Seconds'' type=''nonNegativeInteger'' maxInclusive=''60''>
                        <xsd:element name=''Frames'' type=''nonNegativeInteger''>
                      </xsd:complexType>
                    </xsd:element>
                    <xsd:element name=''MusicalTimeTag''>
                      <xsd:complexType name=''TemporalTimeTagType''>
                        <xsd:element name=''Measure'' type=''nonNegativeInteger''>
                        <xsd:element name=''Beats'' type=''nonNegativeInteger''>
                        <xsd:element name=''Notes'' type=''nonNegativeInteger''>
                        <xsd:element name=''NoteKind'' type=''nonNegativeInteger''>
                      </complexType>
                    </xsd:element>
                  </xsd:choice>
                </xsd:complexType>
              </xsd:element>
              <xsd:element name=''Receiver'' type=''xsd:string''/>
              <xsd:element name=''Message''>
                <xsd:complexType name=''MessageType''>
                  <xsd:element name=''Parameter'' type=''xsd:string''/>
                  <xsd:element name=''Value'' type=''xsd:string''/>
                </xsd:complexType>
              </xsd:element>
            </xsd:complexType>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
    </xsd:complexType>
```

And as we did for the Instrument Definition, by looking at the previous schema we can now build the object-oriented class diagram of the MetriXML Score. This class diagram is illustrated in Figure 6.13.

§6.5 Summary and Conclusions

In this chapter we have presented an object-oriented music model that can be interpreted as an instance of the basic Digital Signal Processing Object-Oriented Metamodel dealing in this case with higher-level symbolic musical data.

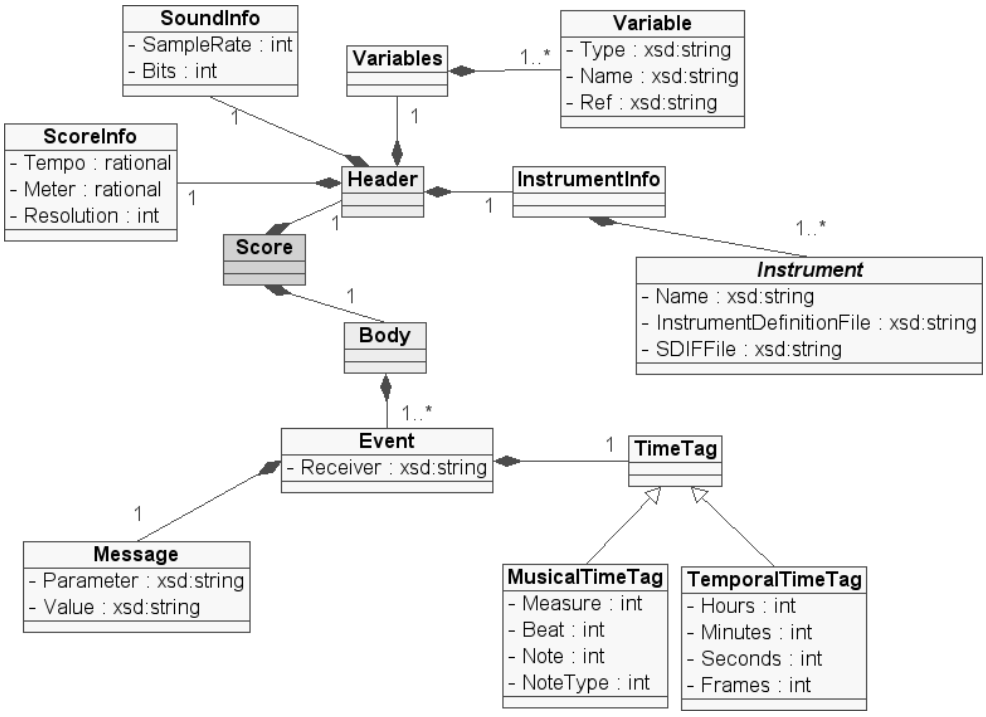

Figure 6.13: MetriXML Score class diagram

Following again the object-oriented paradigm we model a music system as a set of interrelated objects. These objects will in general belong to one of the following abstract classes: Instrument, Generator, Note or Score.

An *Instrument* is a generating Processing object that receives input controls and generates an output sound. An Instrument is as a matter of fact a logical grouping of autonomous units named *Generators*. A Generator is the atomic sound producing unit in an Instrument and can be independently controlled from the other Generators (although it often receives their influence). Examples are the six strings in a guitar or each of the keys in a piano.

A Note is the actual sounding object attached to each Generator. A Note can be turned on and off and its properties depend on the internal state of its associated Generator and Instrument.

Finally the internal state of the whole object-oriented music system changes in response to events that are sent to particular Instruments or Generators. A time-ordered collection of such events is known as a *Score*.

The abstract model just described is implemented in the MetriX language or in its XML-based version MetriX-ML. MetriX-ML is a Music-N language that therefore offers a way of defining both Instruments and Scores. It is implemented in CLAM and, apart from the concepts previously presented, includes support for defining timbre spaces, break-point-functions and relations between control parameters in an Instrument.

Therefore the model and its implementation is fully usable. Nevertheless the model is not complete as it has not been adapted to different synthesis techniques apart from Spectral Modelling Synthesis. But its definition and implementation in the context of the CLAM framework make it extensible and adaptable to future needs.

Conclusions and Future Work

In this Thesis we have presented a metamodel for digital signal processing named DSPOOM.

This metamodel uses the Object-Oriented paradigm and exploits the relation between this paradigm and the Graphical Models of Computation used mainly in system engineering. Because these and other concepts used in the Thesis are borrowed from different areas of knowledge it is first necessary to establish a coherent corpus and conceptual framework. This is introduced in chapter 1. We believe that this chapter cannot be only understood as a conceptual introduction, though. The way that the different concepts are presented and related in between them constitutes a set of hypotheses in itself, hypotheses that are then elaborated and demonstrated in the following chapters. The main hypothesis can be summarized in: "Systems are object-oriented"; "Signal processing systems can be modeled through object-oriented and graphical models of computation"; and "Frameworks generate metamodels".

A metamodel cannot be defended in any other way than through the evaluation of the models it generates and their usefulness for modeling systems in a particular domain. In this Thesis this is accomplished through the CLAM framework. CLAM, which is a framework for music and audio processing, is as a matter of fact the origin of the DSPOOM metamodel that was obtained as an abstraction of the framework and the different models included in it. In chapter 3 we have presented the main features of the CLAM framework and we have compared them to some included in similar frameworks. For doing so we have had to first present a thorough review of similar environments in chapter 2. As already pointed out in the corresponding chapter we believe that CLAM represents a different approach based especially on the use of good software engineering techniques.

The Digital Signal Processing Object-Oriented Metamodel (DSPOOM) is an abstraction of different conceptual conclusions we reached during the framework development process. Nevertheless it shares many concepts and ideas with the frameworks reviewed in this thesis. It presents a clear way to model a signal processing system while it enhances the separation between data and process,

and idea that is intuitive to signal processing engineers. It also offers and explicit graphical Model of Computation that helps in modeling complex systems and understanding them.

The Object-Oriented Content Transmission Metamodel is a particular instance of DSPOOM that can be used to model the so-called "Content-based applications". The OOCTM is also demonstrated through some particular applications that partially instantiate the metamodel. The applicability of the whole metamodel is still not guaranteed as it depends on some ongoing research that is supposed to develop the necessary technology for some of the building blocks. Nevertheless we believe that the conceptual framework is useful in itself and will help in understanding also the practical applicability of the metamodel.

Finally the object-oriented music model presented through the MetriX language is a particular model that can be understood as a proof of concept that object-orientation in general and DSPOOM in particular can be used to effectively model more abstract domains such as music. Although the scope of this music model is not as broad as to give response to any musical application and its implementation presents some limitations, we believe it is generic enough so as to become useful in most situations with a few minor additions (see again section 6.6 on Future Work)

After this concluding conceptual summary we will finish this section by completing the glossary given at the Introduction and by outlining possible lines of future work.

§6.5.1 Final Glossary

Just as in the introduction we gave a brief glossary of the fundamental concepts upon which this Thesis was to be built, we will now summarize the most important ideas also in the form of a glossary. We will give short definitions and refer to the related section in this document.

- CLAM: Acronym for "C++ Library for Audio and Music". Audio and music development framework that represents the practical realization of the DSPOOM metamodel. See chapter 3.

- DSPOOM: Acronym for "Digital Signal Processing Object-Oriented Metamodel". Abstract metamodel that includes a complete object-oriented approach and a graphical model of computation that can be instantiated to model any digital signal processing system especially in the audio and music domain. See chapter 4.

- Processing Object: Basic building block of a DSPOOM model. It is the object-oriented encapsulation of a process. See section 4.1.1

- Processing Data Object: Object that encapsulates all data in a DSPOOM model offering general services such as a homogeneous interface or passivation/activation facilities. Inputs and outputs to a Processing object are all Processing Data objects (or Controls if the input is asynchronous). See 4.1.2

- Network: Dynamic grouping of Processing objects in DSPOOM. A Network basically contains a list of Processing objects and a list of connections between Ports and Controls. Internal flow control issues are automatically handled. See section 4.1.3.

- Processing Composite: Static grouping of DSPOOM Processing objects. In a Processing Composite class the developer defines the internal behavior and flow control issues. See section 4.1.3.

- OOCTM: Acronym for Object-Oriented Content Transmission Metamodel. Following this abstract model, information transmission can be seen as a sequence of the following processes: analysis, understanding, encoding, transmission, decoding, interpreting and synthesis. See chapter 5.

- Content: Any semantic information contained in a signal that can be interpreted by the targeted user. See 5.1.1

- Sound Object: A Sound Object is considered to be any entity in a sound system. Following the DSPOOM metamodel Sound Objects can also be classified into Processing Sound Object such as Instruments or Generators and Data Sound Objects such as Tracks or Notes. See section 5.1.2.

- Semantic Transmitter: Transmitter that, in the OOCTM, is in charge of analyzing, understanding, encoding and transmitting the content in a signal. See section 5.2.1.

- Semantic Receiver: Receiver that, in the OOCTM is in charge of receiving, decoding, interpreting, and synthesizing the transmitted content into a signal. See 5.2.2.

- Content-based Analysis: Multi-step signal analysis in which the goal is to extract content description. See 5.3.3.

- Content-based Transformation: Signal transformation that addresses the content-level by offering the user semantically meaningful control. The basic scheme is based on an analysis-synthesis process derived from the OOCTM (see 5.3.4).

- Instrument Object: Music object that receives input controls in the form of music events and maps them into Generator controls. See 6.1.

- Generator Object: A Generator is the minimum entity in an Instrument capable of producing sound by itself. See 6.1.

- Note Object: A musical Note is defined as a sound object that has a precise and explicit active lifespan (i.e. the note duration), a certain loudness, an optional pitch and other optional attributes. A Note Object is actually a placeholder for a Generator state and can be directly synthesized into audio. See 6.2

- MetriX: Framework that provides an implementation of the Object-Oriented Music Model and offers two languages: the MIDL and the MSDL. See section 6.4.

- MIDL: MetriX Instrument Definition Language. Language for describing an Instrument subclass behavior by specifying its Generators, Timbre Space and Parameter mappings. See 6.4.2.

- MSDL: MetriX Score Definition Language. Language for describing a musical score as a sequence of time ordered events. See 6.4.3.

- MetriXML: Implementation of MetriX in the CLAM framework, mapping concepts to DSPOOM metaclasses and obtaining XML representation for free.

§ Future Work

Finally we will mention the main lines of research that remain open after finishing this Thesis. We will outline them in relation to the main contributions of the thesis: the CLAM framework, the DSPOOM metamodel, the OOCTM metamodel and the MetriX object-oriented music model.

In relation to the CLAM framework, in section A.1.5 we describe what are the main future lines of development. The main idea is that the framework's usability can still be improved in different ways and by doing so we believe that it will be accepted more naturally by more users. In order to do so we must address issues such as an easier deployment in any platform, a cleaner and clearer interface or the addition of more automatic flow control tools. All these issues are detailed in annex 6.6.

The Digital Signal Processing Object-Oriented Metamodel (DSPOOM) is complete and demonstrated through the CLAM framework. Nevertheless, we claimed that the metamodel is valid for any digital signal processing model and this still has to be tested thoroughly. Although some initial experiments confirm that the metamodel is useful in other domains such as image processing it should be tested in different situations and under different conditions. It is possible that the usage of the meta-

model in different situations may end up reverting into the metamodel itself clarifying some details such as the convenience of using one graphical MoC or another (see section 4.3).

The Object-Oriented Content Transmission Metamodel (OOCTM) has been partially instantiated and demonstrated especially through content-based transformations (see 5.3.4). Nevertheless, the overall metamodel could be only proven through a rather limited application (see 5.4). These limitations are due to some restrictions especially on the analyzer process. This block addresses problems such as automatic instrument classification or musical transcription, which are still unsolved research topics. The full strength of the metamodel will only be visible when this technologies are finally available. Nevertheless we believe that its formulation will help in structuring related systems and already gives a conceptual framework for research in this area. In this sense it will also be important to exploit OOCTM's relation to other more well-stablished models/metamodels such as Structured Audio or S&W (see 5.3).

Finally we already commented that the MetriX music model presented in chapter 6 does not intend to be as general as the other metamodels previously presented. MetriX is used as a proof of concept and demonstration that the object-oriented paradigm and the DSPOOM metamodel in particular may be used to effectively model the symbolic music domain. Nevertheless, we believe that the resulting model could with minor enhancements become of general applicability. The major shortcomings of the models are that it lacks a way of describing performers and that analysis information is not fully integrated.

Concentrating on the former limitation we observe that any music performance is made of at least three main actors: the score, the instrument, and the performer. The performer reads and interprets the score and acts on the instrument accordingly. The definition of Score in our OO music model is a hybrid between the traditional score written by a composer and the interpretation given by the performer. We would like to have a more clear distinction between the three levels and would therefore be interested in defining a third "document" specifying how a performer behaves. This file would contain a list of constraints to be applied to the way score events would be sent to the instrument.

As for the second limitation, it would be interesting to better integrate MetriX with our OOCTM. In order to do so new ways to include analysis information into both the Score and the Instrument definition need to be devised. By doing so we would be accomplishing a more compact conceptual integration of the three main axis in this thesis: CLAM/DSPOOM, OOCTM and MetriX.

It is clear that any piece of work leaves many open issues and lines of future research. It is even more so in a Thesis such as this where models of very general applicability are presented. Nevertheless it is important to ask oneself whether the ground that has been built is solid enough so as to continue

constructing from it. In this sense we believe that we have accomplished our goals and we hope to have contributed to the advance of not only present but also future research in our field.

Bibliography

[Abbot, 1983] Abbot, R. (1983). Program design by informal english descriptions. *P*rogram Design by Informal English Descriptions, 26(11):882–894.

[Ackermann, 1994a] Ackermann, P. (1994a). Design and Implementation of an Object Oriented Media Composition Framework. In *Proceedings of the 1994 International Computer Music Conference ICMC94*. Computer Music Association.

[Ackermann, 1994b] Ackermann, P. (1994b). Direct Manipulation of Temporal Structures in a Multimedia Application Framework. In *Proceedings of the 1994 ACM Multimedia Conference*.

[Adair, 1995] Adair, D. (1995). Building Object-Oriented Frameworks (Part 1). *AIXpert*.

[Adaptive et al., 2003] Adaptive et al. (2003). Meta Object Facility (MOF) 2.0 Core Proposal. Technical report, ACM's Object Management Group. Modification to the MOF 1.4 standard still to be adopted.

[Agha, 1986] Agha, G. (1986). *Actors: A model of concurrent computation in Distributed Systems*. MIT Press, Cambridge, MA.

[Agon and Assayag, 2002] Agon, C. and Assayag, G. (2002). Object-Oriented Programming in OpenMusic. In *Topos of Music*. Verlag Ed.

[Agon et al., 2000] Agon, C., Stroppa, M., and Assayag, G. (2000). High Level Musical Control of Sound Synthesis in OpenMusic. In *Proceedings of the 2000 International Computer Music Conference (ICMC '00)*, Berlin, Allemagne.

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

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

[Amatriain et al., 2003] Amatriain, X., Bonada, J., Loscos, A., Arcos, J. L., and Verfaille, V. (2003). Content-based Transformations. *Journal of New Music Research*, 32(1).

