Models, Analysis and Execution of Audio Graphs in Interactive Multimedia Systems

Pierre Donat-Bouillud

SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS DEPUIS 1257

THÈSE DE DOCTORAT DE SORBONNE UNIVERSITÉ

Spécialité

Informatique

École doctorale Informatique, Télécommunications et Électronique (Paris)

Présentée par

Pierre DONAT-BOUILLUD

Pour obtenir le grade de

DOCTEUR de SORBONNE UNIVERSITÉ

Sujet de la thèse :

Models, Analysis and Execution of Audio Graphs in Interactive Multimedia Systems

soutenue le 6 décembre 2019

devant le jury composé de :

M. Florent Jacquemard	Directeur de thèse
M. Jean-Louis Giavitto	Encadrant de thèse
Mme Myriam Desainte-Catherine	Rapporteur
M. Pierre Jouvelot	Rapporteur
M. Alain Girault	Examinateur
M. Christoph Kirsch	Examinateur
M. Yann Orlarey	Examinateur
M. Dumitru Potop-Butucaru	Examinateur

Abstract

Interactive Multimedia Systems (IMSs) are used in concert for interactive performances, which combine in real time acoustic instruments, electronic instruments, data from various sensors (gestures, midi interface, etc.) and the control of different media (video, light, etc.). This thesis presents a formal model of audio graphs, via a type system and a denotational semantics, with multirate timestamped bufferized data streams that make it possible to represent with more or less precision the interleaving of the control (for example a low frequency oscillator, velocities from an accelerometer) and audio processing in an MIS. An audio extension of Antescofo, an IMS that acts as a score follower and includes a dedicated synchronous timed language, has motivated the development of this model. This extension makes it possible to connect Faust effects and native effects on the fly safely. The approach has been validated on a mixed music piece and an example of audio and video interactions.

At last, this thesis proposes offline optimizations based on the automatic resampling of parts of an audio graph to be executed. A quality and execution time model in the graph has been defined. Its experimental study was carried out using a prototype IMS based on the automatic generation of audio graphs, which has also made it possible to characterize resampling strategies proposed for the online case in real time.

Résumé

Les Systèmes Interactifs Multimédia (SIM) sont utilisés en concert pour des spectacles interactifs, qui mêlent en temps-réel instruments acoustiques, instruments électroniques, des données issues de divers capteurs (gestes, interface midi, etc) et le contrôle de différents média (vidéo, lumière, etc). Cette thèse présente un modèle formel de graphe audio, via un système de types et une sémantique dénotationnelle, avec des flux de données bufferisés datés multi-périodiques qui permettent de représenter avec plus ou moins de précisions l'entrelacement du contrôle (par exemple un oscillateur basse fréquence, des vitesses issues d'un accéléromètre) et des traitements audio dans un SIM. Une extension audio d'Antescofo, un SIM qui fait office de suiveur de partition et qui comporte un langage synchrone temporisé dédié, a motivé le développement de ce modèle. Cette extension permet de connecter des effets Faust et des effets natifs, à la volée, de façon sure. L'approche a été validée sur une pièce de musique mixte et un exemple d'interactions audio et vidéo.

Enfin, cette thèse propose des optimisations hors-ligne à partir du rééchantillonnage automatique de parties d'un graphe audio à exécuter. Un modèle de qualité et de temps d'exécution dans le graphe a été défini. Son étude expérimentale a été réalisée grâce à un SIM prototype à partir de la génération automatique de graphes audio, ce qui a permis aussi de caractériser des stratégies de rééchantillonnage proposées pour le cas en ligne en temps-réel.

Acknowledgements

Je remercie mes encadrants de thèse, Florent et Jean-Louis, qui m'ont appris à faire de la recherche, avec deux visions différentes et complémentaires, la vision Inria, et la vison Ircam. Merci Jean-Louis, pour toutes tes anecdotes de directeur adjoint sur le mille-feuille administratif de la recherche en France. Merci à Virginie, à Kevin, à Meriem, les assistants d'équipe à Inria, et à Anne-Marie, Sylvie, et Vasiliki, à l'Ircam. Merci à l'équipe d'Antescofo des débuts qui m'ont donné envie de poursuivre à l'Ircam, Arshia, José, et Philippe. Merci à Clément, qui m'a montré l'exemple du thésard impliqué. Merci aux autres doctorants de l'Ircam, dont j'ai eu le plaisir d'être le très actif délégué ! Merci à Léo, et à Pierre. Merci aussi à Pierre pour avoir relu les premiers chapitres de cette thèse. Merci à Martin qui va prendre la relève dans le bureau A112. Merci à José-Miguel, grâce à qui j'ai pu aller donner quelques master class en Russie et découvrir à la fois la truculence perpignanaise et la rigueur suisse, y compris en Chine et en Russie. Merci aux membres de l'Ensemble Flashback, Alex, Thomas et Thomas, Philippe, Emmanuel.

I would like to thank my reviewers, Myriam Desainte-Catherine and Pierre Jouvelot, for the insightful comments and corrections on my dissertation.

Merci à Yann Orlarey, Dominique Fober, Stéphane Letz, et Romain Michon, qu m'ont accueilli au Grame une semaine pour y découvrir les entrailles du compilateur Faust, et m'ont fait redécouvrir ma ville de naissance.

I would also like to thanks Christoph Kirsch, whose lab I visited in Salzburg in Austria, where he told me how and about the importance of finding interesting problems, and where I first got the idea of the optimization by resampling. Many thanks to Ana, Martin, Ouafae, and Sebastian, who tried to remind me how to ski!

Merci à mes parents et à mes frères, qui m'ont toujours soutenu. Merci à mes amis qui étaient toujours là, Yann, Sylvain, Yasya, Goce, Victor, Alina, Gabriel, Élie, y compris pendant la rédaction anachorétique du manuscrit. Merci à mon merveilleux colocataire, Michel/Matthieu hybride, prescripteur en humour, à la faconde tantôt croquignolesque, tantôt érudite, sans qui cette thèse aurait été moins joyeuse.

Спасибо русским и украинским друзьям. Благодаря вам, я узнал красивыйе язык и културу. 谢谢曼，她带我认识了中国，并与我度过了愉快的时光

Contents

1. Introduction

Interactive Multimedia Systems (IMS) [Row93] offer powerful signal processing functionalities for audio and video and ways of controlling audio and video by reaction to events coming from the analysis of the signal or from the physical environment. They are used for concerts of mixed music, as in Anthèmes II by Pierre Boulez, where human musicians with acoustical instruments play along with electronic sounds driven by computers. They can also be used in theatre plays where they can control lights in addition to the music, or in live coding [Col+03], where the performer-coder programs a musical piece in real time in front of the audience. IMSs combine multiple inputs ranging from the move of a slider, periodic low-frequency events sampled on a curve, to the detection of a note in an audio stream or the detection of gestures [Bev+10; Fer+17] through accelerometers and gyroscopes. They operate as a central hub that gathers inputs from various sensors, analyses and processes them, and dispatches the results to control some synthesis and transformation of the multimedia materials, as illustrated in Figure 1.1. The sensors and the processing can be directly integrated into the IMS or communicate via messages, thanks to various protocols such as MIDI [Loy85], OSC [Wri05] or O2 [DC16], to mention only the most popular ones. IMSs interact also with other IMSs and with human performers, establishing a feedback loop, often called *human-in-the-loop* [Con+12]. Sound can also be distributed and transmitted to the IMS hub, using Jack Connection Kit [LOF05] for instance. Upon receiving these events and signals, IMSs react by processing the sounds, triggering processes, and changing parameters. The *interactivity* in IMSs emphasizes the need to state precisely in time how to control the diverse processes running in the IMS.

IMSs are difficult to design and implement because they must orchestrate heterogeneous models of computation, and have to accommodate periodic signals, such as audio signals, typically at 44 100 Hz, video signals, which are carried at a smaller frequency, for instance 24 Hz (commonly said as 24 FPS, or frame per seconds), or gesture data from an accelerometer, and control events, which are aperiodic.

More generally, IMSs have to take into account many computations at very

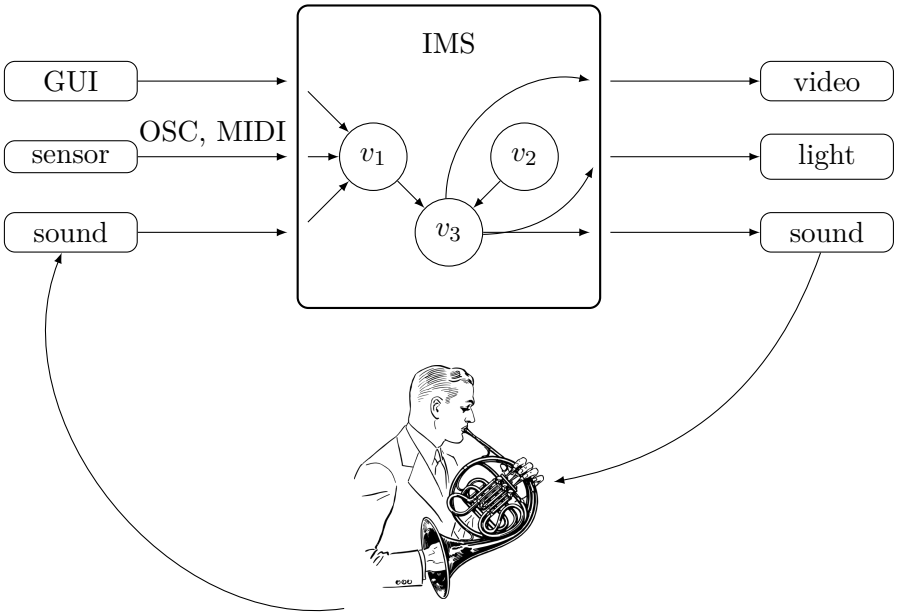


Figure 1.1.: The big picture of an IMS as a central hub connecting sensors and controlling synthesis and signal processors, with a human in the loop.

different rates. Moreover, they are *real-time systems*: not outputting audio at the right time can entail a discontinuity in the signal, hence an unpleasant click. Dealing with a control event slightly late can also be noticed: two events are perceived by our ears to be simultaneous if they are less than 20 ms apart. Video constraints are less stringent: studies [Ste96] have shown that the human ear is more sensitive to audio gaps and glitches than the human eye is to video jitter. Dropping one frame[1] in a video is usually not visible. Therefore, in this work, we will focus on IMSs that mainly deal with audio. However, there still remains a large amount of signal processing tasks with different rates and timing constraints, such as spectral processing, physical modelling of acoustical models, spatialization, or transactional processing to query sound files in databases. We aim at designing an uniform model to handle multiple rates.

Audio graphs. In IMSs, the interconnection of heterogeneous sound processors naturally leads to the concept of audio graphs to model the dependencies among tasks. In audio graphs, audio processing is performed by nodes that exchange the audio signals through ports, called inlets and outlets in Max/MSP [Zic02] and Puredata [Puc+96]. This audio graph model suits particularly well IMSs with unit generators as the base processing unit, but is also relevant for systems such as Faust that internally transforms [OFL04] the program representation into a graph of operations on signals flowing on the edges. It stems from the more general *dataflow paradigm* [Liu+91] where nodes are connected together and exchange data, called *tokens*. When they have enough tokens on their inputs, they can be fired and generate a certain amount of tokens on their outputs. In the context of audio graphs, tokens are audio samples. In addition to the audio signals, the audio processing nodes are also connected to control parameters and can themselves generate control parameters for another node. Depending on the case, this graph can be statically defined or can also be reconfigurable during execution.

Precision of control. If we look at the audio signal more closely, we see that it becomes a discrete signal after entering the computer and going through the analog to digital conversion, *i.e.*, it is sampled. *Samples* are values of a signal measured at precise moments in time, and usually periodically. The periodicity of the measurement defines the *sample rate* or *audio rate*, the number of mea-

[1] This is not true with some modern compression schemes, where a video is encoded with key frames and with indications on how to deduce surrounding frames from the changes of the frame.

surements per second. Hence, the audio signal is seen as a *stream* of samples and some languages, such as Faust, describe signal processing as operations on one sample at a time. For performance reasons, these samples are grouped into *buffers* (also called blocks), and processed together at the same time. Apart from the values of the signal, one may get the frequency spectrum of the signal, which is usually computed on overlapping windows of the signal. The result can also be seen as a stream with a lower rate (depending on the successive shifts of the windows) that carries buffers of spectral bins.

On the contrary, control events are aperiodic and furthermore do not necessarily coincide with a multiple of the audio rate. Therefore, IMSs strive to reconcile those various known rates and the unpredictability of the timings of the control. Indeed, controls can be taken more or less precisely into account, immediately at the next sample in time, or at the next block, as in Figure 1.2. The delay in acknowledging the control for the audio processing is called *control latency*. The precision also depends on the signal processing task in use: some require sample accuracy, such as granular synthesis or some physical models; other behave correctly with only block accuracy; others need their control parameters to be smoothed.

In this work, we attempt to develop a common formal model with *time* explicitly represented, handling multiple rates, buffering, and the subtle handling of control and audio, as we think it will help:

- to better characterize the control precision in IMSs;
- to design more efficient and precise architectures.

Precision of the audio signal. IMSs are often deployed on mainstream operating systems such as Windows, macOS or Linux, which are not real-time systems but rather best-effort systems, meaning that there is absolutely no guarantees that audio and controls will be delivered at the right time or will be late due to interferences with other programs. Indeed, the IMS has to coexist along with other applications such as a web browser. The complex temporal scenarios driven by sensors capturing the physical environment, as well as the human-in-the-loop, lead to high unpredictability. It entails high-load situations where many audio processing effects compete for the processor. That has brought composers and musicians to use more powerful machines, and programmers to optimize their IMS. One of the current trends is to parallelize the processing tasks to put into use the multicore processors of today.

To tackle this problem, we rather want to take advantage of another limitation of the human auditory system: human beings cannot hear sounds whose

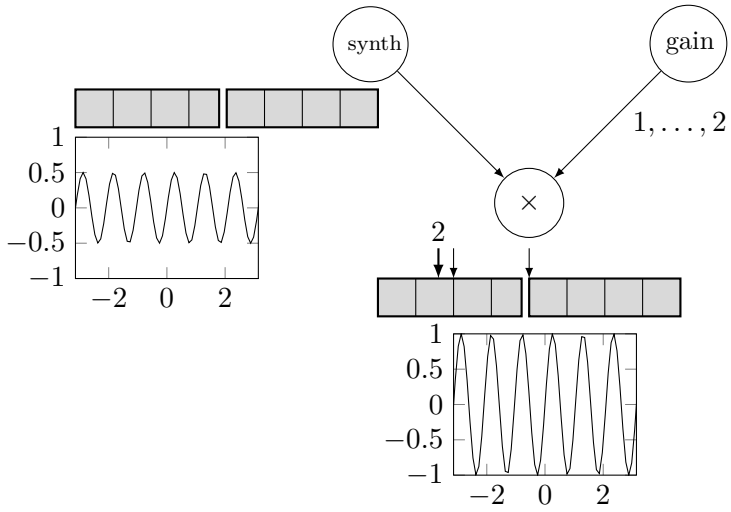


Figure 1.2.: An audio graph processing signals, seen as streams of buffered samples, with control parameters. A synthesizer generates an audio stream and its result is multiplied by a gain. Depending on when the gain is sent, and when it is applied, the shape of the sinusoid out of node × will be different. The first arrow on the left shows when gain 2 is changed. The second one corresponds to sample accuracy, and the third one, to block accuracy. If the change in gain between 1 and 2 is not smoothed, there will be a discontinuity in the signal.

frequency is more than 20 kHz and more generally, do not hear well the higher range of the spectrum. It means that multirate can be leveraged to decrease the amount of computations to perform. For instance, downsampling by 2 typically results in half of the computations. Actually, the precision of control can also be seen as resampling problem: is the control at the rate of the audio, or is it downsampled down to the rate of a block? Or is it not downsampled at all but rather new audio samples are created to reach the so-called *subsample accuracy*?

In this work, we will explore the trade-off between performance and precision, where precision is understood either as precision of the control or precision of the signal.

Contributions

A formal model for audio graphs in IMSs with multirate streams

We develop a formal model of IMSs where signals with buffers are first-class citizens. The model represents signals (whether they are audio signals or control signals) as a stream of timestamped buffers. These signals are processed by typed audio nodes connected together in a graph. We introduce a type system that can handle multiple rates, *i.e.*, constraints on the timestamps of the buffers, as well as the buffering in the stream, or aperiodic streams. The type system can also distinguish between sample-accurate nodes and block-accurate nodes. We present a denotational semantics of the execution of an audio graph on the input streams, and characterize precisely, depending on the types of the nodes, how and when aperiodic control is applied to an audio stream. That formalization is the subject of Part I.

An audio extension for Antescofo

Antescofo [Con+12] is a score follower and a programming language. The Antescofo IMS lacked audio processing: it is embedded into Max/MSP or Puredata and delegates all audio processing to its host. We have developed an audio extension inside Antescofo, which has inspired the formal semantics. The audio extension is high-level, *i.e.*, it connects (potentially dynamically during execution) blackbox nodes, code in Faust or C++, annotated with types that describe how they can be connected together. The audio extension is described in Chapter 6 of Part II.

Offline and online optimizations of audio graphs

We explore the tradeoff between audio signal precision and performance. We show how we can downsample parts of an audio graph, and how to choose these parts, given time constraints, while maximizing some quality model. It works as a kind of compilation pass that can take an audio graph of an IMS, for instance a Max patch, and output an optimized version of the patch. These optimized versions can then be used during the execution of the graph directly, or swapped to replace the non-degraded graph in case of a permanent overload. We also explore heuristics that can choose subgraphs to degrade at the time of execution, at the middle of an audio cycle, in case of transient overload. Offline and online optimizations are presented in Chapters 7 and Chapter 8 in Part II.

Outline

In Chapter 2, we present an overview of Interactive Multimedia Systems (IMSs), challenges in IMSs, and a comparison of the main IMSs. We also focus on real-time aspects of IMSs and on optimizations, especially parallelism. We also present the real-time paradigms that are often used to describe IMSs.

Part I describes a model of audio graphs and audio and control streams with arbitrary rates. Chapter 3 introduces the objects that we formalize, including the domain of streams. In Chapter 4, we present a syntax for audio graphs and a type system that indicates how nodes can be connected together and how streams are processed by the nodes. The execution of the audio graph on streams is formalized with a denotational semantics in Chapter 5. In this chapter, we also compare our approach to other formalization of IMSs and languages for signal processing.

Part II presents more practical work. In Chapter 6, we describe an audio extension for Antescofo. It brings signal processing in an integrated way to Antescofo and makes it possible to connect annotated heterogeneous nodes coded in Faust and in C++ ; we present two real-case studies. An optimization of audio graphs, the resampling of chosen parts of the graph, is described in Chapter 7 (offline) and in Chapter 8 (online). In Chapter 7, we introduce models of quality and execution time and we evaluate some of our strategies in real experiments. In Chapter 8, we present heuristics to be used during the execution of an audio graph, and we compare our strategies to other adaptive scheduling strategies.

Chapter 9 discusses the contributions and their limitations and suggests some perspectives.

Chapter 2, Part I, Chapter 6, Chapter 7 with Chapter 8 and Chapter 9 can be essentially read independently. However, we advise to read Part I before Chapter 6.

2. An overview of Interactive Multimedia Systems

Interactive Multimedia Systems (IMS) [Row93] are programmable systems that combine audio and video signal processing with control in real time. In the sequel, we restrict our focus mostly on music- or audio-specific IMSs. At run time, during a concert, they process or synthesize audio signals in real time, using various audio effects. For that purpose, they periodically fill audio buffers and send them to the soundcard. They also make it possible to control the sound processing tasks, with aperiodic control (such as changes in a graphical interface) or periodic control (for instance, with a low-frequency oscillator). Audio signals and controls are dealt with by an *audio graph* whose nodes represent audio processing tasks (filters, oscillators, synthesizers...) and edges represent dependencies between these audio processing tasks. Sometimes, this graph can be dynamically modified, *i.e.*, nodes can be added or removed during execution.

Puredata [Puc02a] and Max/MSP [Zic02] are examples of IMSs. They graphically display the audio graph, but modifying it at run time as a result of a computation can be complicated. Other IMSs, such as ChucK [Wan09] or SuperCollider [McC96], are textual programming languages. They are also more dynamic. In Antescofo [Ech+13], human musicians and a computer can interact on stage during a concert, using sophisticated synchronization strategies specified in an augmented score, programmed with a dedicated language that can also specify dynamic audio graphs [Don+16]. Commercial software such as Ableton Live, Cubase or ProTools can also be qualified as IMSs. We present a classification of IMSs in Section 2.1.

IMSs are real-time systems, with real-time constraints: audio must be sent to the sound card periodically before a deadline. If the deadline is missed, we hear a click, as the audio human ear is sensitive to audio gaps and glitches. We present the real-time challenges of audio in IMSs in Section 2.2.

More and more complex pieces, as well as the trend to put the electronics of the pieces on small embedded cards, have led to various optimizations in

IMSs: parallelizing or vectorizing the audio graph for instance [Ble11; BFW10; Kie+15; Cam15], explicitly or implicitly, using static [OLF09] or dynamic strategies [LOF10], or more ad-hoc and handcrafted solutions. We present these optimizations, with a focus on parallelization, in Section 2.3. We also present formal models developed for IMSs, especially to describe the interactions between control and signal processing, and the precision of the timing, in Section 2.4.

2.1. A bestiary of Interactive Multimedia Systems

Interactive Multimedia Systems [Row93] combine audio and video processing, but also light, gesture, or movement. They provide a way of controlling the processing of those signals in *real time* by reacting to events coming from the computer system or from the outside environment thanks to sensors, possibly with a human-in-the-loop. Inputs and outputs can be audio signals, MIDI [Loy85] events or OSC [Wri05] messages. Events drive audio processing, from sound transformation to spatialization. They can trigger processes or cascade changes of parameters.

We do not deal here with *out-of-time* systems that can handle audio processing and keep track of time, but do not run in real time. For instance, Computer-aided Composition (CAC) systems where a composer creates a score and an audio file such as OpenMusic [BAA11] are not considered as IMSs.[1]

Here, we will focus on systems that deal at least with audio. Some also add video and other media.

We start by identifying classifying criteria for IMSs; then we will present a few ones in more details.

Programmable or not. We distinguish between IMSs that can be programmed such as Max/MSP and *PureData* and IMSs that are delivered "batteries-included", with a strict workflow, such as commercial Digital Audio Workstations (DAWs), for instance, Ableton Live.[2]

Visual versus textual. We separate IMSs that use a graphical paradigm, for instance the Patcher paradigm of Max/MSP and *PureData*, and IMSs that

[1] Although the difference is somehow becoming less and less relevant, as recent versions of OpenMusic offer a reactive extension [BG14] that makes it possible to create real-time interactions, while IMSs such as Max/MSP start to offer subsystems for out-of-time composition of scores, with Bach [AG12].

[2] `https://www.ableton.com/`

are programmed textually, which we can call *musical programming languages*, such as Faust or Supercollider.

Interactiveness. We rank IMSs on how much can be controlled in the language, such as which protocols can be used and how it handles a *human-in-the-loop*. For instance, i–score [ADA08; Cel+15] allows musicians to write complex abstract temporal constraints to describe interactions between musicians and electronics, using the OSC protocol to communicate. Antescofo [Con10] can also specify complex scenarios where the musician tempo is fed back to the system to adapt its temporal awareness. We also indicate how dynamic the computations are, *i.e.* if the audio graph can be easily reconfigured during execution.