[Amatriain et al., 2001] Amatriain, X., Bonada, J., Loscos, A., and Serra, X. (2001). Spectral Modeling for Higher-level Sound Transformations. In *Proceedings of the first MOSART Workshop on Current Research Directions in Computer Music*.

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

[Amatriain et al., 1998] Amatriain, X., Bondada, J., and Serra, X. (1998). METRIX: A Musical Data Definition Language and Data Structure for a Spectral Modeling Based Synthesizer. In *Proceedings of the 1st International Conference on Digital Audio Effects (DAFX98)*, Barcelona.

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

[Amatriain and Herrera, 2001a] Amatriain, X. and Herrera, P. (2001a). Audio Content Transmission. In *Proceedings for the 4th International Conference on Digital Audio Effects (DAFX01)*, Limerick.

[Amatriain and Herrera, 2001b] Amatriain, X. and Herrera, P. (2001b). Transmitting Audio Content as Sound Objects. In *Proceedings of the AES 22nd Conference on Virtual, Synthetic, and Entertainment Audio*, Helsinki. Audio Engineering Society.

[Ambler, 2003] Ambler, S. W. (2003). Agile Model Driven Development Is Good Enough. *IEEE Software*.

[Anderson, 1983] Anderson, J. (1983). *The Architecture of Cognition*. Harvard University Press, Cambridge, Massachusets.

[Arcos et al., 1998] Arcos, J. L., de Mántaras, R. L., and Serra, X. (1998). Saxex: a Case-Based Reasoning System for Generating Expressive Musical Performances. *Journal of New Music Research*, 27(3).

[Assayag and Agon, 2000] Assayag, G. and Agon, C. (2000). OpenMusic : un langage de programmation visuelle pour la composition. In *Encyclopédie pour l'ingénieur informaticien*. Hermés.

[Barton, 1994] Barton, R. (1994). Metamodeling: A State of the Art review. In *Proceedings of the 1994 Winter Simulation Conference*.

[Bastide, 1995] Bastide, R. (1995). Approaches in unifying petri nets and the object-oriented approach. In *P*roceedings of the Application and Theory of Petri Nets 1995 - Workshop on Object-Oriented Programming and Models of Concurrency.

[Beck, 1999] Beck, K. (1999). *Extreme Programming Explained*. Addison Wesley.

[Beck, 2002] Beck, K. (2002). The Metaphor Metaphor. Invited Talk at the 2002 Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA '02).

[Beck and Johnson, 1994] Beck, K. and Johnson, R. (1994). Patterns Generate Architectures. In *Proceedings of the 8th European Conference on Object-Oriented Programming*, Bologna, Italy.

[Bencina, 1998] Bencina, R. (1998). Oasis Rose the Composition - Real-time DSP with AudioMulch. In *Proceedings of the Australian Computer Music Conference*, Camberra.

[Bencina, 2003] Bencina, R. (2003). Port audio and media synchronization. In *P*roceedings of the 2003 Australasian Computer Music Association (ACMC'03).

[Bencina and Burk, 2001] Bencina, R. and Burk, P. (2001). Port Audio: an Open Source Cross Platform Audio API. In *Proceedings of the 2001 International Computer Music Conference (ICMC '01)*. Computer Music Associaciation.

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

[Bonada, 1997] Bonada, J. (1997). Desenvolupament d'un entorn gráfic per a l'análisi, transformació i síntesi de sons mitjanant models espectrals. Master's thesis, UPC. Barcelona.

[Bonada, 2000] Bonada, J. (2000). Automatic Technique in Frequency Domain for near-Lossless Time-Scale Modification of Audio. In *Proceedings of the 2000 International Computer Music Conference (ICMC '00)*, San Francisco. Computer Music Association.

[Booch, 1994a] Booch, G. (1994a). Designing an Application Framework. *Dr. Dobb's Journal*, 19(2):24.

[Booch, 1994b] Booch, G. (1994b). *Object-Oriented Analysis and Design with Applications*. Benjamin/Cummings, second edition edition.

[Bosch et al., 1999] Bosch, J., Molin, M., Mattson, M., and Bengtsson, P. (1999). *Building Application Frameworks*, chapter Object-oriented frameworks - Problems & Experiences. Wiley and Sons.

[Boulding, 1969] Boulding, K. (1969). *Modern Systems Research for the Behavioral Scientist, A Sourcebook*, chapter General Systems Theory - The Skeleton of Science. Aldine Publishing Company, Chicago.

[Bregman, 1990] Bregman, A. (1990). *Auditory Scene Analysis: the Perceptual Organization of Sound*. MIT Press, Cambridge, MA,.

[Buck and Lee, 1994] Buck, J. and Lee, E. A. (1994). *Advanced Topics in Dataflow Computing and Multithreading*, chapter The Token Flow Model. IEEE Computer Society Press.

[Budd, 1991] Budd, T. (1991). *An Introduction to Object-Oriented Programming*. Addison-Wesley.

[Burk, 1998] Burk, P. (1998). JSyn- A Real-time Synthesis API for Java. In *Proceedings of the 1998 International Computer Music Conference (ICMC '98)*. Computer Music Associaciation.

[Buschman et al., 1996] Buschman, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. (1996). *Pattern-Oriented Software Architecture - A System of Patterns*. John Wiley & Sons.

[Camurri, 1999] Camurri, A. (1999). Music Content Processing And Multimedia: Case Studies and Emerging Applications of Intelligent Interactive Systems. *Journal of New Music Research*, 28(4):351–363.

[Casey, 2001] Casey, M. (2001). MPEG-7 Sound-Recognition Tools. *IEEE Transactions on Circuits and Systems for Video Technology*, 11(6).

[Caughlin, 1997] Caughlin, D. (1997). Automating the Metamodeling Process. In *Proceedings of the 1997 Winter Simulation Conference.*

[Chai and Vercoe, 2000] Chai, W. and Vercoe, B. (2000). Using User Models in Information Retrieval Systems. In *Proceedings of the 1st International Symposium on Music Information Retrieval (ISMIR 00).*

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

[Cherry, 1957] Cherry, E. (1957). *On Human Communication.* Wiley, New York.

[Chiariglione, 2000] Chiariglione, L. (2000). The Value of Content. *Technology Reviews.*

[Chion, 1983] Chion, M. (1983). *Guide des Objets Sonores. Pierre Schaeffer et la Reserche Musicale.* INA-GRM/BUCHET, Chastel.

[Cockburn, 2002] Cockburn, A. (2002). *Agile Software Development.* Addison-Wesley.

[Cook, 1996] Cook, P. (1996). Synthesis Toolkit in C++. In *Proceedings of the 1996 SIGGRAPH.*

[Cook, 2004] Cook, P. (2004). *Synthesis toolKit Instrument Network Interface (SKINI) 0.9 Implementation notes.* Princeton University.

[Cook and Scavone, 1999] Cook, P. and Scavone, G. (1999). The Synthesis Toolkik (STK). In *Proceedings of the 1999 International Computer Music Conference (ICMC99)*, Beijing, China. Computer Music Association.

[Cook and Scavone, 2003] Cook, P. and Scavone, G. (2003). *STK software documentation.* http://www-ccrma-stanford.edu/software/stk.

[Dahl and Nygaard, 1966] Dahl, O. and Nygaard, K. (1966). Simula: An Algol-based Simulation Language. In *Communications of the ACM*, volume 9.

[Danks, 1997] Danks, M. (1997). Real-time image and video processing in GEM. In *Proceedings of the 1997 International Music Conference (ICMC '97)*, pages 220–223. Computer Music Association.

[Dannenberg, 1993] Dannenberg, R. (1993). The Implementation of Nyquist, a Sound Synthesis Language. In *Proceedings of the 1993 International Computer Music Conference (ICMC '93)*, pages 168–171. Computer Music Association.

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

[Dannenberg and Brandt, 1996a] Dannenberg, R. B. and Brandt, E. (1996a). A Flexible Real-Time Software Synthesis System. In *Proceedings of the 1996 International Computer Music Conference (ICMC96)*, pages 270–273.

[Dannenberg and Brandt, 1996b] Dannenberg, R. B. and Brandt, E. (1996b). A Portable, High-Performance System for Interactive Audio Processing. In *Proceedings of the 1996 International Computer Music Conference (ICMC96)*, pages 270–273. International Computer Music Association.

[Dannenberg and Rubine, 1995] Dannenberg, R. B. and Rubine, D. (1995). Toward Modular, Portable. Real-Time Software. In *Proceedings of the 1995 International Computer Music Conference (ICMC95)*, pages 65–72. International Computer Music Association.

[Darnell, 1972] Darnell, D. (1972). *Approaches to Human Communication*. Spartan Books, New York.

[de Champeaux et al., 1993] de Champeaux, D., Lea, D., and Faure, P. (1993). *O*bject-Oriented System Development. Addison Wesley.

[DeGreene, 1970] DeGreene, K. (1970). *Systems Psychology*, chapter Systems and Psychology. McGraw-Hill.

[Devedzic, 2002] Devedzic, V. (2002). Understanding Ontological Engineering. *Communications of the ACM*, 45(4).

[Dobrian et al., 2000] Dobrian, C. et al. (2000). *MSP: Getting Started, Tutorial and Topics, and Reference*.

[Déchelle, 2000] Déchelle, F. (2000). jMax : Un environnement pour la réalisation d'applications musicales sur Linux. In *Journées d'Informatique musicale*, Bordeaux, France.

[Déchelle, 2003] Déchelle, F. (2003). Various IRCAM free software: jMax and OpenMusic. In *Linux Audio Developers Meeting*, Karlsruhe, Allemagne.

[Déchelle et al., 1998] Déchelle, F., Borghesi, R., de Cecco, M., Maggi, E., Rovan, J. B., and Schnell, N. (1998). jMax: a new JAVA-based Editing and Control System for Real-time Musical Applications. In *Proceedings of the 1998 International Computer Music Conference(ICMC '98)*.

[Déchelle et al., 1999a] Déchelle, F., Borghesi, R., de Cecco, M., Maggi, E., Rovan, J. B., and Schnell, N. (1999a). jMax: An Environment for Real-Time Musical Applications. *Computer Music Journal*, 23-3:50–58.

[Déchelle et al., 1999b] Déchelle, F., Borghesi, R., de Cecco, M., Maggi, E., Rovan, J. B., and Schnell, N. (1999b). jMax Recent Developments. In *Proceedings of the 1999 International Computer Music Conference(ICMC '99)*, Pekin, Chine.

[Déchelle et al., 2000] Déchelle, F., Borghesi, R., Orio, N., and Schnell, N. (2000). The jMax environment: an overview of new features. In *ICMC: International Computer Music Conference*, Allemagne, Berlin.

[Déchelle and Tisserand, 2003] Déchelle, F. and Tisserand, P. (2003). Free software at IRCAM: jMax, OpenMusic. In *A*GNULA - Bring Your Own Laptop, Prato, Italie.

[Ebrahimi and Christopoulos, 1998] Ebrahimi, T. and Christopoulos, C. (1998). Can MPEG-7 be used beyond database application? Technical Report M3861, MPEG, Atlantic City.

[Edwards, 1995] Edwards, S. (1995). Streams: a Pattern for "Pull-Driven. In Coplien, J. O. and Schmidt, D. C., editors, *Pattern Languages of Program Design*, volume vol.1, chapter 21. Addison-Wesley.

[Fletcher and Munson, 1933] Fletcher, H. and Munson, W. (1933). Loudness, its Definition, Measurement and Calculation. *Journal of the Acoustical Society of America*, 5:82–108.

[Fogel, 1999] Fogel, K. (1999). *Open Source Development with CVS*. CoriolisOpen Press.

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

[Frigo and Johnson, 1998] Frigo, M. and Johnson, S. G. (1998). FFTW: An adaptive software architecture for the FFT. In *P*roc. 1998 IEEE Intl. Conf. Acoustics Speech and Signal Processing, volume 3, pages 1381–1384. IEEE.

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

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

[Garlan and Shaw, 1993] Garlan, D. and Shaw, M. (1993). *Advances in Software Engineering and Knowledge Engineering*, volume 1, chapter An introduction to Software Architecture. World Scientific Publishing Company.

[Graham, 1991] Graham, I. (1991). *Object Oriented Methods*. Addison-Wesley.

[Griffin, 1997] Griffin, E. (1997). *A First Look at Communication Theory*. McGraw-Hill, Inc., third edition edition.

[Gómez et al., 2003a] Gómez, E., Grachten, M., Amatriain, X., and Arcos, J. (2003a). Melodic characterization of monophonic recordings for expressive tempo transformations. In *Proceedings of Stockholm Music Acoustics Conference 2003*.

[Gómez et al., 2003b] Gómez, E., Peterschmitt, G., Amatriain, X., and Herrera, P. (2003b). Content-based melodic transformations of audio for a music processing application. In *Proceedings of 6th International Conference on Digital Audio Effects*.

[Halbert and O'Brien, 1987] Halbert, D. and O'Brien, P. (1987). Using Types and Inheritance in Object-oriented Programs. *IEEE Software*.

[Hall and Fagen, 1956] Hall, A. and Fagen, R. (1956). *Yearbook of the Society for the Advancement of General Systems Theory*, volume General Systems I, chapter Definition of System. Ann Arbor.

[Hebel, 1991] Hebel, K. J. (1991). *The Well-tempered Object. Musical Applications of Object-Oriented Software Technology*, chapter Javelina: An Environment for Digital Signal Processor Software Development, pages 171–187. MIT Press.

[Helmuth, 1990] Helmuth, M. (1990). PATCHMIX A C++ Interface to Cmix. In *Proceedings of the 1990 International Computer Music Conference (ICMC '90)*, pages 273–275. Computer Music Association.

[Herrera et al., 2000] Herrera, P., Amatriain, X., Batlle, E., and Serra, X. (2000). Towards instrument segmentation for music content description: a critical review of instrument classification techniques. In *P*roceedings of the 1st International Symposium on Music Information Retrieval.

[Hewit, 1977] Hewit, C. (1977). Viewing control structures as patterns of passing messages. *Journal of Artificial Intelligence*, 8(3):323–363.

[Hickey, 1995] Hickey, R. (1995). Callbacks in C++ using Template Functors. *C++ Report*.

[Honing, 1990] Honing, H. (1990). Poco: An environment for analysing, modifying and generating expression in music. In *P*roceedings of the 1990 International Music Conference (ICMC '90), pages 364–368.

[Huron, 1995] Huron, D. (1995). *T*he Humdrum Toolkit: Reference Manual. Center for Computer Assisted Research in the Humanities.

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

[Jackson, 1995] Jackson, M. (1995). *Software Requirements and Specifications*. Addison-Wesley, Harlow, England.

[Jaffe and Boynton, 1991] Jaffe, D. and Boynton, L. (1991). *The Well-tempered Object. Musical Applications of Object-Oriented Software Technology*, chapter An Overview of the Sound and Music Kits for the NeXT Computer, pages 107–118. MIT Press.

[Jafry, 2000] Jafry, Y. (2000). A Modular Real-Time PC-Based Audio Processing Tool for Effects Developers, Engineers, Musicians, and Educators. In *Proceedings of the 2000 Conference on Digital Audio Effects (Dafx-00)*.

[Jaimes et al., 2000] Jaimes, A., Benitez, A., and Chang, S. (2000). Multiple Level Classification of Descriptions for Audio Content. Technical Report M6114, MPEG, Geneva.

[Janneck and Esser, 2002] Janneck, J. W. and Esser, R. (2002). Higher-order petri net modelling - techniques and applications. In *P*roceedings of the Workshop on Software Engineering and Formal Methods, Petri Nets 2002, Adelaide, Australia.

[Johnson, 1992] Johnson, R. E. (1992). Documenting Frameworks with Patterns. In *Proceedings of the 7th Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA '92)*, Vancouver, Canada.

[Johnson, 1993] Johnson, R. E. (1993). How to Design Frameworks. Tutorial Notes for the 1993 Conference on Object Oriented Programming, Systems, Languages and Systems (OOPSLA '93).

[Johnson, 1997] Johnson, R. E. (1997). Components, Frameworks, Patterns. In *Proceedings of the 1997 symposium on Software reusability*.

[Johnson and Foote, 1988] Johnson, R. E. and Foote, J. (1988). Designing Reusable Classes. *Journal of Object Oriented Programming*, 1(2):22–35.

[Kahn, 1974] Kahn, G. (1974). The semantics of a simple language for parallel programming. *Information Processing*, pages 471–475.

[Karjalainen, 1999] Karjalainen, M. (1999). Immersion and Content - A Framework for Audio Research. In *Proceedings of the 1999 IEEE Workshop of Applications of Signal Processing to Audio and Acoustics*. IEEE.

[Kay, 1993] Kay, A. (1993). The Early History of Smalltalk. In *Proceedings of 2nd ACM SIGPLAN History of Programming Languages Conference*, volume 28 of ACM SIGPLAN Notices, pages 69–75.

[Kodish, 1993] Kodish, B. (1993). Getting off hayakawa's ladder. General Semantics Bulletin, (57):65–76.

[Kruchten, 2000] Kruchten, P. (2000). *The Rational Unified Process: An Introduction*. Addison-Wesley, second edition.

[Kuhn, 1962] Kuhn, T. (1962). *The Structure of Scientific Revolutions*. University of Chicaho Press.

[Landis and Niklasson, 1995] Landis, N. and Niklasson, A. (1995). Development of Object-Oriented Frameworks. Master's thesis, Lund University.

[Lanski, 1990] Lanski, P. (1990). The architecture and musical logic and cmix. In *Proceedings of the 1990 International Computer Music Conference (ICMC 90)*.

[Larman, 2002] Larman, C. (2002). *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*. Prentice-Hall, second edition.

[Lassfolk, 1995] Lassfolk, K. (1995). Sound processing kit. In *Proceedings of the 1995 International Computer Music Conference (ICMC 95)*.

[Law and Kelton, 2000] Law, A. M. and Kelton, W. D. (2000). *Simulation Modeling and Analysis*. McGrawHill, 3rd edition.

[Lazzarini, 1998] Lazzarini, V. (1998). A Proposed Design for an Audio Processing System. *Organized Sound*.

[Lazzarini, 2000a] Lazzarini, V. (2000a). Some Applications of the SndObj Library. In *Proceedings of the VII Brazilian Computer Music Symposium*.

[Lazzarini, 2000b] Lazzarini, V. (2000b). The Sound Object Library. *Organized Sound*, 5(1):35–49.

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

[Lazzarini and Accorsi, 1998] Lazzarini, V. and Accorsi, F. (1998). Designing a Sound Object Library. In *Proceedings of the V Brazilian Computer Music Symposium*.

[Lazzaro and Wawrzynek, 2001] Lazzaro, J. and Wawrzynek, J. (2001). Compiling MPEG 4 Structured Audio into C. In *Proceedings of the Second IEEE MPEG-4 Workshop and Exhibition*, San Jose, California.

[Lee and Park, 1995] Lee, E. and Park, T. (1995). Dataflow Process Networks. In *Proceedings of the IEEE*, volume 83, pages 773–799.

[Lindermann, 1991] Lindermann, E. (1991). ANIMAL - a Rapid Prototyping Environment for Computer Music Systems. *Computer Music Journal*, 15(3):78–100.

[Lindsay and Kriechbaum, 1999] Lindsay, A. and Kriechbaum, W. (1999). There's More Than One Way to Hear It: Multiple Representations of Music in MPEG-7. *Journal of New Music Research*.

[Liu et al., 2004] Liu, J., Eker, J., Janneck, J. W., Liu, X., and Lee, E. A. (2004). Actor-oriented Control System Design: A Responsible Framework Perspective. *IEEE Transactions on Control System Technology*, 12(2).

[Makhoul, 1975] Makhoul, J. (1975). Linear Prediction: A Tutorial Review. In *Proceedings of the IEEE*, volume 63, pages 561–580.

[Manjunath et al., 2002] Manjunath, B., Salembier, P., and Sikora, T., editors (2002). *Introduction to MPEG 7: Multimedia Content Description Language*. John Wiley and Sons, Ltd, West Sussex, England.

[Manolescu, 1997] Manolescu, D. A. (1997). A Dataflow Pattern Language. In *Proceedings of the 4th Pattern Languages of Programming Conference*.

[Martínez, 2002] Martínez, J. (2002). Overview of MPEG-7 Standard (version 5.0). Technical Report ISO/IEC JTC1/SC29/WG11 N4031, MPEG. available at http://www.cselt.it/mpeg/standards/mpeg-7/mpeg-7.htm.

[Mathews and Pasquale, 1981] Mathews, M. and Pasquale, J. (1981). RTSKED, a Scheduled Performance Language for the Crumar General Development System. In *Proceedings of the 1981 International Computer Music Conference (ICMC '81)*, page 286.

[Mathews, 1969] Mathews, M. V. (1969). *The Technology of Computer Music*. MIT Press.

[Mc Millen, 1994] Mc Millen, K. (1994). ZIPI: Origins and Motivations. *Computer Music Journal*, 18(4):48–96.

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

[McReynolds et al., 1999] McReynolds, D., Duggins, S., Galli, D., and Mayer, J. (1999). Distributed Characteristics of Subject Oriented Programming: An Evaluation with the Process and Object Paradigms. In *Proceedings of the 1999 ACM Southeast Regional Conference*.

[Mead, 1910] Mead, G. H. (1910). Social consciousness and the consciousness of meaning. *P*sychological Bulletin, (7):397–405.

[Meequel et al., 1997] Meequel, J., Horton, T. B., France, R. B., Mellone, C., and Dalvi, S. (1997). From Domain Models to Architecture Frameworks. *ACM SIGSOFT Software Engineering Notes*, 22(3).

[Meller et al., 2003] Meller, S. J., Clark, A. M., and Futagami, T. (2003). Model Driven Development. *IEEE Software*.

[Mellinger et al., 1991] Mellinger, D. K., Garnett, G. E., and Mont-Reynaud, B. (1991). *The Well-tempered Object. Musical Applications of Object-Oriented Software Technology*, chapter Virtual Digital Signal Processing in an Object-Oriented System, pages 188–194. MIT Press.

[Meunier, 1995] Meunier, R. (1995). The Pipes and Filter Architecture. In Coplien, J. O. and Schmidt, D. C., editors, *Pattern Languages of Program Design*, volume vol.1, chapter 22. Addison-Wesley.

[Microsystems,] Microsystems, S. How to write doc comments for the javadoc tool. Published online at java.sun.com/j2se/javadoc/writingdoccomments.

[Mili et al., 1995] Mili, H., Pachet, F., Benyahia, I., and Eddy, F. (1995). Metamodeling in OO. OOPSLA '95 Workshop summary.

[MMA, 1998] MMA (1998). *MIDI 1.0 Detailed Specification*. MIDI Manufacturers Association, Los Angeles.

[Moore et al., 1997] Moore, B., Glasberg, B., and Baer, T. (1997). A Model for the Prediction of Thresholds, Loudness, and Partial Loudness. *Journal of the Audio Engineering Society*, 45(4):224–240.

[Moser and Nierstrasz, 1996] Moser, S. and Nierstrasz, O. (1996). The Effect of Object-Oriented Frameworks on Developer Productivity. *IEEE Computer*, pages 45–51.

[Murata, 1989] Murata, T. (1989). Petri Nets: Properties, Analysis and Applications. In *Proceedings of the IEEE*, volume 77.

[Nakatani and Okuno, 1998] Nakatani, T. and Okuno, H. (1998). Sound Ontology for Computational Auditory Scene Analysis. In *Proceeding for the 1998 conference of the American Association for Artificial Intelligence*.

[Nelson, 1994] Nelson, C. (1994). A Forum for Fitting the Task. *IEEE Computer*, 27(3):104.

[NeXT, 1990] NeXT (1990). *ScoreFile Language Reference. Release 2.0*. NeXT Computer Inc.

[Niu et al., 2004] Niu, J., Zou, J., and Ren, A. (2004). Oopn: An object-oriented petri nets and its integrated environment. Technical report, City University of New York.

[Noble et al., 2002] Noble, J., Biddle, R., and Tempero, E. (2002). Metaphor and Metonymy in Object-Oriented Design Patterns. In *Proceedings of the 25th Australasian Computer Science Conference (ASC2002)*, Melbourne, Australia.

[Nygaard, 1986] Nygaard, K. (1986). Basic Concepts in OO Programming. In *Proceedings of the 1986 SIGPLAN workshop on Object-oriented programming*, pages 128–132.

[Nygaard, 2001] Nygaard, K. (2001). OO is Easy to Learn but Seldom Taught. Invited talk in the 2001 Conference on Object Oriented Programming, Systems, Languages and Applications, OOPSLA 01.

[Nygaard and Dahl, 1978] Nygaard, K. and Dahl, O. (1978). The Development of the Simula Languages. In *Proceedings of the 1st Conference on the History of Programming Languages*.

[Ogden and Richards, 1946] Ogden, C. K. and Richards, I. A. (1946). *The Meaning of Meaning*. Harcourt, Brace & World, New York.

[OMG, 2003] OMG (2003). Unified Modeling Language (UML) Specification: Infrastructure, version 2.0.

[Opdyke, 1992] Opdyke, W. (1992). *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign.

[Opdyke and Johnson, 1990] Opdyke, W. and Johnson, R. (1990). Refactoring, an Aid in Designing Application Frameworks and Evolving Object-oriented Systems. In *Proceeding of Symposium on Object Oriented Programming Emphasizing Practical Applications (SOOPA)*.

[Openheim and Willsky, 1997] Openheim, A. V. and Willsky, A. S. (1997). *Signals and Systems*. Prentice Hall, second edition.

[O'Riordan, 2002] O'Riordan, M. (2002). Technical Report on C++ Performance. Technical Report N1396, ISO/IEC JTC1/SC22/WG21.

[Oswood, 1976] Oswood, C. E. (1976). *Focus on Meaning Volume 1: Explorations in Semantic Space*. Moulon, The Hague.

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

[Peeters et al., 2000] Peeters, G., McAdams, S., and Herrera, P. (2000). Instrument Sound Description in the Context of MPEG-7. In *Proceedings of the 2000 International Computer Music Conference*, San Francisco. Computer Music Association.

[Petri, 1962] Petri, C. (1962). *Kommunikation mit Automaten*. PhD thesis, Technische Universitat Darmstadt, Germany.

[Petters et al., 1999] Petters, G., Herrera, P., and Amatriain, X. (1999). Audio CE for Instrument Description (Timbre Similarity). Technical report, MPEG.

[Pfeiffer, 1999] Pfeiffer, S. (1999). The Importance of Perceptive Adaptation of Sound Features in Audio Content Processing. In *SPIE Storage and Retrieval for Image and Video Databases VII*, pages 328–337, San Jose, California, USA.

[Philipsen, 1995] Philipsen, G. (1995). *W*atershed Research Traditions in Human Communications Theory, chapter The Coordinated Management of Meaning Theory of Pearce, Cronenn and Associates. State University of New York Press.

[Pope, 1991a] Pope, S., editor (1991a). *The Well-tempered object, Musical Applications of Object-Oriented Technology*. MIT Press.

[Pope, 1998a] Pope, S. (1998a). The Siren Music/Sound Package for Squeak Smalltalk. In *Proceedings of the 1998 Conference on Object Oriented Programming, Systems and Application (OOPSLA 98)(Companion Material)*.

[Pope et al., 2001] Pope, S., Engberg, A., Holm, F., and Wolf, A. (2001). The Distributed Processing Environment for High-Performance Distributed Multimedia Applications. In *Proceedings of the 2001 IEEE Multimedia Technology and Applications Conference*.

[Pope, 1987] Pope, S. T. (1987). A Smalltalk-80-based Music Toolkit. In *Proceedings of the 1987 International Computer Music Conference (ICMC '87)*. Computer Music Association. Also in Journal of Object-Oriented Programming 1(1): 6-14.

[Pope, 1991b] Pope, S. T. (1991b). Object-Oriented Design Elements in the MODE System. *Journal of Object-Oriented Programming*.

[Pope, 1991c] Pope, S. T. (1991c). *The Well-tempered Object. Musical Applications of Object-Oriented Software Technology*, chapter Introduction to MODE: The Musical Object Development Environment, pages 83–106. MIT Press.

[Pope, 1991d] Pope, S. T. (1991d). *The well-tempered Object. Musical Applications of Object-Oriented Software Technology*, chapter Machine Tongues XI: Object-Oriented Software Design, pages 32–48. MIT Press.

[Pope, 1992] Pope, S. T. (1992). The SmOKe Music Representation, Description Language, and Interchange Format. In *Proceedings of the 1992 International Computer Music Conference (ICMC '92)*. Computer Music Association. Also in Journal of Object-Oriented Programming 1(1): 6-14.

[Pope, 1994] Pope, S. T. (1994). The Musical Object Development Environment: MODE (Ten Years of Music Software in Smalltalk). In *Proceedings of the 1994 International Computer Music Conference (ICMC94)*. Computer Music Association.

[Pope, 1997] Pope, S. T. (1997). *Musical Signal*, chapter Object-oriented Music Representation. Swets and Zeitlinger.

[Pope, 1998b] Pope, S. T. (1998b). Modeling Musical Structures as EventGenerators. In *Proceedings of the 1998 International Computer Music Conference (ICMC98)*. Computer Music Association.

[Pope, 2001] Pope, S. T. (2001). *Squeak: Open Personal Computing and Multimedia*, chapter Music and Sound Processing in Squeak Using Siren. Prentice Hall.

[Pope, 2003] Pope, S. T. (2003). Recent Developments in Siren: Modeling, Control and Interaction for Large-scale Distributed Music Software. In *Proceedings of the 2003 International Computer Music Conference (ICMC '03)*. Computer Music Association. Also in Journal of Object-Oriented Programming 1(1): 6-14.

[Pope, 2004] Pope, S. T. (2004). Sound and Music Processing in SuperCollider. Unpublished book draft.

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

[Posnak et al., 1996] Posnak, E. J., Lavender, R. G., and Vin, H. M. (1996). Adaptive pipeline: an object structural pattern for adaptive applications. In *Proceedings of the 3rd Pattern Languages of Programming Conference*, Monticello, Illinois.

[Puckette, 1988] Puckette, M. (1988). The Patcher. In *Proceedings of the 1988 International Music Conference (ICMC '88)*, pages 420–429.

[Puckette, 1991a] Puckette, M. (1991a). Combining Event and Signal Processing in the MAX Graphical Programming Environment. *Computer Music Journal*.

[Puckette, 1991b] Puckette, M. (1991b). FTS: A Real-time Monitor for Multiprocessor Music Synthesis. *Computer Music Journal*, 15(3):58–67.

[Puckette, 1996] Puckette, M. (1996). Pure Data: Another Integrated Computer Music Environment. In *Proceedings of the Second Intercollege Computer Music Concerts*, pages 37–41, Tachikawa.

[Puckette, 1997a] Puckette, M. (1997a). Pure Data. In *Proceedings of the 1997 International Music Conference (ICMC '97)*, pages 224–227. Computer Music Association.

[Puckette, 1997b] Puckette, M. (1997b). Pure Data: Recent Progress. In *Proceedings of the Third Intercollege Computer Music Festiva*, pages 1–4, Tokyo, Japan.

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

[Puckette, 2004] Puckette, M. (2004). *Pd Documentation*.

[Puckette et al., 1998] Puckette, M., Apel, T., and Zicarelli, D. (1998). Real-time Audio Analysis Tools for Pd and MSP. In *Proceedings of the 1998 International Music Conference (ICMC '98)*. Computer Music Association.

[Purnhagen and Meine, 2000] Purnhagen, H. and Meine, N. (2000). Hiln - the mpeg-4 parametric audio coding tools. In *Proceedings of ISCAS 2000*.

[Reggio, 2002] Reggio, G. (2002). Metamodeling Behavioural Aspects: the Case of UML State Machines. *Integrated Design and Process Technology*.

[Riddell and Bencina, 1996] Riddell, A. and Bencina, R. (1996). Cmix on non-unix platforms. In *Proceedings of the 1996 International Computer Music Conference (ICMC 96)*.