Scheduling of events. This criterion is more technical and deals with how precise control events are taken into account: are they block-accurate, sample-accurate, subsample-accurate and in which circumstances? IMSs deal with audio and video streams, where a buffer has to be filled periodically, and with controls that can be aperiodic (a GUI change) or periodic (a low-frequency oscillator). How to articulate control and processing is one of the challenges of scheduling an IMS. Control can take too much time and delay audio processing. An audio task often processes by chunks (or buffers, or blocks) to use the vector instructions of the targeted processor, so that control cannot occur inside such a chunk, as shown in Figure 2.1. However, for some audio processing or synthesis, for instance when the phase must be accurate or in some physical synthesis, and also in granular synthesis, the timing of a control event must be taken into account at the sample level, which is more precise than the block level.

Multiple rates. When dealing with multimedia, being able to tackle multiple rates is crucial: audio is usually driven at a 44 100 Hz rate while a typical video uses 24 Hz. Some controls are also driven by low-frequency oscillators. Even only in audio, several rates are useful, for instance for a spectral analysis using overlapping windows of the signal with a lower rate than the sample rate.

Level of abstraction. We consider here whether the system is mainly used to define low-level signal processing tasks or rather high-level abstractions to compose the signal processing tasks and interaction building blocks.

Imperative or functional, implicit-time or explicit-time model. In the *implicit-time paradigm*, programs are built by composing operations on sig-

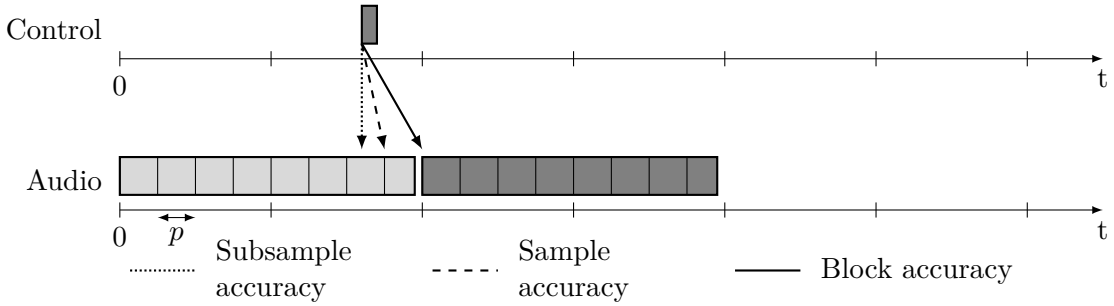


Figure 2.1.: Three possible ways of dealing with a control event occurring during audio processing: at the boundaries of a buffer, at the sample level, or with a subsample accuracy. Audio is precise with an accuracy of p, the sample period. This is the sample accuracy. Block accuracy is $8 \times p$ here, as we have 8 samples per block.

nals as a whole and do not directly refer to individual samples. They provide two primitive operands to build sample-shifted signals to navigate in time, `delay` and `feedback`. Faust [OFL04] is a language that uses this paradigm. The *explicit-time paradigm* does not sample time but uses an explicit duration in physical time, such as advancing 2 s as in ChucK [Wan09]. We also say that an IMS uses *explicit time* when it defines signals as a set of recursive equations where a time index appears explicitly, such as in the Alpha language [Cha+04; LMQ91] or the Arrp language [Leb16]. *Actors* are the building blocks in the *dataflow* model of computation [Bha+18], where these building blocks are thought as autonomous parallel communicating entities. They represent signal processing blocks, connected together with *channels*, with *tokens* flowing on the channel representing the signal. Actors are amenable to the *functional paradigm* of programming. Other languages, such as Faust, use combinators to represent the audio graph and are also *functional*. On the contrary, languages with mutable variables and side-effects, such as Antescofo, are *imperative*. We give examples of the combinations between those paradigms in Table 2.1.

	Implicit-time	Explicit-time
Imperative	Threads	Antescofo
Functional	Max/MSP, Faust	Arpp

Table 2.1.: Classifying IMSs according to the model of time and the programming paradigm.

IMS	Visual	Interactive	Control	Multirate	Abstraction	Prog. model	Time-model	Remarks
Ableton	visual		sample-accurate (automation)	no	high-level			DAW
Bitwig Studio	visual		idem	no	high-level			DAW
Max/MSP [Zic02]	visual	OSC, messages	block-accurate	one audio rate	high-level	functional	implicit	sample-accurate with Gen
Puredata [Puc97]	visual	OSC, messages	block-accurate	One audio rate	high-level	functional	implicit	Change block-size with `block~`
ChucK [Wan09]	textual	OSC	sample-accurate	No	both	imperative	explicit-time	
Faust [OFL09]	textual	OSC	block-accurate, sample-accurate for MIDI	experimental [OJ16]	low-level	functional	implicit-time	
SuperCollider [Ben11]	textual	live-coding, OSC, dynamic	block-accurate	One audio rate	high-level	imperative	explicit-time	client-server
PWGLSynth [LNK05]	visual	OSC	sample-accurate	Not directly	high-level	functional	implicit-time	Embeds Kronos
Marsyas [LT14]	textual		sample-accurate	multirate	both	imperative	implicit-time	scripting language and C++ lib
Antescofo [Con10]	textual	OSC, score follower, dynamic	sample-accurate for Curve	multirate	high-level	imperative	explicit-time	
ossia [CDC16]	visual	OSC	sample-accurate (libAudioStream)	one audio rate	high-level	functional	explicit-time	ossia = successor of i–score [ADA08]
Nyquist [Dan+93]	textual		control-rate	one audio rate	high-level	imperative	explicit-time	
Arrp [Leb16]	textual		sample-accurate	multirate	low-level	functional	explicit-time	
Sig [TL15]	textual		possibly	multirate	low-level	functional	implicit-time	
LuaAV [WSR10]	textual	OSC, dynamic	sample-accurate	one audio rate	high-level	imperative	explicit-time	synths coded in C or csound
exTempore [Sor18]	textual, dynamic	live-coding	sample-accurate	one audio rate	high-level	functional	explicit-time	opensource of Impromptu
XTLang [Sor18]	textual	live-coding	sample-accurate	one audio rate	low-level	imperative	explicit-time	used with exTempore
Csound [VE90]	textual	score	`ksmps`-accurate	one audio rate	high-level	functional	implicit-time	`ksmps` can be set to 1 sample.
Kronos [Nor15]	textual	dynamic	sample-accurate	multirate	low-level	functional	implicit-time	
LC [NIS14]	textual		sample-accurate	multirate	low-level	imperative	explicit-time	
Serpent [Dan02]	textual	OSC, O2, Aura msgs		one audio rate	high-level	imperative	implicit-time	real-time GC
Aura [DB96]	textual	OSC, O2, dynamic	block-accurate	sub-multiples of audio rate	high-level	imperative	implicit-time	

Table 2.2.: Comparison of 22 IMSs. All of the IMSs here are programmable, except Bitwig Studio and Ableton (but it has now an embedded version of Max, Max for Live). For the multirate criteria, we analyze whether there are multiple audio rates or not. O2 is an extension of OSC [DC16]. *Prog. model* refers to programming model. If a column is not filled for a specific IMS, it means either that we could not find the relevant information or that it does not apply to the IMS.

There are dozens of IMSs. We give more details about some them in the section, chosen for their popularity and representativeness, and compare succinctly about 20 of those in Table 2.2. This is not by far an exhaustive list. For instance, we do not speak about the oldest musical programming languages.

2.1.1. The Patcher family

The Patcher family, with Max/MSP [Zic02], PureData [Puc97], jMax [Déc+99], MAX/FTS [Puc02b] originally developed by Miller Puckette at Ircam [Puc88], emphasizes *visual programming* for dataflow processing. Functions are represented by boxes, placed on a screen called canvas. Boxes are connected together with links, and data flow from one object to another through these links. Functions can process audio signals, process control, and some boxes are also control-flow structures. Here, we give more details about *PureData*, which is open-source.

In PureData, actions can be timestamped by some boxes, for instance the boxes `metro`, `delay`, `line`, or `timer`. GUI updates are also timestamped. Other incoming events are MIDI events, with `notein` and OSC. The scheduling in Puredata is block-synchronous, *i.e.* control occurs at boundaries (at the beginning) of an audio processing on a block of 64 samples, at a given sample rate. For a usual sample rate of 44.10 kHz, a scheduling tick lasts for 1.45 ms. Depending on the audio backend,[3] the scheduler can be a polling scheduler (see Figure 2.2) or use the audio callback of the backend.

PureData can handle multiple block sizes and overlapping blocks by changing them in a subpatch using the `block~` box.

2.1.2. SuperCollider

SuperCollider [Ben11] is an interactive multimedia system mainly used for audio synthesis and for live coding.[4] SuperCollider uses a client/server architecture (see Figure 2.3): on the client side, an object-oriented and functional, Smalltalk-like language makes it possible to compose the piece, and generates OSC messages which are sent to the server. OSC messages can be timestamped and in this case, they are put in a priority queue and executed on time, or can be sent without a timestamp and are then processed when received. The SuperCollider server processes audio through an ordered tree of unit generators

[3] Jack or ALSA on Linux, CoreAudio on macOS, or WASAPI on Windows, for example.

[4] Live coding consists of creating a musical piece at the time of the performance by coding it in a DSL in front of the attendance: the addition, deletion or modification of the lines of code is usually projected as the musical piece is created.

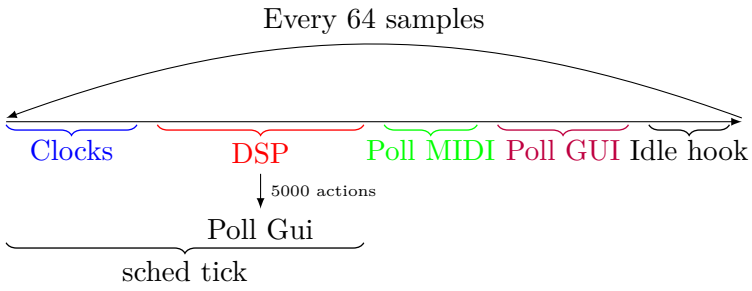


Figure 2.2.: Scheduling cycle in Puredata (polling scheduler) A cycle starts by executing all the timestamped operations events (handled by clocks) then the actual signal processing is performed. After that, MIDI events then GUI events are taken into account. The idle hook is a custom processing than the user can add. For the GUI not to be too unresponsive, after 5000 DSP actions, the GUI is polled for events.

(UGen) for analysis, synthesis and processing. It can use any number of input and output channels. UGen are grouped statically in a higher-level processing unit, called a *synth*, as in Code 2.1.

```
{ SinOsc.ar(440, 0, 0.1) + WhiteNoise.ar(0.01) }.play;
```

Code 2.1: This simple program generates a sine at 440 Hz with phase 0 and an amplitude of 0.1, and white noise at the audio rate (*i.e.* `ar`), adds them, and plays them. The sine and the white noise are both *synths*. The *sclang* language implements the evaluation of this expression by sending OSC messages to the *scynth* server.

All events with timestamps lower than the tick size (or scheduling period) are executed at once. This tick size is by default linked to a block size of 64 samples. However, better accuracy can be achieved if a trade-off is made for more latency, by explicitly setting it with `s.latency = 0.2` for instance. Nevertheless, the client and the server do not share a sample clock, which makes it difficult to get very precise timings. The `OffsetOut` UGen can help starting new synths with sample accuracy. Another UGen, `SubsampleOffset`, can start a synth with subsample accuracy.

However, by default, only two rates are used in SuperCollider, control rate

(at block boundaries) called `kr` and audio rate, `ar`. Control values are interpolated linearly [McC02] to prevent a discontinuity in the control signal.

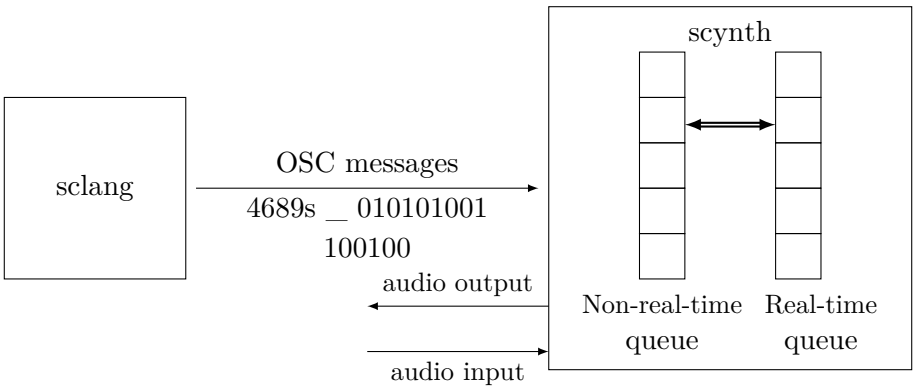


Figure 2.3.: Client-server architecture of SuperCollider.

The server uses three threads: a network thread, a real-time thread for signal processing, and a non-real-time thread for purposes such as reading files. Commands among threads are communicated via lock-free FIFOs.

There are three clocks in the language: `AppClock`, a clock for non-critical events that is allowed to drift, `SystemClock` for more precise timing, and not allowed to drift, and `TempoClock`, the ticks of which are beats synchronized with a fixed tempo. If the live coding performer changes the tempo, the actual next event date can be recomputed.

2.1.3. ChucK

Chuck [Wan09] is a *strongly timed* (*i.e.* with an explicit manipulation of time, thanks to variable `now`), synchronous, concurrent, and *on-the-fly* music programming language. It is mainly used for live coding. It is compiled to bytecode and executed in a virtual machine.

Time and event-based programming mechanism. Time can be represented as a duration (type `dur`) or a date (type `time`). The keyword `now` makes it possible to read the current time, and to go forward in time with the Chuck operator `=>`. Time can also be advanced until an event, such as a MIDI event, an OSC event, or a custom event, is triggered, as in Code 2.2.

```
1::second => now;//advance time by 1 second

now + 4::hour => time later;
later => now;//Advance time to date now + 4 hours

.5::samp => now;//subsample advance

Event e;
e => now;//Wait on event e
```

Code 2.2: Advancing time by assigning `now` in ChucK.

`now` allows time to be controlled at any desired rate, such as musical rate (beats), as well as sample or subsample rate, or control rate.

Chuck makes it possible to use an arbitrary control rate (control occurs after the assignment to `now` stops blocking) and a concurrent control flow of execution. Audio is synthesized from the audio graph one sample at a time.

The audio graph is composed of UGens dynamically connected together with the ChucK operator, as in Code 2.3. Unit generators can also be dynamically disconnected with the UnChucK operator `=<`.

```
SinOsc s => Gain g => JCRev r => dac;
```

Code 2.3: A linear chain composed of a sinusoidal oscillator, a gain, a reverb, and then an output to the soundcard.

Cooperative scheduling. ChucK processes are cooperative lightweight user-space threads called *shreds* (and are scheduled by a *shreduler*); they share data and timing. When time advances thanks to `... => now`, the current *shred* is blocked until `now` attains the desired time, giving room to other *shreds* to execute. *Shreds* are synchronous (sample-based clock at 44.10 kHz in general) and *shreduling* is deterministic: instantaneous instructions in every *shred* are executed until reaching an instruction that next advances in time. Shreds are *shreduled* using a queue of shreds sorted by waiting time before wake up (see Figure 2.4).

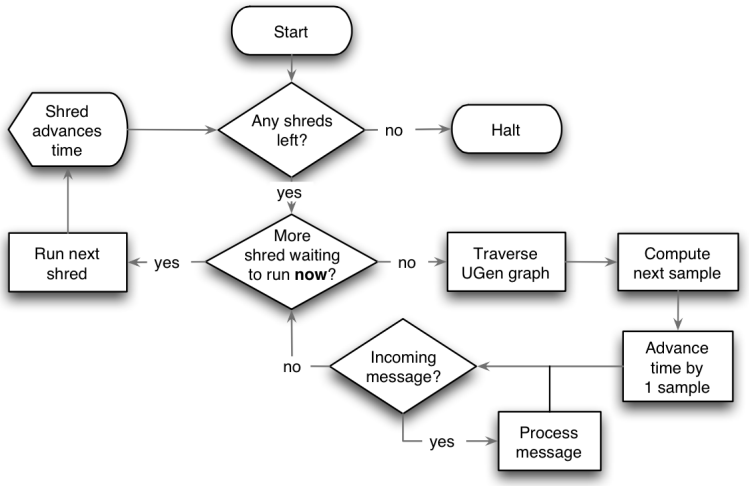


Figure 2.4.: Shreduling in ChucK. Figure from [WCS15].

2.1.4. LC

LC [NIS14; Nis18b] is a *mostly-strongly timed* programming language. It is inspired by ChucK and also exhibits a `now` special variable to advance time. It puts forward adding an explicit switch between a synchronous and non-preemptive context, and an asynchronous and preemptive context (*i.e. mostly-strongly timed*) to be used when audio tasks would be at risk of taking too much time and outputting audio too late, using keywords `sync` and `async`. The two contexts can be nested. For instance, loading a wavefile will be undertaken in an asynchronous context. LC also provides strategies to handle timing violations with the `within(duration) ... timeout handling`.

The language and runtime also make it possible to do sample-accurate control and start an audio processing with sample accuracy.

2.1.5. Imprompu/ExTempore

ExTempore [Sor18] (called Imprompu before being opensourced) is a music programming language dedicated to live coding, and according to its author, *cyberphysical programming*. It is composed of a Scheme interpreter and a XTLang just-in-time compiler using a LLVM backend. XTLang is used to program low-level audio processing effects while the ExTempore Scheme language enables the live coder to organize its performance in time. Advancing in time is

approached through *temporal recursion*, *i.e.* functions that call themselves in some specified time in the future, as in Code 2.4. A relative deadline for the function can also be specified.

```
(lambda (i)
    (println 'i: i)
    (if (< i 5)
        (callback (+ time 4000) my-func (+ i 1)))))
```

Code 2.4: Temporal recursion in ExTempore: a function reschedules itself to be executed 4000 samples later.

The audio processing is *in fine* handled in a special XTLang function `dsp` that processes one sample at a time.

2.1.6. Faust

Faust [OFL09] is a functional synchronous domain-specific language dedicated to digital signal processing, which compiles to C++, directly to machine code using LLVM as a backend, or to webassembly [Haa+17; LOF18b] to be deployed on the web and executed in a web browser. Faust code can also be easily deployed on an Android phone [Mic13]. It also aims at facilitating the creation of audio plugins for digital audio workstations, and thus provides a way of defining a graphical interface or to communicate to the audio process via OSC [FOL11]. Faust first generates the signal processing code and then injects it into an *architecture wrapper* which defines the actual implementation of the graphical interface, inputs and outputs with the soundcard and so on.

In Faust, signals are combined with operators that either compute one sample of each signal at a time, or delay the signal in time. In Code 2.5, operator `+` adds two signals together. Expressions are called *block diagrams* and are combined with routing operators:

- parallel `A,B`, to execute to two block diagrams simultaneously, where the inputs of `A,B` are the inputs of A and B, and its outputs, the outputs of A and B;
- sequential `A : B`, which connects the outputs of A to the inputs of B;
- split `A <: B`, which distributes the outputs of A to the inputs of B, when B as more inputs than A has outputs;
- merge `A :> B`, which mixes the outputs of A into the inputs of B, when A has more outputs than B has inputs;

- recursion, `A ~ B`, when some outputs of B are connected, with a one sample delay, to some inputs of A.

An *infix notation* (traditionally used for numerical operators) and a *prefix notation* (common for function application) are also available:

```
2*3 //infix
2,3: * //combinator
*(2,3) //prefix
```

Macro operators can be used to duplicate expressions or build new expressions. For instance, `par` duplicates an expression in parallel and can build new versions of the expression that depends on an iteration variable. A delay in samples can be written with `x@n`, meaning that signal `x` is delayed by n samples.

Audio is processed sample by sample in the language, but after compilation, it is processed in buffers (32 samples by default) within the generated `compute` method, and control only occurs at the boundaries of buffers. More recently, Faust has been able to handle MIDI events with sample accuracy [Let+17], by using the timestamps of the MIDI messages. In that case, the MIDI events are dealt with one buffer of latency.

2.1.7. Antescofo

We present here in more details Antescofo, an IMS that is embedded in a host IMS (typically PureData or Max/MSP). Antescofo harnesses the audio effects of the host environment. One of the goals of the thesis is to embed directly digital signal processing in Antescofo (see Chapter 6), and to formalize the interaction of control and audio computations (see Part I).

Antescofo [Con10] is an IMS for *musical score following* and *mixed music*. It is an artificial musician that can play with human musicians in real time during a performance, given an augmented score prepared by the composer. Since 2008, Antescofo has been featured in more than 40 original mixed electronic pieces by well-known ensembles and composers, such as Pierre Boulez, Philippe Manoury, Marco Stroppa, the Radio France orchestra and Berlin Philharmonics.

Figure 2.5 shows the architecture of the system, in two components: a *listening machine* and a *reactive machine*. The *listening machine* processes an audio or MIDI stream and estimates in real time the *tempo* and the *position* of the live performers in the score, trying to conciliate performance time and score time. Position and tempo are sent to the *reactive machine* to trigger

```
import("music.lib");

upfront(x)  = (x-x') > 0.0;
decay(n,x)  = x - (x>0.0)/n;
release(n)  = + ~ decay(n);
trigger(n)  = upfront : release(n) : >(0.0);

size        = hslider("excitation (samples)", 128, 2, 512, 1);



dur         = hslider("duration (samples)", 128, 2, 512, 1);
att         = hslider("attenuation", 0.1, 0, 1, 0.01);
average(x)  = (x+x')/2;

resonator(d, a) = (+ : delay(4096, d-1.5)) ~
    (average : *(1.0-a)) ;

process = noise * hslider("level", 0.5, 0, 1, 0.1)
: vgroup("excitator", *(button("play"): trigger(size)))
: vgroup("resonator", resonator(dur, att));
```

Code 2.5: A Karplus-Strong string model in Faust. Some graphical interface elements are defined in the code: `hslider`, `button`, and `vgroup` to group other GUI elements. Some signal processing elements are grouped into macro functions, such as `release(n)`, to be used later. The `process` instruction on the last line gives access to the audio inputs and outputs of the target architecture.

actions specified in the mixed score. These actions are emitted to an audio environment, either Max/MSP or *PureData*, in which Antescofo is embedded in a performance patch. As for human musicians, Antescofo relies on adapted synchronization strategies informed by a shared knowledge of tempo and the structure of the score. Those strategies include synchronizing only with the tempo, or only with the position in the score, or even on particular position targets in the score.

The mixed scores of Antescofo are written with a dedicated reactive and timed synchronous language, where events of the physical world (pitches and durations of notes played by musicians for instance), the part of the artificial musician, and the synchronization between them are specified. It has to take into account the temporal organization of musical objects and musical clocks (*tempi*, which can fluctuate during performance time) which are actualized in a physical performance time. This performative dimension raises particular challenges for a real-time language:

- multiple notions of time (events and durations) and clocks (tempi and physical time);
- explicit specification of the musical synchronisation between computer-performed and human-performed parts;
- robustness to errors from musicians (wrong or missed notes) or from the listening machine (recognition error).

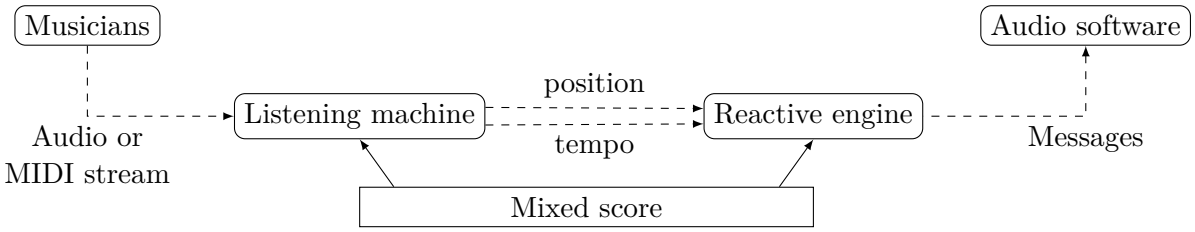
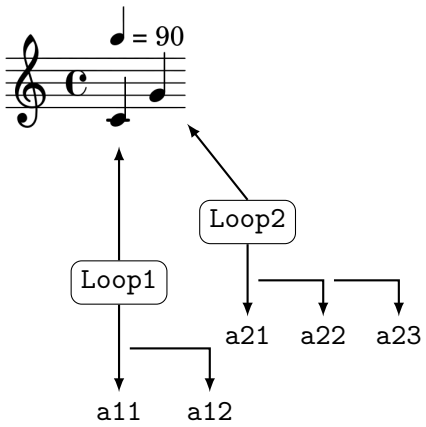


Figure 2.5.: Architecture of the Antescofo system. Continuous arrows represent pre-treatment, and dashed ones, real-time communications.