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

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

[Rodet and Cointe, 1984] Rodet, X. and Cointe, P. (1984). FORMES: Composition and Scheduling of Processes. *Computer Music Journal*, 8(3).

[Rodet and Cointe, 1991] Rodet, X. and Cointe, P. (1991). *The well-tempered Object. Musical Applications of Object-Oriented Software Technology*, chapter FORMES: Composition and Scheduling of Processes, pages 64–82. MIT Press.

[Rovan et al., 1997] Rovan, J. B., Wanderlay, M., Dubnov, S., and Depalle, P. (1997). Instrumental Gestural Mapping Strategies as Expressivity Determinants in Computer Music Performance. In *Proceedings of Kansei- The Technology of Emotion Workshop*, Genova.

[Rowe, 1965] Rowe, W. (1965). Why System Science and Cybernetics? *IEEE Transactions on Systems and Cybernetics*, 1:2–3.

[Sarria and Diago, 2003] Sarria, G. and Diago, J. (2003). OpenMusic for Linux and MacOS X. Technical report.

[Scaletti, 1991] Scaletti, C. (1991). *The Well-tempered Object. Musical Applications of Object-Oriented Software Technology*, chapter The Kyma/Platypus Computer Music Workstation, pages 119–140. MIT Press.

[Scaletti, 2002] Scaletti, C. (2002). Computer Music Languages, Kyma, and the Future. *Computer Music Journal*, 26(4):69–82.

[Scaletti and Hebel, 1991] Scaletti, C. and Hebel, K. (1991). *Representations of Musical Signals*, chapter An Object-based Representation for Digital Audio Signals. MIT Press.

[Scaletti and Johnson, 1988] Scaletti, C. and Johnson, R. E. (1988). An Interactive Environment for Object-Oriented Music Composition and Sound Synthesis. In *Proceedings of the 1988 Conference on Objec-Oriented Programming, Systems, Languages, and Applications (OOPSLA'88)*, pages 25–30.

[Scavone, 2002] Scavone, G. (2002). RtAudio: A Cross-Platform C++ Class for Realtime Audio Input/Output. In *P*roceedings of the 2002 International Computer Music (ICMC'02).

[Schaeffer, 1966] Schaeffer, P. (1966). *Traité des Objets Musicaux*. Editions Du Seuil.

[Scheirer, 1998a] Scheirer, E. (1998a). The MPEG-4 Structured Audio Orchestra Language. In *Proceeding for the 1998 International Computer Music Conference (ICMC 98)*, San Francisco. Computer Music Association.

[Scheirer, 1998b] Scheirer, E. (1998b). The MPEG-4 Structured Audio Standard. In *Proceedings of the IEEE ICASSP 1998*, Seattle, Washington, USA.

[Scheirer, 1999a] Scheirer, E. (1999a). AudioBIFS: Describing Audio Scenes with the MPEG-4 Multimedia Standard. *IEEE Transactions on Multimedia*, 1(3):237–250.

[Scheirer, 1999b] Scheirer, E. (1999b). SAOL: the MPEG-4 Structured Audio Orchestra Language. *Computer Music Journal*, 23(2):31–51.

[Scheirer, 1999c] Scheirer, E. (1999c). Structured Audio and Effects Processing in the MPEG-4 Multimedia Standard. *Multimedia Systems*, (7):11–22.

[Scheirer, 2000] Scheirer, E. (2000). *Music Listening Systems*. PhD thesis, Massachusets Institute of Technology (MIT).

[Scheirer, 2001] Scheirer, E. (2001). Structured Audio, Kolmogorov Complexity, and Generalized Audio Coding. *IEEE Transactions on Speech and Audio Processing*, 9(8).

[Scheirer and Kim, 1999] Scheirer, E. and Kim, Y. (1999). Generalized Audio Coding with MPEG-4 Structured Audio. In *Proceedints of the AES 17th Audio Conference on High Quality Audio Coding*, Villa Castelletti Signa, Italy.

[Scheirer et al., 2000] Scheirer, E., Lee, Y., and Yang, J. (2000). Synthetic and SNHC audio in MPEG-4. *Signal Processing: Image Communication*, (15):445–461.

[Scheirer et al., 1998] Scheirer, E., Väänänen, R., and Huopaniemi, J. (1998). AudioBIFS: The MPEG-4 Standard for Effects Processing. In *Proceedings of the first Digital Audio Effects Workshop (DAFX '98)*, Barcelona.

[Schoner et al., 1998] Schoner, B., Cooper, C., Douglas, C., and Gershenfeld, N. (1998). Data-driven Modeling and Synthesis of Acoustical Instruments. In *Proceedings of the 1998 International Computer Music Conference (ICMC 98)*, San Francisco. Computer Music Association.

[Schottstaedt, 2004] Schottstaedt, B. (2004). *S*nd Sound Editor Manual. Available at http://ccrma.stanford.edu/software/snd/snd/snd.html.

[Schottstaedt, 2000] Schottstaedt, W. (2000). *Common Lisp Music Documentation.* CCRMA-Stanford University, http://www-ccrma-stanford.edu/software/clm.

[Schwarz and Wright, 2000] Schwarz, D. and Wright, M. (2000). Extensions and Applications of the SDIF Sound Description Interchange Format. In *Proceedings of the 2000 International Computer Music Conference (ICMC '00).*

[Seidewitz, 2003] Seidewitz, E. (2003). What Models Mean. *IEEE Software.*

[Seppänen and Kananoja, 1998a] Seppänen, J. and Kananoja, S. (1998a). *Sonic Flow: A Program for the Design and Simulation of Audio Signal Processing Networks. Functional Specification.*

[Seppänen and Kananoja, 1998b] Seppänen, J. and Kananoja, S. (1998b). *Sonic Flow: A Program for the Design and Simulation of Audio Signal Processing Networks. Technical Specification.*

[Serra, 1989] Serra, X. (1989). *A System for Sound Analysis/Transformation/Synthesis based on a Deterministic plus Stochastic Decomposition.* PhD thesis, Stanford University.

[Serra, 1990] Serra, X. (1990). Spectral Modeling Synthesis: A Sound Analysis/Synthesis System based on a Deterministic plus Stochastic Decomposition. *Computer Music Journal*, 14(4):12–24.

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

[Serra and Bonada, 1998] Serra, X. and Bonada, J. (1998). Sound Transformations Based on the SMS High Level Attributes. In *Proceedings of the 1st International Conference on Digital Audio Effects (DAFX98)*, Barcelona, Spain.

[Shannon and Weaver, 1949] Shannon, C. and Weaver, W. (1949). *The Mathematical Theory of Communication.* University of Illinois Press, Urbana.

[Shaw, 1996] Shaw, M. (1996). Some Patterns for Software Architecture. In Vlissides, J. M., Coplien, J. O., and Kerth, N. L., editors, *Pattern Languages of Program Design*, volume vol.2, chapter 16. Addison-Wesley.

[Shreiner, 2004] Shreiner, D., editor (2004). *OpenGL 1.4 Reference Manual 4th Edition.* Addison Wesley Professional.

[Simon, 1996] Simon, H. A. (1996). *The Sciences of the Artificial.* MIT Press, 3erd edition edition.

[Snyder and Oswood, 1967] Snyder, J. G. and Oswood, C. E., editors (1967). *Semantic Differential Technique.* Aldine, Chicago.

[Solà, 1997] Solà, J. (1997). Disseny i Implementació d'un Sintetitzador de Piano. Master's thesis, Universitat Politècnica de Catalunya (UPC).

[Strom, 1986] Strom, R. (1986). A Comparison of the Object-Oriented and Process Paradigm. In *Sigplan Notices*, volume 21.

[Stroustrup, 1995] Stroustrup, B. (1995). Why C++ is not only an Object-Oriented Programming Language. In *Proceedings of the 1995 Conference on Object Oriented Programming, Systems and Languages (OOPSLA 95).*

[Stroustrup, 1997] Stroustrup, B. (1997). *The C++ Programming Language*. Addison Wesley, special edition edition.

[Taligent, 1994] Taligent (1994). Building Object-Oriented Frameworks, A Taligent White Paper. Technical report, Taligent Inc.

[Taube, 1990] Taube, H. (1990). Common Music:A Music Composition Language in Common Lisp and CLOS. *Computer Music Journal*, 15(2).

[Taube, 1998] Taube, H. (1998). Introduction to Common Music. *Computer Music Journal*, 18(4):48–96.

[Temperley, 2004] Temperley, D. (2004). *T*he Cognition of Basic Musical Structures. MIT Press.

[Todoroff, 2002] Todoroff, T. (2002). *DAFX: Digital Audio Effects*, chapter Control of Digital Audio Effects. John Wiley and Sons, Ltd.

[Tolonen, 2000] Tolonen, T. (2000). Object-Based Source Modeling for Musical Signals. In *Proceedings of the 109th Audio Engineering Society Convention*, Los Angeles.

[Tzanetakis, 2002] Tzanetakis, G. (2002). *Manipulation, Analysis and Retrieval Systems for Audio Signals*. PhD thesis, Princeton University.

[Tzanetakis and Cook, 1999] Tzanetakis, G. and Cook, P. (1999). A Framework for Audio Analysis based on Classification and Temporal Segmentation. In *Proceedings of Euromicron 99, Workshop on Music Technology and Audio Processing*.

[Tzanetakis and Cook, 2000] Tzanetakis, G. and Cook, P. (2000). Marsyas: A Framework for Audio Analysis. *Organized Sound*, 4(3).

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

[van Dijk et al., 2002] van Dijk, H. W., Sips, H. J., and Deprettere, E. F. (2002). On Context-aware Process Networks. In *Proceedings of the International Symposium on Mobile Multimedia & Applications (MMSA 2002))*.

[Varró and Patarizca, 2002] Varró, D. and Patarizca, A. (2002). VPM: Mathematics of Metamodeling is Metamodeling Mathematics. *SoSyM Journal - Special section on UML*.

[Vercoe, 1992] Vercoe, B. L. (1992). *CSound . The CSound Manual Version 3.48. A Manual for the Audio Processing System and supporting program with Tutorials*.

[Vercoe et al., 1998] Vercoe, B. L., Gardner, W. G., and Scheirer, E. (1998). Structured Audio: Creation, Transmission, and Rendering of Parametric Sound Representations. In *Proceedings of the IEEE*, volume 86.

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

[W3C, 1999] W3C (1999). World Wide Web Consortium (W3C)'s XSL Transformations (XSLT) Version 1.0. www.w3.org/TR/xslt.

[Wanderley and Battier, 2000] Wanderley, M. and Battier, M., editors (2000). *Trends in Gestural Control of Music*. Ircam, Paris.

[Webb et al., 1999] Webb, D., Wendelborn, A., and Maciunas, K. (1999). Process Networks as Higher-level Notation for Metacomputing.

[Weinand et al., 1989] Weinand, A., Gamma, E., and Marty, R. (1989). Design and Implementation of ET++, a Seamless Object-Oriented Application Framework. *Structured Programming*, 10(2).

[Wessel, 1979] Wessel, D. (1979). Timbre Space as a Musical Control Structure. *Computer Music Journal*, 2(3).

[West, 2002] West, D. (2002). Metaphor, Architecture and XP. In *Proceedings of the 2002 XP Conference*.

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

[Wright, 1998b] Wright, M. (1998b). New applications of the sound description interchange format. In *Proceedings of the 1998 International Computer Music Conference (ICMC98)*.

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

[www-Agnula,] www-Agnula. AGNULA (A GNU Linux Audio Distribution) homepage, http://www.agnula.org.

[www-AudioMulch,] www-AudioMulch. AudioMulch webpage: http://www.audiomulch.com.

[www-BarryVercoe,] www-BarryVercoe. Professor Barry Vercoe's home page. http://web.media.mit.edu/ bv.

[www-CLAM,] www-CLAM. CLAM website: http://www.iua.upf.es/mtg/clam.

[www-Doxygen,] www-Doxygen. Doxygen documentation system homepage: www.doxygen.org.

[www-FFTW,] www-FFTW. Faster fourier transform of the west (fftw) homepage: http://www.fftw.org.

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

[www-id3lib,] www-id3lib. The id3 tagging library homepage: http://id3lib.sourceforge.net/.

[www-JaffeMusicKit,] www-JaffeMusicKit. David jaffe's musickit site: www.jaffe.com/mk97.html.

[www-libsndfile,] www-libsndfile.

[www-Mantis,] www-Mantis. Mantis bug-tracker homepage: www.mantisbt.org.

[www-Metamodel,] www-Metamodel. Community site for meta-modeling and semantic modeling. http://www.metamodel.com.

[www-MTG,] www-MTG. Homepage of the Music Technology Group (MTG) from the Universitat Pompeu Fabra.

[www-MusicKit,] www-MusicKit. The music kit homepage at sourceforge: www.sourceforge.net/projects/musickit.

[www-OSW,] www-OSW. Open Sound World (OSW) webpage: http://www.cnmat.berkeley.edu/OSW.

[www-PD,] www-PD. Pure Data (PD) homepage: http://www.pure-data.org/.

[www-PortAudio,] www-PortAudio. PortAudio homepage: www.portaudio.com.

[www-PortMIDI,] www-PortMIDI. Port Music homepage: http://www-2.cs.cmu.edu/ music/portmusic/.

[www-Ptolemy,] www-Ptolemy. Ptolemy project home page. http://ptolemy.eecs.berkely.edu.

[www-QT,] www-QT. Qt homepage by trolltech. http://www.trolltech.com.

[www-RtAudio,] www-RtAudio. RtAudio homepage: www.music.mcgill.ca/ gary/rtaudio.

[www-Siren,] www-Siren. Siren Webpage: http://www.create.ucsb.edu/Siren.

[www-SndObj,] www-SndObj. SndObj library homepage: http://www.may.ie/academic/music/musictec/SndObj/main.

[www-SoundsLogical,] www-SoundsLogical. Sounds Logical (makers of WaveWarp) webpage: http://www.soundslogical.com.

[www-SymbolicSound,] www-SymbolicSound. Symbolic Sound Corporation (creators of Kyma) webpage: http://www.symbolicsound.com.

[www-XML,] www-XML. World wide web consortium (w3c)'s xml homepage: http://www.w3.org/xml/.

[www-XMLMusic,] www-XMLMusic. Xml and music: http://xml.coverpages.org/xmlmusic.html.

[www-XMLSchema,] www-XMLSchema. World Wide Web Consortium (W3C)'s XML-Schema homepage, http://www.w3.org/XML/Schema.

[Xercesc,] Xercesc. Xerces c++ parser homepage: http://xml.apache.org/xerces-c.

[Zicarelli, 2002] Zicarelli, D. (2002). How I Learned to Love a Program that Does Nothing. *Computer Music Journal*, 26(4):44–51.

APPENDIX A

CLAM Additional Information

§A.1 A brief history of the Framework

In this section we will give a brief overview of the CLAM development process. It must be pointed out that this overview does not assume that the process has ended. As a matter of fact, an open framework such as CLAM will probably never be completely finished. Although at the time of this writing current release is still 0.5.x, which actually means that it is a bet non-stable stage, release 1.0 is not very far away and the framework has already reached a mature stage and the pending issues to make it more stable are clearly identified and are being addressed (see A.1.5).

§A.1.1 How it all started

CLAM was started in October 2000. At that time, it had become clear that some sort of organization had to be introduced to the source code that was being generated at the Music Technology Group [www-MTG,]. The SMSTools, application that had become some sort of flagship of the group[Bonada, 1997], was the result of the work of a single person and not much care had been put into applying any sort of strategy that would enable maintainability or re-usability. Furthermore, the application had been developed exclusively for the Microsoft Windows platform. On the other hand, new projects were being started all of the time. Many of them saw the potential benefits of reusing code and algorithms that were in the SMS code. But the learning curve to understand the code and internal organization was unbearable. This is even more if we take into account that the code had also been intensively optimized for real-time processing, compromising understandability and maintainability to speed, many times with misconceptions.

In this context, the idea of starting the CLAM project was born. The original name was MTG-Classes, name that already reflects two of the original restrictions (which latter were suppressed): MTG, thus the project was to be an internal project, and Classes, thus the initial intention was not to build a framework but rather a class library. The original goal, as quoted from the first draft, was:

"To offer a complete, flexible and platform independent Sound Analysis/Synthesis C++ platform to meet current and future needs of all MTG projects."