The Antescofo language

The Antescofo language is a synchronous and timed reactive language presented for instance in [EGC13; Ech+11; Ech15]. A full documentation is available at `http://antescofo-doc.ircam.fr`. We give here a quick description of the notions needed to grasp the proposed extensions in the following parts.

Instrumental events. The augmented score details events played by human musicians (pitches, silences, chords, trills, glissandi, improvisation boxes) and their duration (absolute, in seconds, or relative to the tempo – quaver, semiquaver...). An ideal tempo can also be given.




```
BPM 90
NOTE 60 1.0 note1
Loop Loop1 4.0
{   a11
    0.5 a12
}
NOTE 67 1.0 note2
Loop Loop2 4.0
{   a21
    0.33 a22
    0.33 a23
}
```

Figure 2.6.: An Antescofo augmented score: actions `a11` and `a12`, with a 0.5 delay, are associated to an instrumental event, here, a C quarter note.

Actions. Actions (see Figure 2.6) are triggered by an event, or follow other actions with some delay. Actions follow the synchronous hypothesis (see Section 2.4.2), and are executed in zero time, and in the specified order.

Atomic actions. It is a variable assignment, an external command sent to the audio environment, or the killing of an action.

Compound actions. A `group` is useful to create polyphonic phrases: several sequential actions that share common properties of tempo, synchronization, and error handling strategies. A `loop` is similar to a `group` but its actions execute periodically. A `curve` is also similar to `group` but interpolates some parameters during the execution of the actions. Compound actions can be nested and inherit the properties of surrounding blocks unless otherwise explicitly indicated.

An action can be launched only if some condition is verified, using `guard` statements, and it can be delayed for some time after the detection of an event (or previous action). The delay is expressed in physical time (seconds or milliseconds) or relative time (beats).

Processes. Processes are parametrized actions that are dynamically instantiated and performed. Calling a process executes one instance of this process and several instances can execute in parallel.

Expressions and variables. Expressions are evaluated to Booleans, ints, floats, strings (scalar values) or data structures such as vectors, maps and interpolated functions (non-scalar values).

Variables are either global, or declared local to a group. The assignment of a variable is an event, to which a logical timestamp is associated. `$v` is the last value of the stream of values associated with `v`, and `[date]:$v` the value at the date `date`.

`whenever` statement. It makes it possible to launch actions when a predicate is satisfied, contrary to other actions which are linked to events sequentially notated in the augmented score.

Temporal patterns, *i.e.* a sequence of events with particular temporal constraints, can also be matched by a `whenever`.

Synchronization strategies and fault tolerance. One strategy, the `loose` strategy, schedules actions according to tempo only. Another strategy, the `tight` strategy, not only schedules according to timing positions, but also with respect to triggering relative event positions. There are also strategies that can synchronize according to a target in the future: the tempo adapts so that the electronic part and the followed part arrive on this target at the same time.

In case of errors (missed events), two strategies are possible: if a `group` is local, the group is dismissed in the absence of its triggering event; if it is `global`, it is executed as soon as the system realizes the absence of the triggering event.

Antescofo runtime

A logical instant in Antescofo, which entails an advance in time for the reactive engine, is either the recognition of a musical event, or the external assignment by the environment of a variable, or the expiration of a delay. In one logical instant, all synchronous actions are executed in the order of their declarations.

The reactive engine maintains a list of actions to be notified upon one of those three types of events. For actions waiting for the expiration of a delay, three waiting queues are maintained: a static timing-static order queue, for actions delayed by a physical time (in seconds, for instance); a dynamic timing-static order queue, for delays related to the musician tempo (in beats); and a dynamic timing-dynamic order queue, for delays depending on local dynamic tempo expressions. Delays are reevaluated depending on changes of tempo, so that the order of actions in the queue should change. The next action to execute is the action that has the minimum waiting delay among the three queues.

We will present an audio extension of Antescofo in more details in Chapter 6.

2.1.8. Current trends for IMS

The most recent IMSs such as ChucK, LuaAV, LC, Kronos or Arpp tend to foster highly synchronous approaches to favor a strong precision of control, up to the sample. Most of these languages, including Faust, implement some kind of compilation (JIT or static) and optimization mechanisms. More and more languages are developed to program at the low level, *i.e.* signal-processing effects, whereas these effects would have been programmed with general-purpose languages such as C or C++ before. Some IMSs actually come with two languages, one for the high-level description of timings and interactions, and one for the low-level programming of the audio processing, such as ExTempore [Sor18] and XTLang, or Meta-Sequencer [Nor16] and Kronos.

Another trend is to port an IMS from the computer platform (and especially, from macOS) to platforms more accessible to the general public, such as Faust to the web [Let+15] or Android [Mic13], or directed to the *maker community* with small embedded cards such as Puredata on Raspberry Pi [Dry15], or with the Bela platform [Mor+17], a card dedicated to audio signal processing.

2.2. IMSs as real-time systems

Audio samples must be written into the input buffer of the soundcard periodically. For a sample rate of 44.10 kHz, such as in a CD, and a buffer size of 64 samples, the audio period is 1.45 ms. The buffer size can range from 32 samples for dedicated audio workstations to 2048 samples for some smartphones running Android, depending on the target latency and the resources of the host system. It means that the audio processing tasks in the audio graph are not activated for each sample but for a buffer of samples.

IMSs are not safety-critical systems: a failure during a performance is not life-critical; it will not generally result in damages or injuries. However, audio real-time processing has strong real-time constraints. Missing a deadline for an audio task is immediately audible.

Buffer underflow The audio driver uses a circular buffer the size of which is a multiple of the size of the soundcard buffer. If the task misses a deadline, it does not fill the buffer quickly enough. Depending on the implementation, previous buffers will be replayed (*machine gun* effect) or silence will be played, which entails cracks or clicks due to discontinuities in the signal, as seen on Figure 2.7. A larger buffer decreases deadline misses but increases latency.

Buffer overflow In some implementations, filling the buffer too quickly can also lead to discontinuities in the audio signal, if audio samples cannot be stored to be consumed later.

On the contrary, in video processing, missing a frame among 24 images per second[5] does not entail a visible decrease in quality so that lots of streaming protocols [ABD11] accept to drop a frame. Therefore, real-time audio constraints are more stringent than for video. Yet, they have not been investigated as much as real-time video processing.

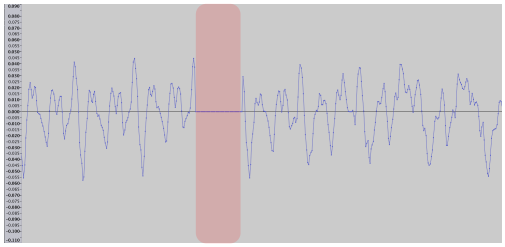


Figure 2.7.: The signal processor has missed its deadline. Thus, no audio sample generated by the processor during this audio cycle can be sent to the audio buffer of the soundcard. In this implementation, the system sends silence, *i.e.*, samples set to zero. It entails a discontinuity in the signal at the red strip, hence, a *click*.

Composers and musicians use IMSs on mainstream operating systems such as Windows, macOS or Linux, where a reliable and tight estimation of the worst-case execution time (WCET) is difficult to obtain, because of the complex hierarchy of caches of the processor, because there are usually no real-time

[5] Although missing a key frame in a compressed stream can be visible.

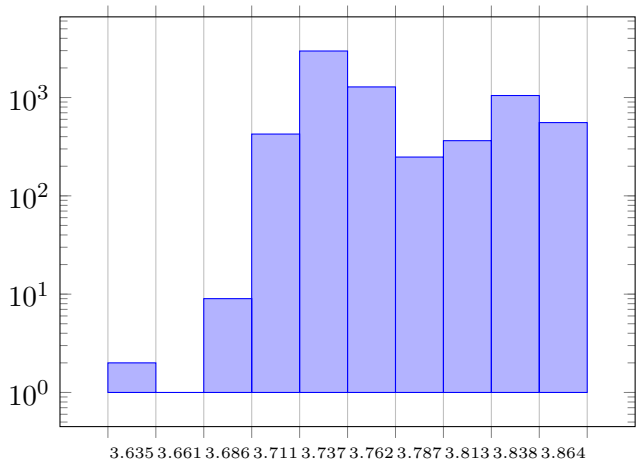


Figure 2.8.: Histogram of time budgets for the *audio callback* on a MacBook Pro with 16 GiB RAM and 3.10 GHz processor, with macOS Sierra. We execute a test program generating a sawtooth signal for 10 s. The time budgets range from 3.64 ms to 3.89 ms, *i.e.*, a 254 µs jitter, with a mean of 3.78 ms.

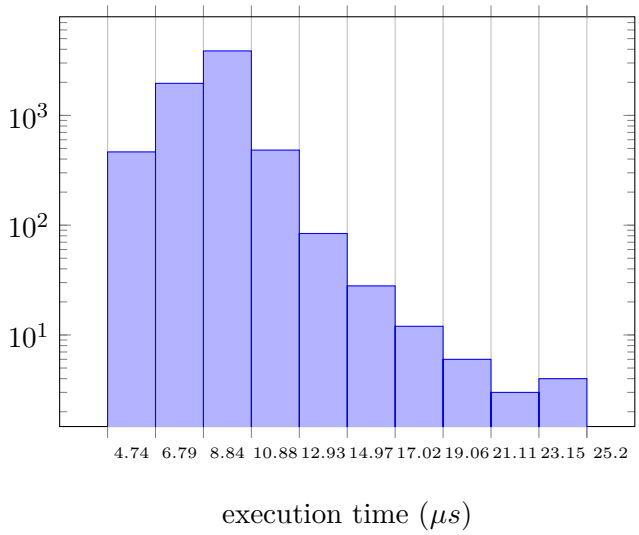


Figure 2.9.: Histogram of the execution time for the same code generating a saw signal for 10 s. Here, we show the execution time at each cycle in the *audio callback* for generating a sawtooth signal on a MacBook Pro with 16 GiB RAM and 3.10 GHz processor, with macOS Sierra. The execution times range from 4.74 µs to 25.20 µs with an average of 9.17 µs. The standard deviation is 1.59 µs.

scheduler or temporal isolation among tasks, and because it is difficult to predict which tasks will be executed at a given time. In addition, to spare energy or avoid heating, the CPU frequency can be dynamically adjusted, which is another important source of execution-time non-determinism. Those operating systems[6] are not real-time systems and do not offer any strong guarantee on deadlines for audio processing (see Figure 2.8) or on execution times (see Figure 2.9). Applications that perform audio computations have to compete for CPU and memory with other applications during a typical execution. It means that hard-real-time scheduling algorithms that depend on knowing the WCET cannot be applied to IMSs in the general case.

Furthermore, IMSs are more and more ported to embedded cards such as Raspberry Pi and have to adapt to limited computation resources on these platforms. Moreover, composers and sound designers create more and more complex musical pieces, with lots of dynamically interconnected nodes, requiring a sampling rate up to 96 kHz, for instance in Re Orso[7] by Marco Stroppa.

When real-time constraints are not critical, modifying the quality of service (QoS) by partially executing some tasks or even discarding them is an option to consider. In the case of IMSs, tasks are dependent, with dependencies defined with the edges of the audio graph. The quality of a task is itself *position-dependent*: it depends on the position of the audio processing task in a path going from an audio input to an audio output. It means one cannot merely discard or degrade any task in the audio graph to achieve an optimal QoS adaptation.

2.3. Optimization in IMSs

Larger pieces, with more and more audio processing effects, require increasing resources on the computer, as well as optimizing the audio processing.