The three main axes of this goal were defined as (from CLAM first draft document):

- Complete: Should include all utilities needed in a Sound Processing Project (input/output, processing, storage, display...)
- Flexible: Easy to use and adapt to any kind of need.
- Platform Independent: Compile under UNIX, Windows and Mac platforms.

Note how similar these main objectives are to those mentioned in [Taligent, 1994] where they state that to be successful, you should design your framework to be complete, flexible, extensible, and understandable.

These initial objectives have slightly changed since then mainly to accommodate to the fact that the library is no longer seen as an internal tool for the MTG but as a library that is made public under the GNU-GPL in the course of the AGNULA IST European Project. But the truth is that as a summary of the philosophy of the framework they are as valid as the first day.

When CLAM was started it was not seen as a framework but rather as a class library that would some day offer ready-to-use C++ classes that would be incorporated in any audio or music processing project. Because of this, in this first phase most effort was put forward in the development of useful signal processing and basic tools such as sound input/output. The implementation of some algorithms was put in the hand of signal processing engineers with hardly any knowledge in programming, object-orientation or C++.

Although some basic coding conventions had been defined, no real effort was put in defining a common infrastructure. The result was highly unstructured and hardly maintainable. All of the algorithms were encapsulated in classes but the classes interfaces often differed and the framework presented too many "hot spots"[Johnson and Foote, 1988, Roberts and Johnson, 1996].

This first phase should be understood as a exploratory phase in which the idea of a "class library" was studied and discarded as not sufficient: MTG-Classes should become a framework and even the name now seemed not appropriate.

About this same time it was clear that the development of the framework should be application-driven as recommended in [Johnson, 1993]. Three different applications were chosen for different reasons. SMSTools was at that time the flagship application of the group (see figure A.1) and had been, as already mentioned, the main reason for starting the development of the project. It was obvious that this application should be one of the chosen to drive the development. SMSTools is an off-line application for analyzing, transforming and re-synthesizing sounds. Another application was what we called the "dummy

test" (see Figure A.2). This application is a multi band spectral delay artificially designed for testing what were thought to be the most demanding signal processing related requirements: spectral domain real-time processing, multiple branch processing graph with variable delay and interacting graphical interface. The last application, Salto, did not exist at that time. It is a real-time spectral domain sax synthesizer designed to be played through a wind controller MIDI interface. (See section 3.2.3 for more details on these and other CLAM sample applications).

These applications introduced more focused requirements for both tools and architecture. New people joined the team and on this next phase many efforts were put on the development of a general infrastructure. The XML format was chosen as a general format for passivation in the framework. MIDI input was also implemented. But the biggest challenge of all was to define a general infrastructure for two basic concepts: Processing and Processing Data classes.

Following the idea of a white-box framework [Johnson and Foote, 1988] abstract classes should be offered so as to enforce a common interface. The base Processing[1] class was implemented and all encapsulated algorithms were forced to follow a common abstract interface (see section 3.2.2.1. As for the data infrastructure a big investment was made in order to have data classes with the following requirements: dynamically instantiable attributes, tree-like structure, introspection and passivation. For all these reasons Dynamic Types were implemented and have been one of the trademarks of the framework ever since (see section 3.2.2.2).

Soon after the project was born the idea of having two different modes related to the framework was discussed. The two modes were called supervised and non-supervised (names that will be kept to honor the history of the framework but that are admittedly ambiguous). In the supervised mode some entity of the system called Flow Control should be in charge of managing the whole data and control flow. The final idea was to create some sort of "visual builder" [Johnson and Foote, 1988] so the user could create rapid prototypes by just combining pre-existing blocks. In the non-supervised mode the user/programmer should be in charge of managing the system flow. The framework was in this sense understood as a combination of black-box and white-box (see section 1.3.3) where ready-to-use components were combined with code completely developed by the application builder.

Ever since the beginning and up until release 0.4, the development has concentrated in the non-supervised mode as it was the one that had the clearest requirements and was first needed, especially due to the example-driven development that had been imposed on the process. Even so, compatibility

[1]Note: the Processing class was called at that time ProcessingObject. This name reinforced the adjective usage of the word "processing". Nevertheless it was later shown to be an ambiguous name especially due to the fact that the word "object" was being used to name a class. Furthermore, if a Processing was to be a "ProcessingObject" then a Processing Data must have become a "ProcessingDataObject" and so on. The word "process" that was then discussed but not approved.

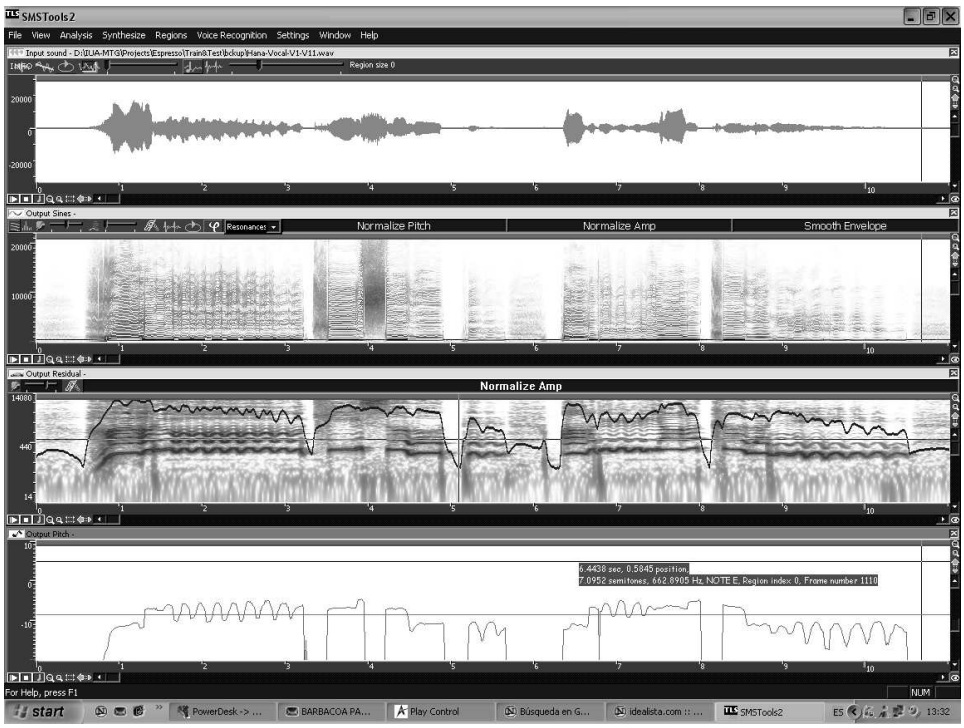

Figure A.1: Original SMSTools interface

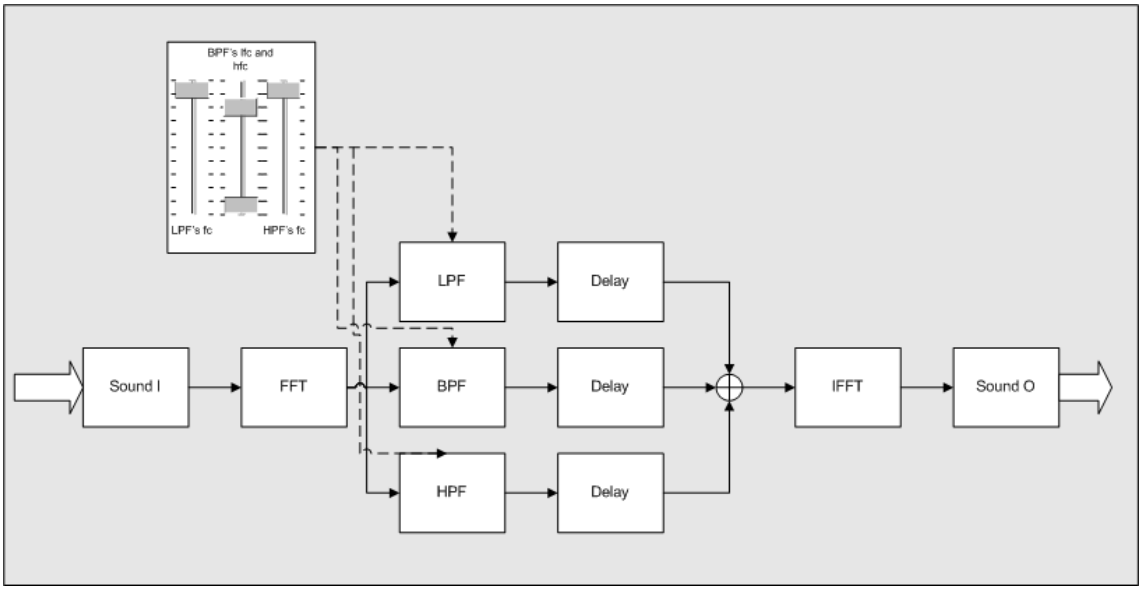

Figure A.2: Dummy test block diagram

with the supervised mode was intended in every new feature that was added: the functionality was not added but the code was ready to accept this addition with minor refactorings.

After the basic infrastructure in the framework was more or less useful, more internal users were convinced of the benefits of using the framework. Now the time investment needed to master CLAM was less than the time most of the developers would need to implement their application from scratch [Booch, 1994a]. It must be said that at that time (and still now) many of the framework users had very limited programming skills and with the help of the framework they were able to implement quite complex applications.

New projects joined the CLAM community. Especially important was the CUIDADO project, an IST European project focused on the analysis and extraction of semantic information from audio signals. This project made extensive use of the XML facilities of the framework in order to produce MPEG7 compliant descriptions [Manjunath et al., 2002]. But CUIDADO also contributed notably to the framework by improving and enhancing analysis algorithms and procedures.

Not only newly started projects decided to use CLAM but also some existing ones decided that it was worth to port the code to CLAM before continuing. The most notable case was that of the Time Machine project. Time Machine is a near-lossless time-stretching algorithm developed for the Yamaha company and included in their SOL software sequencer package. The algorithm was ported to CLAM, gaining in performance, efficiency and code understandability/re-usability.

§A.1.2 CLAM becomes public

The AGNULA IST project [www-Agnula,] started in April 2002 and represented a major contribution for the growth of the CLAM framework. This European "accompanying measure" aimed at providing two distributions of GNU/Linux focused on Audio and Multimedia. One distribution was to be based on Debian and called Demudi and the other based on RedHat and named Rehmudi. Both versions would be entirely made from Free Software (Free as defined by the Free Software Foundation [Free Software Foundation,]). CLAM was the main contribution by the Music Technology Group at the Pompeu Fabra University.

The inclusion of CLAM in such a project brought big and immediate changes into the general conception and philosophy of the framework. The most important change was that a framework that was being designed as an internal tool was to become public and accessible. This would introduce major changes into the development process that would also benefit internal users. Furthermore, CLAM was not only going to become public but also Free and distributed under the GPL license. And finally, although the initial targeted user was exclusively the researcher that needed to test/implement research algorithms now we had to also take into account the regular developer that chose our framework for simply developing and audio or music applications or even the user that was interested in working with one of the sample applications.

The first "public" release (Release 0.3) was published in April 2002. This release was published on the website [www-CLAM,] but was not publicly announced. It was used to test all the different mechanisms related to the public release and to beta-test the framework itself. A mirror of the internal CVS repository was made, and public access, with no commit rights, was granted. The release was tested by internal users and by students with no experience that were exposed to the framework and asked to develop a basic spectral analysis/synthesis application.

CLAM was indeed first publicly presented at the 2002 Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA '02). CLAM was presented both as a poster and a demonstration [Amatriain et al., 2002c, Amatriain et al., 2002a]. For this event different concepts were clarified and discussed but most importantly the sample applications were brought to a more stable stage. We demonstrated SMSTools, SALTO and SpectralDelay (see 3.2.3 for details on these applications).

After this presentation the first formal beta release (release 0.4) was publicly announced in the context of the AGNULA project. This release included, in relation to the previous, many bugfixes, a much better and larger documentation and a new directory structure for the code. The examples

demonstrated at the OOPSLA conference were also integrated into the public release.

The following minor releases introduced many other issues as the testing infrastructure using the cppunit framework. But most efforts were put into the SMSTools application, which was then called AnalysisSynthesisExample, in order to bring it to a usable stage. Because of this enhancements were pushed forward in the Processing repository (i.e. including new transformations based on the SMS algorithm) or the XML and Visualization infrastructure. The build system was also enhanced during this release.

§A.1.3 Release 0.5

Release 0.5 represented a step forward into the framework being more stable, robust, and usable. A new cross-platform build system was devised for making the process of creating a CLAM makefile or Visual project easy and transparent to the user, this tool is generically called Srcdeps. Formal and thorough testing was introduced through the use of he external CppUnit testing framework.

The *network mode* is implemented in about 90%, with a fully working Visual Builder (see definition of Visual Builder in section 1.3.3), called Network Editor, that allows to construct CLAM networks (see 3.2.3). It is becoming clearer every time that it makes no sense to make the distinction between the two modes. The usage of Ports and Networks (see 3.2.2.3) will become first the recommended, then the standard and finally the only way of building a CLAM system. The only difference that still exists and is meant to stay that way between the Visual Builder approach and the more traditional black/white-box approach is that in the former an automatic flow control and scheduling engine is deployed.

Long-sought full 100% support to Mac OSX has been introduced in this release, including support for platform abstraction such as audio input/output.

Other important issues include a major rework of the Visualization module and XML facilities and many new features introduced into the AnalysisSynthesisExample, which from this release on will be called SMS Example (or SMSTools in its GUI version). The implementation of Descriptors was addressed during this release although will not be finished until future releases. Finally many new simple usage examples were introduced and code cleanup was addressed throughout.

§A.1.4 Latest changes

Release 0.6 is the latest major release at the time of this writing. The major goal in this release has been to come up with a fully functional Network Editor promoting the use of automatic flow

control policies and dynamic networks. Many efforts have been put into enhancing CLAM's easiness of deployment by finally building binaries of the library.

Another important issue is the simplification of public interfaces of the main CLAM base classes. It has become clear that many of them were overdesigned and more features than needed were present. The goal is now to offer as simple interfaces as possible in order to make life easier to CLAM users in a white-box manner.

In this release we abandoned support to Microsoft Visual C++ 6.0 and all its non-standard features and turned to the much better compiler in Microsoft Visual .Net for the Microsoft Windows platform. We also added better support for the QT GUI framework, which will become the preferred framework from now on instead of the FLTK toolkit.

Up until this release, CLAM only accepted 16 bit wav, AIFF and raw sound files. With the inclusion of some external libraries, now we are able to accept virtually any PCM format, Vorbis OGG and MP3. See A.2.4 for a brief explanation on the external libraries that are used.

Finally, a major rework and testing was done on the Descriptors infrastructure and repository.

Release 0.6 is at the time of this writing still unfinished. Therefore many of the advances accomplished are very much related with the "CLAM Roadmap towards version 1.0" that will be commented in the next section.

§A.1.5 What is next?

CLAM is expected to very soon reach its maturity with release 1.0. In order to establish what are, at the time of this writing, the missing features a Roadmap towards CLAM 1.0 has been issued. What follows is a summary of this roadmap.

§A.1.5.1 Automated Flow

Although the automatic flow control and network infrastructure is almost fully implemented, a final effort needs to be done in this area. When this infrastructure is completely finished it will also be promoted as the preferred way of working and a few things will have to change in the Processing repository. The main actions that need to be taken or missing features are:

- Full-featured Network Editor: CLAM's Rappid Prototyping tool. Until now the Network Editor was only capable of including simple time-domain Processing objects. The goal is to be able to include all sort of Processing objects. More precisely, the Network Editor should be capable of reproducing the SMS Analysis/Synthesis Algorithm including intermediate transformations.

- Use of Automated Flow features in the Manual Mode. Many of the new features introduced in the Automated Flow will make the life of CLAM users easier even if they are using the Manual Flow. We intend to promote their usage.

- Use Ports. Instead of declaring intermediate data objects and passing them as arguments of the Do operation, it is much easier to use Ports to connect Processing objects in between them.

§A.1.5.2 CLAM deployment.

Now it is still difficult to install and use CLAM. We are heading toward a more usable environment by:

- Removing dependencies with third party libraries when not strictly necessary. Now, especially in GNU/Linux, you need to have all third party libraries installed in your system even though you are never going to use some of them. This issue should be solved in next releases.