Optimizations range from improving memory allocation in order to reduce memory consumption and cache pressure, such as in [Nis18a], to using special hardware such as DSP processors, with languages dedicated to program these hardwares such as Soul [JUC19] which intends to be a kind of generic *audio shader*. Pieces do not benefit from the same optimizations and exploring a set of possible optimizations, by benchmarking subsets of them for a given program as in [LOF18a], helps to find which ones are suitable.

Going from one rate to multiple rates is also a common optimization: not all

[6] There is an earliest-deadline-first scheduler [Fag+09] in Linux, but it's typically not activated on mainstream distributions.

[7] `http://brahms.ircam.fr/works/work/27678/`

computations should be performed at the quickest rate, which is usually the audio rate. Most IMSs have at least two rates, audio rate and control rate. Other ones allow for more rates [OJ16; Nor15], leading to a balanced choice between precision and high load, for high rates, or less precision but less load, for low rates.

Another common approach is to take advantage of multicore and multiprocessor architectures and of vector instructions, with *parallel computing* [Kum02]. Parallelism can be exploited explicitly, using dedicated instructions of the IMS, or automatically and implicitly. We give more details about parallelism in IMSs here.

Data parallelism. In data parallelism, the same computation is performed on several data available simultaneously close in memory. For example, for a vector addition, the same scalar operation (*e.g.*, scalar addition) is performed on all vector elements. Data are divided into components and the same task is run on the multiple components. The data can be distributed on several cores or processors with the same operation executed on them, or use special instructions, such as SIMD instructions [PH13], with Intel MMX [PWW97] or SSE [Lom11]. For instance, the `addps` MMX instruction operates on 128 bits at the same time, and can add together four 32 bit numbers with four other 32 bit numbers.

Task (or control) parallelism. Task parallelism consists of distributing computing tasks on multiple processors or cores. For instance, audio processing tasks can be assigned to different processors, usually using several threads. The dependencies between tasks reduce the amount of parallelism as a task needs to wait for the result of another task to start processing and so cannot be executed in parallel.

Pipelining. When dependent tasks operate on a stream of buffers, a task can start processing a new buffer of data while another task processes the previous buffer which was first processed by the first task, as shown in Figure 2.10. Pipelining increases latency, as the first buffer will be output at least one buffer late. The pipelining can happen at several levels, at a lower level than the stream of buffers for instance, to handle the processing of samples within a buffer.

Static vs. dynamic. The parallelization, *i.e.*, the mapping of computations to threads, can be performed *statically* before executing the tasks, by analysing the

dependencies between tasks, or *dynamically* during the execution of tasks, by observing how tasks communicate together. This dynamic approach is useful if the tasks are not known before execution and can be started or stopped at unknown times, but also if the execution times and communication times (access to memory, to the cache...) among tasks are not known beforehand.

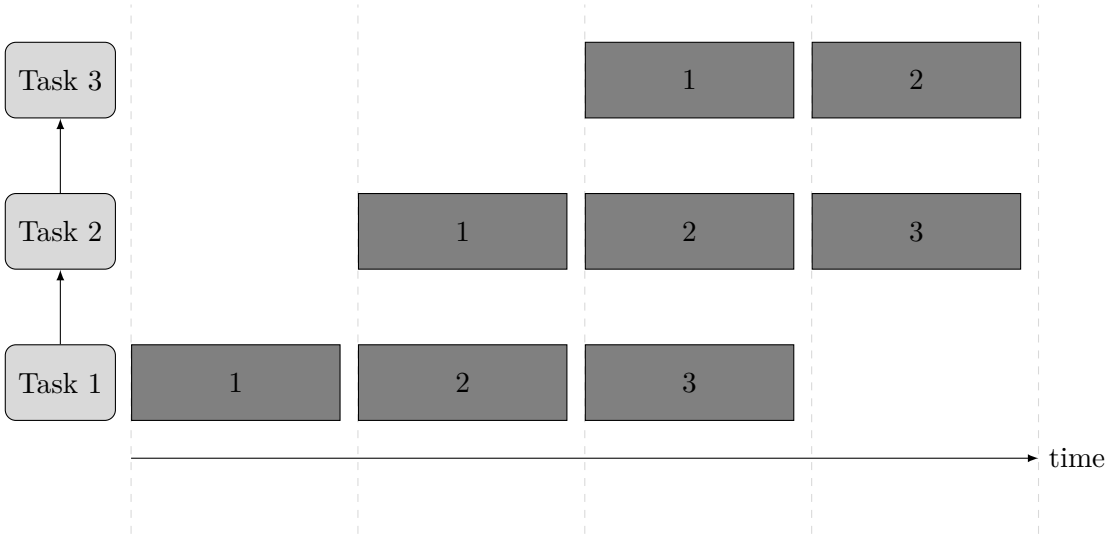


Figure 2.10.: Pipelining on a stream of three buffers with three dependent tasks, tasks 1, 2 and 3, where task 3 needs the results of task 2 and task 2, the results of task 1.

Manual ad-hoc parallelisation. Large musical pieces can often be manually divided into coarse-grained independent tasks and the independent ones are executed on several computers. For instance, several SuperCollider servers are spawned on several cores and different sound processing tasks are manually assigned to each of the servers.

Explicit instructions

Some IMSs do not automatically parallelize the audio graph but provide instructions to parallelize parts of it.

Max/MSP and `poly~` [7419]. `poly~ subpatch n` creates n instances of `subpatch` in order to parallelize them. It is first aimed at creating polyphonic synthesizers. Parallelism can be activated using the `parallel` attribute and the number

of threads can be set with the `threadcount` message. It achieves a rudimentary synchronization between subpatches using the `thispoly~` instruction inside a subpatch. It indicates whether the instance is busy or not and can receive messages, and can also stop and start signal processing in the instance using the `mute` message.

Puredata and `pd~` [Puc09]. The `pd~` object creates a Puredata subprocess within Puredata that communicates with the parent process using FIFOs which can carry audio as well as messages. It adds one buffer of latency. The `stdout` object in the subprocess is used to send messages to the parent.

Supercollider with supernova [Ble11] and `ParGroup`. SuperCollider, when using the ad-hoc server Supernova, provides an explicit instruction `ParGroup` to parallelize tasks. It considers that all the tasks that it is fed are independent, and it executes them in parallel.

Automatic parallelization

Some IMSs can automatically parallelize the audio processing.

Vectorization using SIMD instructions. The Kronos language [NL09] automatically vectorizes the signal processing code. Faust outputs C++ code [OLF09] organized such that a smart-enough C++ compiler would generate SIMD instructions. It works by breaking a single computation loop into several smaller loops easier to autovectorize. A more recent version of Faust uses an intermediate representation in which it uses the same small loop optimization and then generates machine code using LLVM [LOF13] and its autovectorizing capabilities.

Mapping tasks on multiple cores. Csound orchestra files provide information about the instruments and how they communicate through global variables, f-tables or the zak bus. In [ffi09; Wil09], an analysis of the orchestra file builds a dependency graph of the instruments based on the read and write access on the communication variables. Using a database of average execution times, instruments are clustered in order to gather quick instruments to prevent the overhead of assigning a quick instrument alone on a thread.

Faust automatically parallelizes [OLF09] the audio processing by inserting OpenMP instructions [DM98] at the code-generation phase. OpenMP is an API aimed at achieving high-level parallelism in Fortran and C/C++ programs. It

provides a special notation to specify regions of codes that can be executed on several processors. In particular, loops can be annotated with OpenMP in the program, and here are labelled automatically by Faust. The graph of regions is topologically sorted within OpenMP to find which loops can be computed in parallel.

In [LOF10], Faust can also parallelize using a work-stealing scheduler. A pool of worker threads accept tasks that are assigned to one of the threads and put in its queue. An idle thread without task to execute can *steal* a non-executed task in the queue of another thread in the pool. Optionally, it also provides automatic pipelining, by duplicating tasks and running them on a subpart of the audio buffer. Larger buffer sizes give better performance. However, the pipelining does not speed up the computations in most cases.

Using GPUs for audio processing

Graphical Processing Units (GPUs) are massively parallel architectures dedicated to data parallelism [Buc+04]. A GPU typically has thousands of cores and can execute many more threads. In [BFW10], a non-negative matrix factorization (NMF), often used to perform audio source separation [Vir07], is ported on a GPU using the CUDA API for Nvidia processors. However, it appears that the latency of copying the buffers to and from the GPU to the soundcard can be prohibitive. In [Bel+11], GPUs are used to achieve *massive convolution*, *i.e.*, many multiple convolutions on several audio channels to perform effects such as 3D spatial sound. The convolutions are based on Fast Fourier Transforms (FFT). The cost of transferring data between the GPU and the CPU is alleviated through a pipeline structure. GPU audio processing is limited to dedicated effects specially coded manually, although some attempts have been undertaken to automatically generate code for GPU for Faust and csound.

IMSs and parallelism

We compare in Table 2.3 various IMSs on parallelism. The audio tasks in IMSs are not always easily parallelizable as dependencies among the tasks reduce the number of tasks that can be executed at the same time. The overhead of the synchronization between threads is sometimes too costly for small buffers and so entails a tradeoff between long buffers and high latency, and small buffers but a higher overhead of the parallelism. Audio processing is not an *embarassingly parallel problem* [Dan08] in general, but polyphonic instruments for instance or processing on large buffers can benefit from it.

Optimizations we have described here preserve the semantics of the audio processing operations; they are low-level operational properties and should be ideally hidden from the composer's or programmer's point of view. In our work, we will consider optimizations that can change the semantics of the audio processing, by potentially degrading it. It is similar to the distinction made in image and audio compression between lossless and lossy compression: the first one encodes the image by just exploiting redundancies of the signal, such as in the Flac format [Xip19], and the second one takes advantage of psychoacoustics to discard parts of the signal that are less audible, such as in the mp3 format [Sta+93].

IMS	Explicit or implicit	Data or task	Description
csound	Automatic		`-jK` with `K` the number of parallel threads
Max/MSP	Explicit	Task	`poly~` instruction
SuperCollider	Explicit	Task	Use several servers
Supernova (Supercollider)	Explicit	Task	`ParGroup` instruction
Puredata	Explicit	Task	`pd~`
Kronos	Automatic	Data	Vectorization
Faust	Automatic	Data and Tasks	Vectorization, OpenMP, work stealing

Table 2.3.: Comparison of IMSs with respect to parallelism.

2.4. Formal models for IMSs

In this section, we describe formal models that are suitable to describe IMSs: the *dataflow paradigm* and real-time paradigms, including *reactive synchronous programming* and the *logical-execution-time paradigm*, which we illustrate with actual programming languages. In Chapter 5.3, we will describe some of these

paradigms and languages in comparison with the type system and formal semantics of Part I.

2.4.1. The dataflow paradigm

In the *dataflow model* [Pto14], computations are represented by a directed graph of nodes (or *actors*) that communicate using channels materialized by the graph arcs. Each channel is a FIFO queue of tokens. The *dataflow* model is *data-oriented*, *i.e.* when there are enough data (tokens) in each input channel of a node, the node is fired (activated), and consequently yields some tokens in its output channels. The number of tokens consumed or produced by a node in an execution is called *token rate*. The dataflow paradigm does not precise the order of execution.[8] How the token rates are defined, *i.e.* statically before execution or during execution, using some pattern or not, entails various species of dataflow models.

Synchronous dataflow

A dataflow graph is *synchronous* [LM87] if the number of tokens produced and consumed by nodes are known *a priori* and constant, as in Figure 2.11. This restriction ensures properties such as predictability or static scheduling.

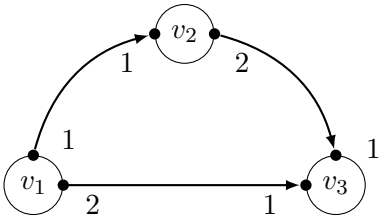


Figure 2.11.: A simple *synchronous dataflow* graph with three nodes v_1, v_2 and v_3. Data flows from v_1 to v_3, from v_1 to v_2 and from v_2 to v_3. v_1 yields 1 token per firing, and v_3 requires 2 tokens to be fired, one from v_1 and one from v_2.

If the number of tokens produced and consumed by every node is the same, the dataflow graph is said to be *homogeneous*. In the more general case with different numbers of tokens, the dataflow model is a *multirate* model.

[8] It is only constrained by the size of the channels holding the tokens, which cannot become negative.

This leads to a *balance equation* that states that the amount of tokens produced by an actor A and consumed by an actor B, as shown in Figure 2.12, must be the same after A is fired k_A times and B, k_B times:

$$k_A \times m = k_B \times n \tag{2.1}$$

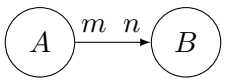


Figure 2.12.: Two actors A and B exchange tokens: m tokens are generated when A is fired and n tokens are consumed by B to be fired.

Such an equation can be added for each edge in the synchronous dataflow graph. For the graph of Figure 2.11, the system of balance equations is:

$$\begin{cases} k_{v_1} = k_{v_2} \\ 2 \times k_{v_1} = k_{v_3} \\ 2 \times k_{v_2} = k_{v_3} \end{cases} \tag{2.2}$$

The least positive (non-zero) integer solution is $k_{v_1} = 1, k_{v_2} = 1, k_{v_3} = 2$. It leads to a schedule that is statically computed. The schedule is, in order, the iteration of $v_1 v_2 v_3 v_3$.

More generally, if the system of balance equations has a non-zero solution, the synchronous dataflow (SDF) graph is said to be *consistent* [LS16] and the buffers to store the tokens are bounded for an infinite execution. Otherwise, it is *inconsistent* and the buffers are unbounded.

Another point to check for a dataflow graph is whether there are cycles. In the presence of cycles, actors do not know when to fire. Indeed, each actor in the cycle expects tokens to arrive in its inputs to be fired but those tokens will reach it if itself generates tokens, therefore entailing a *deadlock*. The typical solution to this problem is to have a special actor that produces initial tokens to ramp up the graph.

More complex dataflow models

In *dynamic dataflow* [LP95], the number of tokens produced and consumed is not constant. The number of produced tokens can depend on the number of consumed tokens and some actors can implement control flow such as conditionals. It has more expressive power, but finding deadlock or checking if token buffers are bounded is undecidable.

Some models are more expressive than SDF but remain decidable. *Cyclo-static dataflow* [Bil+96] allows production and consumption rates to change periodically. In *heterochronous dataflow* [GLL99], changes in production and consumption rates are governed by finite state machines. Given some conditions on the finite state machine, the model becomes decidable. This is a similar approach to *scenario-aware dataflow* [The+06; GS10], where each state of the automaton is a SDF graph. In *parametrized SDF* [BB01b], changes in consumption or production rates, but also initial values of delay actors, can be parametrized. A change of the parameters can only happen at some specific points in time during execution.

Another useful model for multimedia applications is the *multidimensional SDF* [ML02]. In this model, channels between actors carry multidimensional arrays of tokens.

Optimizing some metrics in the dataflow model is for instance studied in the *scalable synchronous dataflow* model [Rit+93]. There, the number of samples consumed or produced per activation can be multiplied by an integer factor. The goal is to minimize context switching and maximize vectorisation: samples are grouped together, but the signal is not resampled.

Audio graphs and the dataflow paradigm

The dataflow model has also been applied to describe real-time audio processing graphs, especially in IMSs. As the IMSs usually have control-flow nodes, *i.e.* conditionals, loops and so on, they are well modelled by the dynamic dataflow model where tokens are audio samples. It means that no static schedule can be computed but rather that execution is totally demand-driven, *i.e.* the arrival of tokens activates audio processing nodes. They also depart from the pure dataflow model in how they deal with control tokens. In Max/MSP, audio tokens always fire the node, but control tokens fire a node only if they are on a *hot inlet*. If they are on a so-called *cold inlet*, new tokens will discard the previous tokens, and only the last one on the cold inlet will be taken into account when the node is fired. In Puredata, in addition to hot and cold inlets, the order of the connections at the moment of creating the patch determines the order in which tokens are sent. Finally, the dataflow graph of those IMSs is traversed depth-first.

Besides, IMSs are real-time systems, which departs from dataflow graphs optimized for throughput. The *time-triggered dataflow model* [AA09] adds time (execution time, firing times, deadlines) to the dataflow model and is used to model a C++ multimedia library, CLAM [AAG06].

2.4.2. Real-time paradigms

We use the classification of real-time programming paradigms in [KS12] in three main models.

Bounded execution time (BET) or *scheduled model*. This is the model used in mainstream programming languages: time is not a first-class citizen, but they use functions that schedule operations to be performed at some instant. Execution times of programs have explicit deadlines.

Zero execution time (ZET) or *synchronous model*. The execution time of a program from input to output, including computations, is assumed to be zero.

Logical execution time (LET) or *timed model* or *time-triggered model*. Input and output are performed in zero time, but the computations in-between span a strictly positive fixed duration.

Here we will focus on the zero execution time and logical time paradigms.

Synchronous reactive languages

Real-time synchronous reactive languages are a common model to describe processing on real-time streams. They assume the zero execution time hypothesis, where processing units execute infinitely quickly or equivalently, execute in zero time. This hypothesis can be checked by computing the *worst-case execution time* of the tasks and ensuring it is smaller than the maximum worst-case execution time (the relative deadline) that we can allow for a correct execution. It is also *reactive* as it computes at least one reaction for any event and it is *deterministic* as it ensures that at most one reaction is computed for any event. Correctness analysis must check the absence of infinite cycles using causality analysis.

They also exhibit a powerful type system on streams, where types on streams are called *clock types*. A clock defines the streams of regular *logical instants* when a variable takes a value and can be over-sampled or under-sampled. They are a good fit for signal processing as they can describe multirate streams and the processing on the streams.

In addition, synchronous languages abstract the implementation details such as buffer management and audio callbacks, through the compiler. They ensure some safety properties that are granted when programming audio systems on the bare metal, especially synchronization properties. A much more detailed review on synchronous languages in computer music can be found in [BJ13].

The main synchronous reactive languages are Lustre [HLR92], Lucid Synchrone [Pou06], Esterel[BG92] and Signal [BLJ91]. We distinguish between two types of synchronous reactive languages:

- dataflow synchronous languages;
- process-algebra synchronous languages.

Dataflow synchronous languages describe computations as a set of equations on signals, and are as such well-suited for audio signal processing.

Lustre [HLR92]. Lustre is a synchronous dataflow programming language. It is declarative as outputs of programs are defined by unordered equations, called *node definitions*, as in Code 2.6.

```
node negate(x: bool) returns (b : bool);
let
    b = not a;
tel
```

Code 2.6: A node definition that takes one sequence as input, negates it and returns it.

The previous value of variable `p` can be accessed with `pre p`: the stream `pre p` is similar to stream `p` except for the first value which is undefined. The expression `q -> (pre p)` can be used to specify it: operator `a -> b` is a stream whose first value is the first value of stream `a` and then is equal to `b`.

Lustre can process several variables seen as an array using array iterators [Mor02].

Clocks. Two operators allow under-sampling, `when`, and over-sampling, `current`. In the code except 2.7, `when` creates a new stream where only the values of `x` when the expression `c` is true are kept (or sampled). Operator `current` on the sampled stream `y` projects it on the clock of `x` and uses the last previous value where `y` is not defined.

Compilation of Lustre programs. Lustre programs are compiled into sequential imperative C programs, with the following shape:

```
y = x when c
x + current(y)
```

Code 2.7: Clock under-sampling and over-sampling.

```
Init memory
Loop {
    read inputs
    compute outputs
    update memories
}
```

They can also be compiled into parallel programs and to circuits [RH91].

Lustre is packaged with a model-checking tool, *lesar*.

Lustre-like languages and buffering The *n-synchronous* Lucy-n [MPP10] language is an extension of Lustre that makes it possible to use clocks that are not necessarily synchronized, but *close* from each other, by using buffers, the size of which is automatically computed. For that, it uses a sub-typing constraint which is added to the clock calculus of Lustre. Another extension of Lustre that describes how to aggregate and disaggregate the elements of a stream is described in [Gua16].

The process-algebra synchronous languages are based on a more imperative approach: the new value of a signal is updated by emitting it and other processes react to the new value by waiting on it.

It is less straightforward to describe signal processing but convenient for controls.

Esterel [BG92]. Esterel is an imperative reactive synchronous language. It can both express parallelism and preemption. In the following example, `await` waits for signal `A` or signal `B`. If one of them is received, signal `O` is sent via the `emit` instruction.

```
[await A || await B];
emit O
```

Antescofo is actually inspired by Esterel, but adds the management of arbitrary delays. The underlying time model is hybrid because it handles both reactive events and timed transitions.

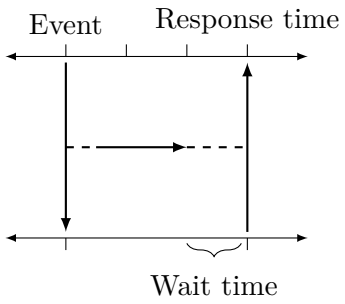


Figure 2.13.: Logical execution time: delay if actual execution finishes before the pre-fixed execution time. Adapted from [Kir02].

ReactiveML [MP05]. ReactiveML is an synchronous reactive extension for ML, here a subset of OCaml. ReactiveML compiles to OCaml code. In [Bau+13], it was used to implement a very simplified version of the Antescofo language.

Logical execution time

In the logical execution time [KS12], although inputs and outputs are read and written in zero time, the processing time itself of a task is a strictly positive fixed number. This is different from BET as this duration is not an upper bound but the actual duration of the execution. If the task finishes earlier, it waits until the fixed duration has elapsed, as in Figure 2.13. Analysis of causality is simplified compared to ZET thanks to the strictly positive duration of a task.

The main languages of this paradigm are Giotto [HHK01] and xGiotto [Gho+04]. Giotto is a time-triggered language with several control modes in which tasks coded in C are periodically executed. Switching between modes is also time-triggered. xGiotto adds event -triggering to Giotto.

Part I.

Formalization

In this part, we present a formal model of timed streams and their computation by an audio graph. The audio graph describes how at some date signal processing and control are dealt within an IMS. The audio graph combines audio processing nodes together with nodes distinguished as *sources* and *sinks* of the audio signal. Here, we formalize both the structure of the graph and the associated domains in Chapter 3 as well as the streams that flow between audio processing nodes in a audio graph, using a type system, in Chapter 4. We present a formal denotational semantics of audio graphs with streams in Chapter 5. We compare our model to models used for signal processing languages in Section 5.3.

Our motivation is to reflect actual buffered implementations where elements in streams, the samples, are grouped together into buffers. Processing nodes execute on buffers, not on samples. We also explain what happens when a control event occurs while a buffer is processed: is the control taken into account immediately, or rather at the next buffer? Indeed, buffering entails that some samples and some controls are dealt with later than their occurrence; this delay is called *latency* and we want to characterize it here. Furthermore, streams can be processed at different rates. We obtain a *buffered sampled representation of multirate signals with timestamps.*

3. Objects

In this chapter, we present the main objects we will employ to describe the types and semantics of an audio graph. In Section 3.1, we formalize the audio graph structure. In Section 3.2, we introduce the domain of streams. Finally, in Section **??**, we present the syntax of nodes and how to combine them in an audio graph.

Notations

We introduce here some operators that will be useful in the chapter.

Operator $\star$ is the Kleene star and $X^\star$ is the set of all finite sequences over set X, *i.e.*, $X^\star = \bigcup_{n\geq 0} \{x_1 x_2 \cdots x_n \,|\, x_1, x_2, \ldots, x_n \in \mathcal{U}\}$. Note that the empty sequence is in $\mathcal{U}^\star$.

Operator ω is used to define the set of infinite sequences, as follows: for a set X, X^ω is the solution of the equation $X^\omega = XX^\omega$.