- Devising a better package structure. Now we have very few namespaces/packages. The idea is to clearly identify more or less independent components and group them into understandable packages such as CLAM::Base, CLAM::ProcessingRepository, CLAM::SMS, etc...

- Distributing Binaries. The compile time and deployment would greatly benefit from having binary packages of the framework.

§A.1.5.3 Enforcing CLAM's framework good practices.

Another change related with the Processing infrastructure 3.2.2.1 is the clarification of a single way of doing things, removing accessory behavior that has not proven as useful as expected. In the first phases of the framework design, there were obvious problems of "over-design", which resulted in offering different ways of doing similar things. This flexibility promotes misuse and unnecessary complexity. There are now many Processing classes and in some cases some of them do similar things such as type checking or default initialization in different ways. A clear guideline in how all Processing objects should behave has been issued but now needs to be promoted into the whole repository.

- Cleaning up and clarifying of Processing interface. The goal is to remove all unnecessary interface and enforce the strict use of some design principles for all CLAM users developing their own Processing class.

- Cleaning up duplicated, old fashioned or unused code around CLAM and CLAM examples.

- Testing has been already introduced in the framework and is spreading at a steady state. Both unit tests and functional tests are compiled and run on a daily basis and help ensure the framework stability and confidence. Nevertheless, a lot has to be done in order to have a well-covered framework. The goal is to extend the test coverage to most of the CLAM code, and enhancing the test infrastructure in order to be easier for the processing writers to do their own testing.

§A.1.5.4 Clarifying Processing Data usage

Just as in the previous point, we have seen that most Processing Data classes offer a too complex interface.

- Making Dynamic only what needs to be Dynamic. Dynamic Types 3.2.2.2 have proven as a very useful mechanism and they offer many interesting services (such as automatic XML storage, homogeneous interface...) but its usage needs to be rationalized. The dynamic mechanism for adding and removing attributes at run-time is sometimes more annoying than useful. The overhead introduced by having the possibility of instantiating and de-instantiating attributes on run-time, for example, is seldom needed. On the other hand, some limitations in the Dynamic Types Structure, such as not being able to use inheritance, have already been solved and a new, improved version of these service is already developed. The problem is that introducing this change will break backward version compatibility in many ways, so the right moment is still to be chosen. The goal is to identify when and where DT are strictly necessary and only use them there.

- In some Processing Data classes it would be interesting to have inheritance capabilities. This is now limited because of the use of DT. The changes in Dynamic Types will clearly affect the way that CLAM Processing Data are structured in some ways. The fact that inheritance is introduced into Dynamic Types, and therefore into Processing Data, opens up the possibility of implementing a Processing Data hierarchy.

- Clarifying the use of the Spectrum class. The Spectrum class is very flexible but also too complex. Also by making some conversions transparent it is hiding from the user the fact that some practices are not efficient and should not be done inside a Do() operation. We may need to offer specialized spectrums which are efficient and explicit while keeping the current interface as a wrapper only to use in particular cases.

- To dB or not to dB. The use of linear/dB magnitudes in CLAM is not well solved so we need to devise a mechanism for taking care of this issue. This solution should aim at being both transparent and efficient.

§A.1.5.5 Descriptors infrastructure

Descriptors are a special kind of Processing Data (see 3.2.2.2) that have proven very useful and necessary for many applications mainly related to audio content processing. At this moment, CLAM offers a limited infrastructure for computing and using descriptors related to the Processing Data. In many of our internal project we have seen the need for a more exhaustive solution to this problem that will address issues such as its efficient computation, storage, representation, retrieval...

§A.1.5.6 Integrating result of CLAM related projects

CLAM has been used in a number of internal projects that have developed algorithms and applications. We are in the process of integrating their contributions into the public repository.

§A.1.5.7 Improved documentation

Current CLAM documentation has grown very large and difficult to maintain, we are heading towards:

- Converting the current manual into a smaller, more modular and conceptual documentation, moving out the interface details to the doxygen documentation.
- Making the doxygen documentation a more useful tool. Add modules grouping related classes, and improving the non-automatic (written) documentation in general.
- Promoting small tutorials.

The three kinds of documentation should provide enough links between them, so helping the user to find what she's looking for.

§A.1.5.8 Improved SMSTools

SMSTools is CLAM's flagship application. Although it is now more or less functional there is still much place left for improvements. We aim at having a better and more complete documentation including tutorials, a better interface (probably in QT), a better design of the overall application, and the possibility of managing workspaces and projects.

§A.1.5.9 Graphical interfaces

And finally, the visualization module in CLAM, together with the graphical interfaces in the examples should be enhanced. Although the preferred toolkit until now has been FLTK [www-FLTK,],

chances are that QT [www-QT,] will replace it soon because of its superior features and performance, albeit its license problems under the Windows platform.

§A.2 Used Tools and Resources

The cross-platform spirit of the CLAM framework has influenced and condition the selection of tools and the way they were used. We will first explain the development tools used such as compiler and concurrent version control systems and later explain the tools and resources that CLAM offers to its users.

CLAM is implemented in C++[Stroustrup, 1997]. The choice of the language was obvious at that time and during this time we have had nothing to make us thing it was not the correct decision. C++ was chosen for the following reasons:

(1) It was the language traditionally used in the group (as well as in most other projects related with signal processing)

(2) It is efficient

(3) It is object-oriented but also allows other paradigms to be integrated [Stroustrup, 1995].

(4) It allows low-level access to operating system, sound cards...

(5) There are plenty of tools and libraries that could be integrated in the framework

(6) It is a standard language (versus a proprietary one like Java) and is usually the language of choice for Free Software.

When the project started there were two obvious choices for C++ compiler support: gcc 2.95 on GNU/Linux and Microsoft Visual 6.0 on Windows. Portability to the Power PC platform was not an immediate need and, although some parts of the framework were compiled with CodeWarrior[2], we knew the best policy was to wait for the release of Mac OSX, a Unix-like operating system where portability was almost guaranteed.

From the very beginning many incompatibility issues between gcc and Visual C++ compiler had to be addressed. Microsoft Visual C++ 6.0 came with a compiler full of flaws and bugs and lacked support for some important parts of the C++ ISO standard such as the absence of template

[2]Most of the framework was compiled and tested under Mac OS but the lack of some system services like multithreading made it really difficult to port some real-time applications with guarantees.

partial specialization. Microsoft Visual C++ 6 also came with other major problems like an extremely bad implementation of the C++ Standard Template Library (STL). As a matter of fact, Microsoft's development environment was always the one that limited the development. Some solutions existed at that time, for instance using Intel's C++ compiler that turned out to be much more robust and compliant to the standard. But we could not make our development depend on yet another proprietary tool that is no that much extended in the development community.

As for gcc, some major problems had to be faced with version 2.96. This was an unstable version of the compiler but that made it through to the RedHat 7 GNU/Linux distribution. The main flaw in this implementation of the GNU compiler was that lacked support for dynamic casting. The CLAM framework could not do without some dynamic casts that were applied on polymorphic variables at crucial points of the code. For this reason, CLAM did not (and still does not) support gcc version 2.96. When version 3.X of the compiler were published, some minor changes had to be applied to the code, especially due to the increased strictness in some syntax rules. But the framework was soon compiling under this version. When CLAM was ported to Mac OSX and compiled with the gcc version there included no major problems were found either.

All these problems are specially true when it comes to the use of the most recent C++ features, such as templates, and related techniques, such as meta-programming and static dispatching. These techniques where initially considered as potentially useful in the CLAM framework, but the lack of language support in most compilers, together with the need of optimizing the compiling speed of the library, has led to a rather scarce use of them.

On the other hand, a technique considered obsolete as it is the use of C macros, has proven very useful to minimize programmer's effort and enable the implementation of rather complex behaviors (one of the good things of developing with a multi-paradigm language like C++ is that you can always find a more or less immediate workaround). Also, C macros are a simple compiler feature which is available in all C++ development platforms.

All code for CLAM is written in a collaborative manner and no code ownership is promoted. CVS [Fogel, 1999] was used from the very beginning although its usage has been improved and rationalized with every new release. First there was a single CVS branch. Later on, a second branch called "devel" was introduced. Major development was on this branch while the "main" branch was far more stable and only bugfixes were committed in between releases. When CLAM became public a new "public" repository was started. This repository was a mirror of the internal main-trunk except for the exclusion of folders marked as "private" (mainly drafts). It was updated on every new release or major bugfix using a semi-automatic script. Internal projects that used CLAM had their own repositories in

which they had some particular CLAM-related applications. In release 0.5 the public CVS repository was removed and substituted by tarballs that are automatically generated from the main branch of the internal repository. Nevertheless, public CVS access is planned when the framework becomes more stable (i.e. reaches the 1.0 version).

Other web-based tools are used in order to manage the framework. Mantis [www-Mantis,] is used for bug reporting and tracking. A public mailing list is also available at clam@iua.upf.es.

As for documentation, several approaches are followed. On one hand Doxygen [www-Doxygen,] is used to automatically generate html source code documentation from javadocs comments available in the source files [Microsystems,]. On the other hand an html document is maintained through CVS and published in the web in different formats including an automatically generated pdf file.

§A.2.1 The development team

In order to understand what CLAM has become it is important to understand how the development team has evolved over time since its initial configuration. Although in order to better understand the roles we will sometimes mention the main contributions of some collaborators it is important to understand that CLAM has always adopted a policy contrary to "code ownership". Therefore it is difficult to say that any one is exclusively responsible for one part of the framework.

Although CLAM's development team (clam-devel for short) has evolved over time it has always had an average of 5 developers. It must be pointed out though that, because of our situation inside an educational institution, it is seldom the case that a developer can dedicate full-time to CLAM, having other responsibilities related to other projects or to teaching/attending classes.

Only two members of the current development team (the author of this thesis and Maarten de Boer) were present in the initial team. Apart from developing many different components of the framework the author of this thesis has been responsible since then of coordinating and managing the team as well as designing the general working guidelines. Maarten de Boer has also contributed in many different parts of the framework but has been the main responsible for most of the Audio and MIDI input/output infrastructure as well as the build system.

This initial team had also three other members, (Emilia Gómez, Fabien Gouyon, and Joachim Haas) that did not have at that time many programming skills and were rather in charge of implementing signal processing algorithms. The first design decisions were basically taken by Maarten de Boer and the author but taking into account inputs from other researchers in the MTG, especially Jordi Bonada who is the developer of the original SMSTools.

The three original non-programmer members were soon replaced by new comers with their focus turning more into their research interests. The next three members of the CLAM development team were Pau Arumi, David Garcia and Enrique Robledo. Out of these three, the first two were interested in developing CLAM as part of their Master Thesis and they are still, at the time of this writing, members of the clam-devel team. Pau Arumi developed Dynamic Types and other low-level CLAM infrastructure for his Master Thesis while David Garcia was responsible for the XML infrastructure. Both of them have been involved in many different development issues since then. Enrique Robledo also contributed on the general infrastructure, especially in Processing and flow control, and developed the first real-time robust CLAM application called Rappid (see 3.2.3). Soon after another student, Miguel Ramirez, also came to do his Master Thesis in the context of the CLAM framework. He was the main responsible for the design of the Visualization Module and he is still an active member of the development team with contributions in many other different areas of the framework.

Other students have contributed to the framework, especially because of their Master Thesis. Out of these the ones that have at some time been part of the clam-devel team are the following: Xavier Rubio, who contributed on the development of the automatic flow and the Network Editor; Albert Mora, who contributed on the development of the SMSTools graphical interface; Merlinj Blaaw, who was in charge of developing the CLAM interface for developing VST plugins and also improved some efficiency issues; and Ismael Mosquera, who developed a voice to MIDI conversor and is currently in charge of graphical user interface issues.

§A.2.2 On methodologies

The truth is that when the CLAM project started not much effort was put into studying and applying specific framework development methodologies. It turns out though that most decisions have been taken in the right direction and the development process now seems to be in line with what most authors recommend. In this section we will briefly describe the methodologies used in the CLAM development.

As already commented, the development of the CLAM framework has followed a bottom-up approach, starting from some sample applications that have gradually evolve into being usage examples of the framework itself. It is important to note that in section 1.3.4 we already reflected how this is an important guideline given by most authors.

User feedback has always been an important component of the CLAM development process. As a matter of fact the framework was used ever since it began to be implemented and in many occasions,

the line between the framework developer and its user has been blurred (i.e. even the same person can be acting as a part-time framework developer and a part-time user).

The CLAM project started using quite traditional software engineering methodologies but has evolved and come closer to more agile methodologies like eXtreme Programming [Beck, 1999]. As even the proposers of the methodology admit though, it is difficult to apply XP to a framework development process. The main reason is that one of XP's fundamental principles, which is to promote simplicity and avoid unnecessary generalizations if not strictly necessary is hard to observe in a framework development process. When developing a framework we are not only interested in making an application work but also on generalizing or abstracting as much of the common behavior as possible in order to build an infrastructure for any further application. Furthermore, organizational issues, such as the fact that most developers in an educational institution are part-time collaborators, make it even harder.

But we are definitely interested in many of XP's objectives such as having a design that is as clean and simple as possible or the idea of continuous integration and thorough testing Therefore many XP practices have been considered and are used to some extent. These include:

- *Small releases*: although the public releases are still not published as often as possible internal iterations last 15 days.
- *Testing* through the use of the cppunit framework and Test-driven development.
- Refactoring
- *Collective ownership* using the CVS and allowing any developer to contribute in any part of the framework.
- *Code Standards* adapted to our own needs.

Other practices such as *Simple Design*, *Pair Programming* or *On-site Costumer* are used as much as possible in our particular limitations.

§A.2.3 The build system

CLAM provides a quite automated Build System that allows to generate and maintain, with little effort, GNU Makefiles and VisualC++ project files to build large volumes of source files. Note that this build system is specialized on building CLAM distribution binaries and CLAM based applications and although it could be surely be adapted to other kind of projects it has not been used outside the framework.

In any project it is difficult to deal with a complex source files dependency graph. Without a build system helper you have to add by hand each .cxx file into the Project IDE or build script in order to compile it. CLAM Build System is able to do this task with little supervision: *srcdeps* is a small and smart application that is able to deduce the source files that need to be compiled following the following simple rule:

'If main.cxx must be compiled and includes both blue.hxx and green.hxx

then blue.cxx and green.cxx must be also compiled'

This rule to is based on the hypothesis that for each header named Foo.hxx a Foo.cxx file exists somewhere, a thing that may or may not be true. Also there can be additional .cxx not related to any .hxx that should be compiled. In both cases, you can provide additional starting points to look for dependencies.

The CLAM build system is designed to be used to build CLAM examples, libraries and tests that are inside the CLAM source tree. But it also provides facilities to anybody building their own applications with CLAM. In order to use CLAM internal build system for building your App you must consider the source tree structure suggested in the documentation and use the following configuration files: defaults.cfg, settings.cfg, clam-location.cfg, system.cfg, and system-win.cfg or system-linux.cfg. Out of these, three need to be edited: the particular path to clam must be introduced into clam-location.cfg, the default global configuration variables must be edited into defaults.cfg and the settings related to the particular project must be entered into settings.cfg.

There are two main kinds of config variables depending on the values they may take: Boolean variables - these can only have values of 0 or 1. Usually 0 means that the variable effect is disabled, and 1 that it is enabled; and Textual variables - they are a string, for instance, a relative path to some file. Also depending on the effect, there are three kinds of variables: Build System variables - variables whose value just affects the CLAM build system behavior while naming binaries or searching for certain files; External Libraries variables - variables whose value determines whether the build system will make your application link or not to some (or any) of CLAM external libraries; and CLAM internal variables - these variables are mainly compile-time flags that activate/deactivate certain framework features or change some framework behavior.

Build system variables reference include the following:

TOP (Textual) - Should contain the relative path from the settings.cfg file to the 'top' of the project source tree

PROGRAM (Textual) - Should contain the name for the program binary

PRJ_SEARCH_INCLUDES (Textual) - Should contain the lists relative paths, from set-

tings.cfg location, to folders where you want srcdeps to look for binary dependencies, usually the folders where you have your sources. Note that srcdeps *will not* perform a recursively descent search on these folders.

PRJ_SEARCH_RECURSE_INCLUDES (Textual) - Should contain the list relative paths, from settings.cfg location, to folders where you want srcdeps to look for binary dependencies, usually the folders where you have your sources. Note that srcdeps *will* perform a recursively descent search on these folders.

SOURCES (Textual) - Should contain the source file that contains the application entry point. While building library binaries or not following for some reason the rule 'for each header file there exists a source file with the same name' then you should add the source relative paths, from current settings.cfg location.

CLAM configuration variables include:

CLAM_DOUBLE(Boolean) - This variable controls whether CLAM::TData datatype is either a single precision floating-point number (ANSI C++ float type) or a double precision floating-point number (ANSI C++ double type).

CLAM_USE_XML(Boolean) - This variable controls whether CLAM code is built with XML-based Object External Storage support. Disabling it could improve compiling speed as well as reduce code size.

CLAM_DISABLE_CHECKS(Boolean) - This variable controls whether CLAM internal pre-condition, postcondition and invariant verification checks are performed or not. Deactivating it could improve code speed in spite of robustness.

CLAM_USE_RELEASE_ASSERTS(Boolean) - This variable controls whether CLAM Asserts behave equally in "debug" and "not debug" mode.

And finally external libraries variables are:

USE_ALSA(Boolean) - Tells the build system to make applications to link against ALSA. Note that this variable can only have effect on GNU/Linux systems.

USE_FFTW(Boolean) - Tells the build system to make applications to link against the FFTW library.

USE_FLTK(Boolean) - Tells the build system to make applications to link against FLTK.

USE_DIRECTX(Boolean) - Tells the build system to make applications to link against DirectX SDK. Note that this variable only has effect on Microsoft Windows(c) systems.

USE_PORTMIDI(Boolean) - Tells the build system to make applications to link against Portmidi. Note that this variable only has effect on Microsoft Windows(c) systems.

USE_RTAUDIO(Boolean) - Tells the build system to make applications to link against RtAudio. Note that this variable only has effect on Microsoft Windows(c) systems.

USE_PTHREADS(Boolean) - Tells the build system to make applications to link against pthreads (POSIX threads library). Note that this variable only has effect on Microsoft Windows(c) systems.

USE_QT(Boolean) - Tells the build system to make applications to link against Qt Toolkit.

USE_CPPUNIT(Boolean) - Tells the build system to make applications to link against cppUnit library.

§A.2.4 External libraries

We have already mentioned that one of the software engineering maximas that CLAM observes is *reuse*. When developing a framework everything is designed with future reuse in mind. In a similar sense a framework must reuse all possible pre-existing solutions. In CLAM we always try not to fall onto the "redesigning the wheel" paradigm. Because of this a number of external libraries that provide specific functionality are used. In this section we will briefly describe their main features.

FFTW [Frigo and Johnson, 1998, www-FFTW,] is a C subroutine library for computing the Discrete Fourier Transform (DFT) in one or more dimensions, of both real and complex data, and of arbitrary input size. We believe that FFTW, which is free software, should become the FFT library of choice for most applications. Our benchmarks, performed on on a variety of platforms, show that FFTW's performance is typically superior to that of other publicly available FFT software. Moreover, FFTW's performance is portable: the program will perform well on most architectures without modification. Another FFT library that has been integrated into CLAM is the new **FFT-Ooura**, which is almost as efficient as the FFTW but has a BSD-style kind of license.

Two different graphical user interface toolkits have been in some way integrated into the CLAM framework: FLTK and QT. The Fast and Light ToolKit (**FLTK**) [www-FLTK,] (pronounced "fulltick") is a LGPL'd C++ graphical user interface toolkit for X (UNIX), MacOS, and Microsoft Windows, that offers lightweight solutions for building GUI's and supports 3D graphics with OpenGL. It is currently maintained by a small group of developers across the world with a central repository on SourceForge. **Qt** [Blanchette and Summerfield, 2004, www-QT,] is a C++ framework for application development. It lets application developers target all major operating systems with a single application source code. Qt provides a platform-independent API to all central platform functionality: GUI, database access, networking, file handling, etc. The Qt library encapsulates the different APIs of different operating

systems, providing the application programmer with a single, common API for all operating systems. The native C APIs are encapsulated in a set of well-designed, fully object-oriented C++ classes. It is clear that Qt offers superior features to those found in FLTK. But it has a downside: the license can only be considered Free for the GNU/Linux platform.

Currently most of CLAM's XML support is given through the use of Xerces-C++. **Xerces-C++** [Xercesc,] is a validating XML parser written in a portable subset of C++. Xerces-C++ makes it easy to give your application the ability to read and write XML data. A shared library is provided for parsing, generating, manipulating, and validating XML documents. Xerces-C++ is faithful to the XML 1.0 recommendation and associated standards (DOM 1.0, DOM 2.0. SAX 1.0, SAX 2.0, Namespaces, and W3C's XML Schema recommendation version 1.0.) The parser provides high performance, modularity, and scalability. Source code, samples and API documentation are provided with the parser. For portability, care has been taken to make minimal use of templates, no RTTI, no C++ namespaces and minimal use of #ifdefs. It must be said though that some of those decisions have produced a code that is less efficient than desirable. For this reason other solutions for the CLAM XML support are already under study.

In order to offer transparent audio input/audio on any platform different solutions have been used in CLAM. As already explained in section 3.2.2.4 CLAM adds an abstraction layer on a number of libraries in order to make them homogeneously accessible. These libraries are PortAudio, RTAudio, Alsa, and DirectX Sound. The first two are indeed cross-platform audio libraries that in CLAM are used both for Microsoft Windows and Apple OSX while the other two are platform-specific.

Port Audio [Bencina and Burk, 2001, Bencina, 2003, www-PortAudio,] is a Free cross-platform library for managing audio input and output on virtually any platform. It offers thorough support for all the platforms offering low latency and high efficiency. **RtAudio**[Scavone, 2002, www-RtAudio,] is a similar library. Although it may not offer as many features as PortAudio, it is much easier to use and deploy as it is just made up of a single C++ class and it is more object-oriented. On GNU/Linux CLAM directly addresses the **ALSA** library interface in order to ensure low-latency and the highest efficiency. This can also be done (up to the operating system's particular limitations) on the Microsoft Windows platform using the **DirectX Sound** interface to CLAM.

In a similar way **PortMIDI** [www-PortMIDI,] is used for MIDI input/output on the Microsoft Windows and Apple OSX operating systems while the ALSA interface is directly used on GNU/Linux.

Three libraries are used for sound file input/output: Libsndfile, LibMad and OggVorbis. **Libsndfile** is a GPL library that gives support to virtually any sound file format. In CLAM it is used especially for encoding/decoding PCM audio formats. Both LibMad and OggVorbis offer support for

compressed formats not available in Libsndfile. OggVorbis supports the Free **Ogg** format while LibMad supports the **MP3** format, which is subject to some patent issues but so extended that it seemed a good idea to integrate into CLAM.

Id3lib[www-id3lib,] is a Free Software, cross-platform software development library for reading, writing, and manipulating ID3v1 and ID3v2 tags. ID3 is a standard specification to add metadata information (such as author or title) to any sound file.

Finally the **Win Pthreads** library is used to ensure the use of this standard in the Microsoft Windows operating system when handling multi-threading issues.

§A.2.5 CLAM User Group

Although CLAM is already available, at the time of this writing its usage is still mainly internal to the MTG. It is still too early to evaluate its public acceptance. It is important to note that the framework will not be publicly advertised in a general sense until it reaches its 1.0 release.

Internally CLAM has been used in many different projects all of them already reviewed in section 3.2.3.

As for its external usage it may be mentioned that the CLAM mailing-list has now well above 100 subscribed members, of which about 25 are external to the MTG (note that we are considering students from the university also as internal users although they formally are not). The truth is that these members are not very active and, although we know they are using CLAM we do not have any news of what have been the final upcoming of their initial efforts.

APPENDIX B

Spectral Processing

Although this subsection may seem a bit off-topic, there are several reasons for its inclusion. First, many of the applications that are mentioned in different chapters are related to spectral processing techniques; second, the CLAM framework was born when the research group was basically involved in research into spectral domain and that definitely biased and conditioned many of the design decisions; and last, much work of the author (see C) is directly related with spectral modeling and is not reflected anywhere else in this Thesis.

The most common approach for converting a time domain signal into its frequency domain representation is the Short-Time Fourier Transform (STFT). It is a general technique from which we can implement loss-less analysis/synthesis systems. Many sound transformation systems are based on direct implementations of the basic algorithm and several examples have been presented in chapter 8.

In this chapter, we will briefly mention the Sinusoidal Model and will concentrate, with a Matlab sample code, in the Sinusoidal plus Residual Model. Anyhow, the decision as to what spectral representation to use in a particular situation is not an easy one. The boundaries are not clear and there are always compromises to take into account, such as: (1) sound fidelity, (2) flexibility, (3) coding efficiency, and (4) computational requirements. Ideally, we want to maximize fidelity and flexibility while minimizing memory consumption and computational requirements. The best choice for maximum fidelity and minimum compute time is the STFT that, anyhow, yields a rather inflexible representation and inefficient coding scheme. Thus our interest in finding higher-level representations as the ones we present in this section.

Using the output of the STFT, the Sinusoidal model represents a step towards a more flexible representations while compromising both sound fidelity and computing time. It is based on modeling the time-varying spectral characteristics of a sound as sums of time-varying sinusoids. To obtain a sinusoidal representation from a sound, an analysis is performed in order to estimate the instantaneous

amplitudes and phases of the sinusoids. This estimation is generally done by first computing the STFT of the sound, then detecting the spectral peaks (and measuring the magnitude, frequency and phase of each one), and finally organizing them as time-varying sinusoidal tracks.

It is a quite general technique that can be used in a wide range of sounds and offers a gain in flexibility compared with the direct STFT implementation.

§B.0.5.1 Sinusoidal plus Residual Model

The Sinusoidal plus Residual model can cover a wide "compromise space" and can in fact be seen as the generalization of both the STFT and the Sinusoidal models. Using this approach, we can decide what part of the spectral information is modeled as sinusoids and what is left as STFT. With a good analysis, the Sinusoidal plus Residual representation is very flexible while maintaining a good sound fidelity, and the representation is quite efficient. In this approach, the Sinusoidal representation is used to model only the stable partials of a sound. The residual, or its approximation, models what is left, which should ideally be a stochastic component. This model is less general than either the STFT or the Sinusoidal representations but it results in an enormous gain in flexibility [Serra, 1989, Serra, 1996, Serra, 1990].

The sinusoidal plus residual model assumes that the sinusoids are stable partials of the sound with a slowly changing amplitude and frequency. With this restriction, we are able to add major constraints to the detection of sinusoids in the spectrum and omit the detection of the phase of each peak.

Within this model we can either leave the residual signal, , to be the difference between the original sound and the sinusoidal component, resulting into an identity system, or we can assume that is a stochastic signal. In this case, the residual can be described as filtered white noise. That is, the residual is modeled by the time-domain convolution of white noise with a time-varying frequency-shaping filter.

The implementation of the analysis for the Sinusoidal plus Residual Model is more complex than the one for the Sinusoidal Model. Figure B.1 shows a simplified block- diagram of this analysis.

The first few steps are the same than in a sinusoidal-only analysis. The major differences start on the peak continuation process since in order to have a good partial-residual decomposition we have to refine the peak-continuation process in such a way as to be able to identify the stable partials of the sound. Several strategies can be used to accomplish this. The simplest case is when the sound is monophonic and pseudo-harmonic. By using the fundamental frequency information in the peak continuation algorithm, we can identify the harmonic partials.

The residual component is obtained by first generating the sinusoidal component with additive

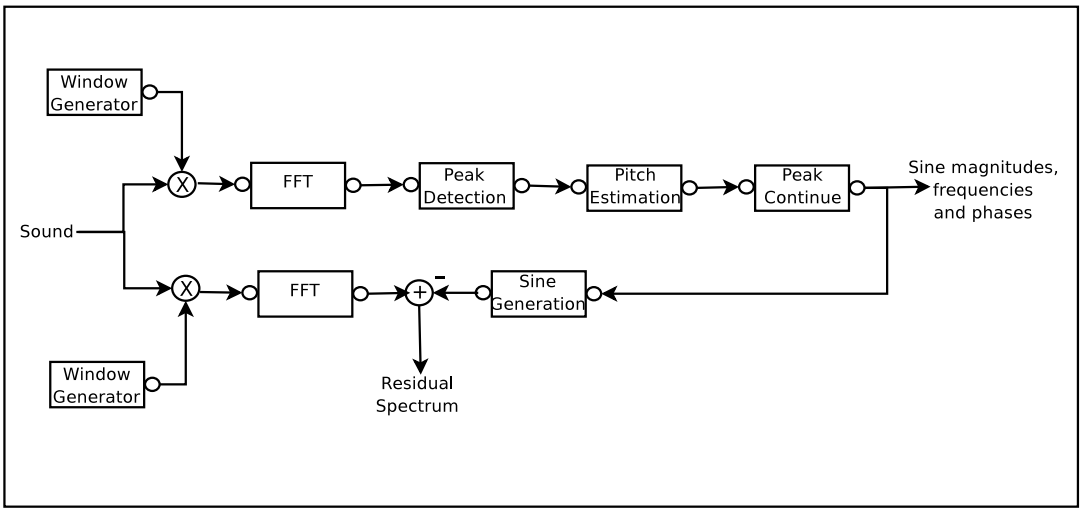

Figure B.1: SMS analysis algorithm

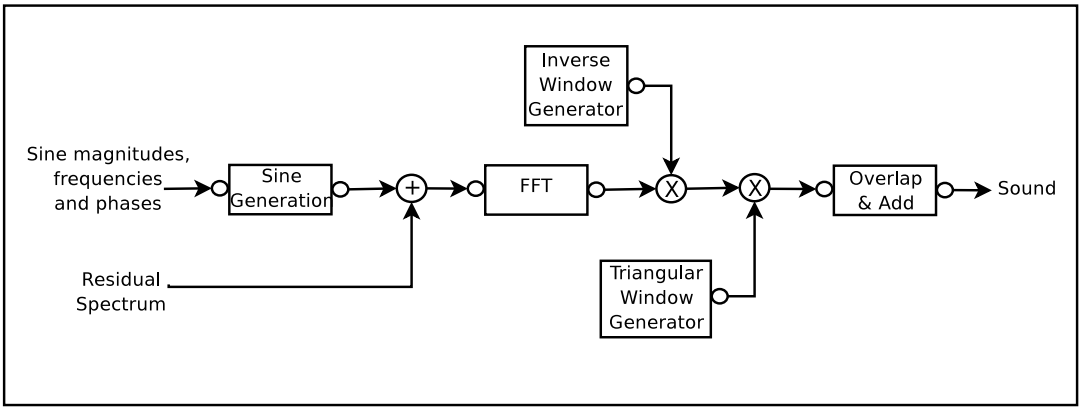

Figure B.2: SMS Synthesis Algorithm

synthesis, and then subtracting it from the original waveform. This is possible because the instantaneous phases of the original sound are matched and therefore the shape of the time domain waveform preserved. A spectral analysis of this time domain residual is done by first windowing it, window which is independent of the one used to find sinusoids, and thus we are free to choose a different time-frequency compromise. An amplitude correction step can improve the time smearing produced in the sinusoidal subtraction. Then the FFT is computed and the resulting spectrum can be modeled using several existing techniques. The spectral phases might be discarded if the residual can be approximated as a stochastic signal.

Once the different components in the SMS have been obtained, different interesting transformations can be applied in the spectral domain [Amatriain et al., 2002b]. After processing the spectral components, these must be synthesized back to produce the output sound. The diagram in figure B.2 illustrates the SMS synthesis algorithm.

The original sinusoidal plus residual model has led to other different spectral models that still share some of its basis. [Ding and Qian, 1997; Fitz, Haken and Christensen, 2000; Verma, 2000]

APPENDIX C

Publications by the Author

In this annex a compilation of the most relevant publications in which the author has participated is given in order to have a better overview of the author's research work. For each publication, we will include the abstract (or the introduction where not available) and the chapter(s) in this Thesis to which it is most relevant. The publications are sorted by date, in decreasing order.

---

author : Amatriain, X. and Bonada, J. and Loscos, A. and Arcos, J. and Verfaille, V.

title : Content-based Transformations

year : 2003

journal : Journal of New Music Research

volume : 32

number : 1

related to chapter : 5

abstract :

Content processing is a vast and growing field that integrates different approaches borrowed from the signal processing, information retrieval and machine learning disciplines. In this article we deal with a particular type of content processing: the so-called content-based transformations. We will not focus on any particular application but rather try to give an overview of different techniques and conceptual implications. We first describe the transformation process itself, including the main model schemes that are commonly used, which lead to the establishment of the formal basis for a definition of content-based transformations. Then we take a quick look at a general spectral based analysis/synthesis approach to process audio signals and how to extract features that can be used in the content-based transformation context. Using this analysis/synthesis approach we give some examples on how content-based transformations can be applied to modify the basic perceptual axis of a sound and how we can

even combine different basic effects in order to perform more meaningful transformations. We finish by going a step further in the abstraction ladder and present transformations that are related to musical (and thus symbolic) properties rather than to those of the sound or the signal itself.

---

author : Gómez, E. and Gouyon, F. and Herrera, P. and Amatriain, X.

title : Using and enhancing the current MPEG-7 standard for a music content processing tool

year : 2003

book title : Proceedings of Audio Engineering Society, 114th Convention

related to chapter : 5

abstract :

The aim of this document is to discuss possible ways of describing some music constructs in a dual context. First, that of the current standard for multimedia content description: MPEG-7. Second, that of a specific software application, the Sound Palette (a tool for content-based management, content edition and transformation of simple audio phrases). We discuss some MPEG-7 limitations regarding different musical layers: melodic (present but underdeveloped), rhythmic (practically absent) and instrumental (present though using an exclusive procedure). Some proposals for overcoming them are presented in the context of our application.

---

author : Gómez, E. and Gouyon, F. and Herrera, P. and Amatriain, X.

title : MPEG-7 for Content-based Music Processing

year : 2003

book title : Proceedings of 4th WIAMIS-Special session on Audio Segmentation and Digital Music

related to chapter : 5

abstract :

The aim of this document is to present how the MPEG-7 standard has been used in a tool for content-based management, edition and transformation of audio signals: the Sound Palette. We discuss some MPEG-7 limitations regarding different musical layers, and some proposals for overcoming them are presented.

---

author : Gómez, E. and Grachten, M. and Amatriain, X. and Arcos, J.

title : Melodic characterization of monophonic recordings for expressive tempo transformations

year : 2003

book title : Proceedings of Stockholm Music Acoustics Conference 2003

related to chapter : 5

abstract:

The work described in this paper aims at characterizing tempo changes in terms of expressivity, in order to develop a transformation system to perform expressive tempo transformations in monophonic instrument phrases.

For this purpose, we have developed an analysis tool that extracts a set of acoustic features from monophonic recordings. This set of features is structured and stored following a description scheme that is derived from the current MPEG-7 standard. These performance descriptions are then compared with their corresponding scores, using edit distance techniques, for automatically annotating the expressive transformations performed by the musician. Then, these annotated performance descriptions are incorporated in a case-based reasoning (CBR) system in order to build an expressive tempo transformations case base. The transformation system will use this CBR system to perform tempo transformations in an expressive manner.

Saxophone performances of jazz standards played by a professional performer have been recorded for this characterization.

In this paper, we first describe which are the acoustic features that have been used for this characterization and how they are structured and stored. Then, we explain the analysis methods that have been implemented to extract this set of features from audio signals and how they are processed by the CBR system. Results are finally presented and discussed.

---

author : Gómez, E. and Peterschmitt, G. and Amatriain, X. and Herrera, P.

title : Content-based melodic transformations of audio for a music processing application

year : 2003

book title : Proceedings of 6th International Conference on Digital Audio Effects

related to chapter : 5

abstract :

The goal of this paper is to present a system that performs melodic transformations to monophonic audio phrases. First, it extracts a melodic description from the audio. This description is presented to the user and can be stored and loaded in a structured format. The system proposes a set of high-level melodic transformations for the audio signal. These transformations are mapped into a set of low-level transformations of the melodic description that are then applied to the audio signal. The algorithms for description extraction and audio transformation are presented.

---

author : Geiger, G. and Mora, A. and Rubio, X. and Amatriain, X.

title : AGNULA: A GNU Linux Audio Distribution

year 2003

book title : Proceedings of II Jornades de Software Lliure

related to chapter : 3

abstract (in original Catalan language):

En aquest document es presenta el projecte AGNULA , enmarcat dins la tasca del foment de programari lliure a nivell europeu. S'expliquen els seus objectius, promotors i les diferents distribucions que en formen part. Finalment, es fa un resum de les principals aplicacions incloses.

---

author : Arumi, P. and Garcia, D. and Amatriain, X.

title : CLAM, Una llibreria lliure per Audio i Música

year : 2003

book title : Proceedings of II Jornades de Software Lliure

related to chapter : 3

abstract (in original Catalan language):

CLAM és un framework lliure i orientat a objectes en C++ que ofereix als desenvolupadors solucions de disseny i un conjunt de components reusables per construir aplicacions musicals i d'audio i per la recerca en l'àmbit del processat del senyal. Algunes d'aquestes eines, també lliures, ja s'han desenvolupat per part de l'MTG. La metodologia de desenvolupament de CLAM assegura la seva qualitat. Degut, sobretot, a la incorporació de CLAM a diverses distribucions de GNU/Linux està facilitant l'aparició d'eines multimèdia de tecnologia avançada en entorns lliures.

---

author : Amatriain, X. and Bonada, J. and Loscos, A. and Serra, X.

title : Spectral Processing

year : 2002

book title : DAFX: Digital Audio Effects

editor : Udo Zölzer

publisher : John Wiley and Sons, Ltd.

related to chapter : 3

introduction :

In the context of this book, we are looking for representations of sound signals and signal processing systems that can give us ways to design sound transformations in a variety of music applications

and contexts. It should have been clear throughout the book, that several points of view have to be considered, including a mathematical, thus objective perspective, and a cognitive, thus mainly subjective, standpoint. Both points of view are necessary to fully understand the concept of sound effects and to be able to use the described techniques in practical situations.

The mathematical and signal processing points of view are straightforward to present, which does not mean easy, since the language of the equations and of flow diagrams is suitable for them. However, the top-down implications are much harder to express due to the huge number of variables involved and to the inherent perceptual subjectivity of the music making process. This is clearly one of the main challenges of the book and the main reason for its existence.

The use of a spectral representation of a sound yields a perspective that is sometimes closer to the one used in a sound engineering approach. By understanding the basic concepts of frequency domain analysis, we are able to acquire the tools to use a large number of effects processors and to understand many types of sound transformations systems. Moreover, being the frequency domain analysis a somewhat similar process than the one performed by the human hearing system, it yields fairly intuitive intermediate representations.

The basic idea of spectral processing is that we can analyze a sound to obtain alternative frequency domain representations, which can then be transformed and inverted to produce new sounds. Most of the approaches start by developing an analysis/synthesis system from which the input sound is reconstructed without any perceptual loss of sound quality. The techniques described in the previous chapter are clear examples of this approach. Then the main issue is what is the intermediate representation and what parameters are available for applying the desired transformations.

Perceptual or musical concepts such as timbre or pitch are clearly related to the spectral characteristics of a sound. Even some common processes for sound effects are better explained using a frequency domain representation. We usually think on the frequency axis when we talk about equalizing, filtering, pitch shifting, harmonizing... In fact, some of them are specific to this signal processing approach and do not have an immediate counterpart on the time domain. On the other hand, most (but not all) of the sound effects presented in this book can be implemented in the frequency domain.

Another issue is whether or not this approach is the most efficient, or practical, for a given application. The process of transforming a time domain signal into a frequency domain representation is, by itself, not an immediate step. Some parameters are difficult to adjust and force us to take several compromises. Some settings, such as the size of the analysis window, have little or nothing to do with the high-level approach we intend to favor, and require the user to have a basic signal processing understanding.

In that sense, when we talk about higher level spectral processing we are thinking of an intermediate analysis step in which relevant features are extracted or computed from the spectrum. These relevant features should be much closer to a musical or high-level approach. We can then process the features themselves or even apply transformations that keep some of the features unchanged. For example, we can extract the fundamental frequency and the spectral shape from a sound and then modify the fundamental frequency without affecting the shape of the spectrum.

Assuming the fact that there is no single representation and processing system optimal for everything, our approach will be to present a set of complementary spectral models that can be combined to be used for the largest possible set of sounds and musical applications.

In the next section we introduce two spectral models: Sinusoidal and Sinusoidal plus Residual. These models already represent a step up on the abstraction ladder and from either of them, we can identify and extract higher-level information of a sound, such as: harmonics, pitch, spectral shape, vibrato, or note boundaries, that is Higher Level Features. This analysis step brings the representation closer to our perceptual understanding of a sound. The complexity of the analysis will depend on the type of feature that we want to identify and the sound to analyze. The benefits of going to this higher level of analysis are enormous and open up a wide range of new musical applications.

Having set the basis of the Sinusoidal plus Residual model, we will then give some details of the techniques used both in its analysis and synthesis process, providing Matlab code to implement an analysis-synthesis framework. This Matlab implementation is based on the Spectral Modeling Synthesis framework. SMS [http://www.iua.upf.es/~sms] is a set of spectral based techniques and related implementations for the analysis/transformation/synthesis of an audio signal based on the scheme presented in .

We will provide a set of basic audio effects and transformations based on the implemented Sinusoidal plus Residual analysis/synthesis. Matlab code is provided for all of them.

We will finish with an explanation of content dependant processing implementations. We introduce a real-time singing voice conversion application that has been developed for use in Karaoke applications, and we define the basis of a nearly loss less Time Scaling algorithm. The complexity and extension of these implementations prevent us from providing the associated Matlab code, so we leave that task as a challenge for advanced readers.

---

author : Amatriain, X. and Herrera, P.

title : Transmitting Audio Content as Sound Objects

year : 2002

book title : Proceedings of AES22 International Conference on Virtual, Synthetic and Entertainment Audio

related to chapter : 5

abstract :

As audio and music applications tend to a higher level of abstraction and to fill in the gap between the signal processing world and the end-user we are more and more interested on processing content and not (only) signal. This change in point of view leads to the redefinition of several "classical" concepts, and a new conceptual framework needs to be set to give support to these new trends. In [Amatriain and Herrera 2001], a model for the transmission of audio content was introduced. The model is now extended to include the idea of Sound Objects. With these thoughts in mind, examples of design decisions that have led to the implementation of the CLAM framework are also given.

---

author : Amatriain, X. and de Boer, M. and Robledo, E. and Garcia, D.

title : CLAM: An OO Framework for Developing Audio and Music Applications

year : 2002

book title : Proceedings of 17th Annual ACM Conference on Object-Oriented Programming, Systems, Languages and Applications

related to chapter : 3

abstract :

CLAM (C++ Library for Audio and Music) is a framework for audio and music programming. It may be used for developing any type of audio or music application as well as for doing more complex research related with the field. In this paper we introduce the practicalities of CLAM&acutes first release as well as some of the sample application that have been developed within the framework. See [1] for a more conceptual approach to the description of the CLAM framework.

---

author : Amatriain, X. and Arumi, P. and Ramírez, M.

title : CLAM, Yet Another Library for Audio and Music Processing?

year : 2002

book title : Proceedings of 17th Annual ACM Conference on Object-Oriented Programming, Systems, Languages and Applications

related to chapter : 3

abstract :

CLAM (C++ Library for Audio and Music) is a framework that aims to offer extensible,

generic and efficient design and implementation solutions for developing Audio and Music applications as well as for doing more complex research related with the field. Although similar libraries exist, some particularities make CLAM of high interest for anyone interested in the field.

---

author : Garcia, D. and Amatriain, X.

title : XML as a means of control for audio processing, synthesis and analysis

year : 2001

book title : Proceedings of MOSART Workshop on Current Research Directions in Computer Music

related to chapter : 3 and 5

abstract :

This paper discusses about benefits derived from providing XML support to the component based framework for audio systems that we are developing. XML is used as data format for persistence, visualization and inter-application interface. Direct XML support is a very useful feature for an audio framework because of the popularity of the XML format as data interchange format, and the introduction of MPEG7 standard, an XML based description format for multimedia content. Formatting task has been distributed along the system objects in a compositional way, making easy to format a single object from its parts. The system minimizes the overhead added to a class and the programmer effort to support XML I/O. A default XML implementation has been provided for most of the future data structures, giving the chance to customize it. The system has been designed to be reused with other formats with a minimal impact on the system.

---

author : Amatriain, X. and Bonada, J. and Loscos, A. and Serra, X.

title : Spectral Modeling for Higher-level Sound Transformation

year : 2001

book title : Proceedings of MOSART Workshop on Current Research Directions in Computer Music

related to chapter : 3 and 5

abstract :

When designing audio effects for music processing, we are always aiming at providing higher-level representations that may somehow fill in the gap between the signal processing world and the end-user. Spectral models in general, and the Sinusoidal plus Residual model in particular, can sometimes offer ways to implement such schemes.

author : Amatriain, X. and Herrera, P.

title : Audio Content Transmission

year : 2001

book title : Proceedings of COST G6 Conference on Digital Audio Effects 2001

related to chapter : 5

abstract :

Content description has become a topic of interest for many researchers in the audiovisual field. While manual annotation has been used for many years in different applications, the focus now is on finding automatic content-extraction and content-navigation tools. An increasing number of projects, in some of which we are actively involved, focus on the extraction of meaningful features from an audio signal. Meanwhile, standards like MPEG7 are trying to find a convenient way of describing audiovisual content. Nevertheless, content description is usually thought of as an additional information stream attached to the actual content and the only envisioned scenario is that of a search and retrieval framework. However, in this article it will be argued that if there is a suitable content description, the actual content itself may no longer be needed and we can concentrate on transmitting only its description. Thus, the receiver should be able to interpret the information that, in the form of metadata, is available at its inputs, and synthesize new content relying only on this description. It is possibly in the music field where this last step has been further developed, and that fact allows us to think of such a transmission scheme being available on the near future.

---

author : Herrera, P. and Amatriain, X. and Batlle, E. and Serra, X.

title : Towards Instrument Segmentation for Music Content Description: a Critical Review of Instrument Classification Techniques

year : 2000

book title : Proceedings of International Symposium on Music Information Retrieval

related to chapter : 5

abstract :

A system capable of describing the musical content of any kind of sound file or sound stream, as it is supposed to be done in MPEG7-compliant applications, should provide an account of the different moments where a certain instrument can be listened to. In this paper we concentrate on reviewing the different techniques that have been so far proposed for automatic classification of musical instruments. As most of the techniques to be discussed are usable only in "solo" performances we will evaluate their

applicability to the more complex case of describing sound mixes. We conclude this survey discussing the necessity of developing new strategies for classifying sound mixes without a priori separation of sound sources.

---

author : Amatriain, X. and Bonada, J. and Serra, X.

title : METRIX: A Musical Data Definition Language and Data Structure for a Spectral Modeling Based Synthesizer

year : 1998

book title : Proceedings of COST G6 Conference on Digital Audio Effects 1998

related to chapter : 6

abstract :

Since the MIDI 1.0 specification, well over 15 years ago, many have been the attempts to give a solution to all the limitations that soon became clear. None of these have had a happy ending, mainly due to commercial interests and as a result, when trying to find an appropriate synthesis control user interface, we had not many choices but the use of MIDI. That\x{2019}s the reason why the idea of defining a new user interface aroused. In this article, the main components of this interface will be discussed, paying special attention to the advantages and new features it reports to the end-user.

Appendix D

Free Software Tools

This Thesis has been developed using exclusively Free Software tools. In particular, we have used:

- Text editing: LaTeX and LyX for the final document and Emacs for auxiliary tasks and bibtex editing
- Vector Graphics: Dia
- UML Graphics: Umbrello and Dia
- Other Graphics: the GIMP
- Version control: CVS and LinCVS
- Operating System: GNU/Linux