The $\lfloor\cdot\rfloor$ operator is the floor function and $\lceil\cdot\rceil$ is the ceil function, such that for $k \in \mathbb{R}$, $\lfloor k \rfloor = \max\{m \in \mathbb{Z} \,|\, m \leq k\}$ and $\lceil k \rceil = \min\{m \in \mathbb{Z} \,|\, m \geq k\}$

3.1. Audio graphs

An audio signal is processed by various functions connected together: a transformation of the audio signal can be described through audio processing nodes in a graph that takes signals as input and output signals. Here, we describe the structure of this graph.

Usually, graphs have only one edge between two vertices. In actual audio graphs, two nodes can be linked together with several edges between the nodes, for instance a node that would expose two outputs, one for each stereo channel, to a mixer node, as in Figure 3.1. A graph with multiple edges between two nodes is called a *multigraph*. We explicitly add the connection points, called *ports*, to nodes, and the edges attach to ports of nodes. The edges are *directed*,

as the signal flows in only one direction, and so there are two categories of ports, *input ports* and *output ports*.

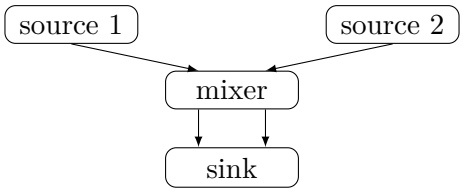


Figure 3.1.: Example of an audio graph, with two mono sources, a mixer, and a stereo sink. There are multiple edges between the mixer node and the sink node.

We adapt the port graph formalism, as in [AK08], to audio graphs. An *audio graph* is a triplet $G = (V, P, E)$ where:

- V is a set of vertices;
- $P = (P_i, P_o)$ is a pair of finite sets of ports where $P_i = \{\check{1}, \ldots, \check{n}\}$ is the set of input ports and $P_o = \{\hat{1}, \ldots, \hat{m}\}$ is the set of output ports;
- $E \subset ((V \times P_o) \times (V \times P_i))$ is a set of edges. An edge $e = ((v, p_o), (v', p_i))$ represents the data flowing between vertices v and v' and connects the *output* port p_o of vertice v to the *input* port p_i of vertice v'. We note $v.p_i$ port p_i of v and edge e as $v.p_o \to v'.p_i$. $v \rightsquigarrow v'$ will denote any edge $v.p \to v'.p'$ between v and v'.

If G is a graph, we denote by V_G its set of vertices and by E_G its set of edges. An edge $v \rightsquigarrow v$ is called a loop. For an edge $v \rightsquigarrow v'$, v is called the *source* and v' the *target*, and v and v' are *adjacent* or *neighbour* nodes. A *subgraph* G' of G is a graph whose node set, port set and edge set are subsets of those of G. A *path* π of G is a sequence $v_1, \ldots, v_n$ such that, for all $i \in \{1, \ldots, n-1\}, v_i \rightsquigarrow v_{i+1}$. We will note $\pi = v_1 \rightsquigarrow \cdots \rightsquigarrow v_n$. The number of incoming edges in v is the *in degree* denoted $i(v)$ of v, and the number of outgoing edges in v is the *out degree* $o(v)$ of v. Also, we enforce that given a node v, an input port p_i and output port p_o of v, $p_i \in \{\check{1}, \ldots, \check{i(v)}\}$ and $p_o \in \{\hat{1}, \ldots, \hat{o(v)}\}$.

The *degree* of v is the number of incoming and outgoing edges of v. If K is the set of shortest paths between any vertices v and v' of G, the *diameter* d_G of G is the longest of those paths in K.

Distinguished nodes. The nodes without input ports are called *sources*. They are typically audio stream generators. The nodes without output ports are the

sinks. They are audio sinks and are actually inputs to the soundcard, for instance. Nodes that are neither *sources* nor *sinks* are called *effects*.

Acyclic Audio graph

A graph G is *cyclic* if there is a node $v \in V$ and a path $v \rightsquigarrow \cdots \rightsquigarrow v$ in G. A graph is *acyclic* if it is not cyclic. Although some audio processing algorithms require feedback, they actually implement it with a delay, which is said to *break* the cycle in the graph. For instance for a one-pole filter, which adds a weighted output signal to the current input signal, we replace the feedback by a memory with one node that stores the output and another node that retrieves it after some delay, as shown in Figure 3.2. For that reason, we will assume that all audio graphs are *acyclic* in the following pages.

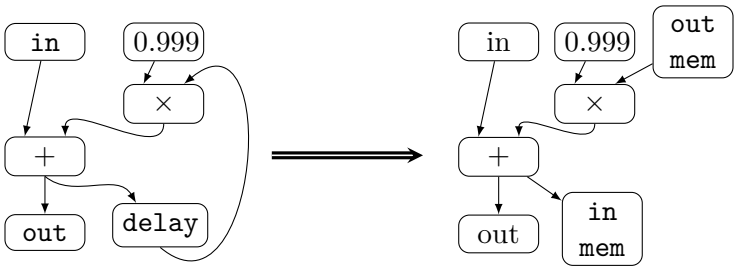


Figure 3.2.: A one-pole filter that exhibits feedback: the output of + goes back into an input of + with a delay. On the right, we implement this delay, by adding two ad-hoc nodes: `in mem` stores its input, and `out mem` outputs what was stored by `in mem`.

3.2. The domain of discrete streams

Digital audio processing systems do not compute on the continuous signal but on a discretization of it.[1] The dataflow paradigm [LM87] (see Section 2.4.1) is well suited to describe digital signal processing graphs, where tokens are typically audio samples or control parameters and vertices are audio processing nodes. A drawback of the standard dataflow paradigm is that it does not take time into account explicitly. In the following section, we will address this drawback and add time to this model.

[1] See Section 7.2.1 for a compact description.

A *sample* is the value of a signal at a given date. We associate a *timestamp* to a sample, and this timestamp can represent different moments, as shown in Figure 3.3:

- production or acquisition, when the signal is the result of a synthesizer or is recorded with a microphone for instance;
- processing, when the signal exits a processing node;[2]
- delivery, when the signal is output to the soundcard to be played by the speakers.

All three different times could be assigned to one sample by an audio processing node. The first one makes it possible to process synchronously two signals entering a node, the second one corresponds to the delay introduced by the processing latency in the node, and the third one is the deadline of the whole audio graph, before which the signal has to be processed.

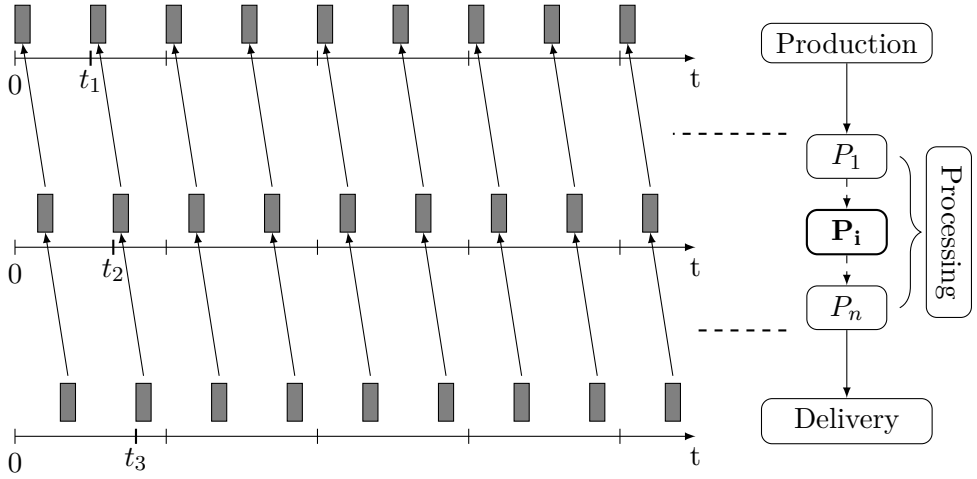


Figure 3.3.: A timestamp can represent various dates in the lifetime of a sample: production or acquisition, processing or delivery. Here, the second sample, after being processed by P_i, can be associated timestamps t_1 (production, in a source), t_2 (processing, output of P_i) or t_3 (deadline, for a sink).

Here we will consider only the first two kinds of timestamps. As in the *logical execution time* paradigm [KS12], we assume that the delivery or output date

[2]The time it enters a processing node can be deduced from the time it exits the connected input nodes, assuming instantaneous communications.

of the sample happens no later than the timestamp of production of the next sample in the sequence. Therefore, we do not use a timestamp that would represent the deadline for the whole graph. It means that we do not track the processing time of a node here. We will deal with the deadline of the audio graph in Chapter 8.

Audio processing nodes also actually process the samples grouped together in *buffers*. Audio samples are grouped in buffers because buffers can be processed more quickly by the processor; but grouping also increases the *buffering latency* as samples in the buffer are output at the date of the last sample. The *buffering latency* corresponds to a production latency as inputs are first delivered as buffers. Note that it is different from the processing latency, which we do not consider here. There are also other production latencies such as the latency caused by analog to digital converters, for instance.

Often in IMSs, *control* is aperiodic and will be represented by a timestamped sequence of control samples, as in Figure 3.4. *Audio* is represented by a timestamped sequence of buffers, as in Figure 3.5, which can be canonically represented by a timestamped sequence of samples, as shown in Figure 3.6. A periodic timestamped sequence of buffers can also be used to model periodic control such as control from a *low-frequency oscillator*. However in general, the sequence of buffer timestamps is not necessarily periodic. We also differentiate between *buffers of samples*, where we can attach a date to every sample, given the timestamp of the beginning of the buffer and the sample rate, and *buffers of data*, where each individual piece of data is not temporally localized in the buffer but is dated by the timestamp of the buffer. In the following, a buffer refers only to a *buffer of samples*, and an aggregate of data will be called an *array* and considered as one multidimensional sample. To find out the best tradeoff between the sample latency and the efficiency of the audio processing, we can choose to process longer buffers or smaller buffers, by splitting or fusing buffers in the ideal periodic buffer sequence, as in Figure 3.7.

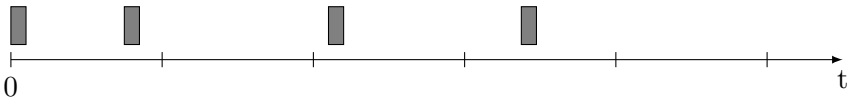


Figure 3.4.: An aperiodic timestamped sequence, used to model aperiodic *control*.

We want to model how samples are grouped together when processed by a signal processing node. We will give types (see Chapter 4) that describe statically how sequences of samples can be grouped together and if they are periodic, or not, how periodic, and show a semantics (see Section 5.2) of how an audio

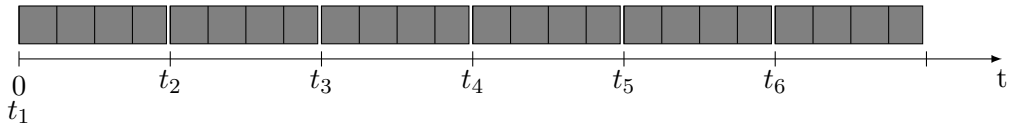


Figure 3.5.: Periodic timestamped sequence of 4-sample buffers.

Figure 3.6.: Canonical periodic timestamped sequence associated to the periodic timestamped sequence of 4-sample buffers of Figure 3.5

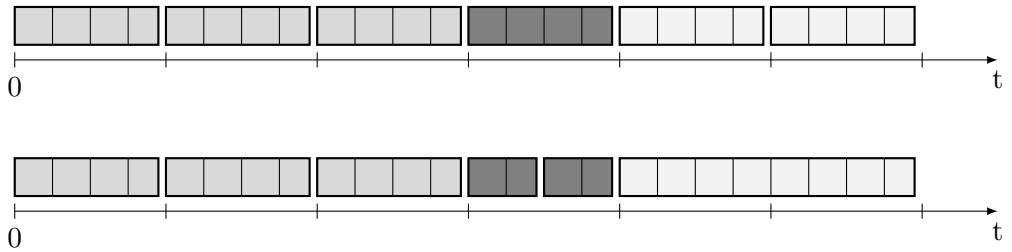


Figure 3.7.: A stream of timestamped 4-sample buffers at the top. At the bottom, the buffers of the same stream have been split (buffer 4) or fused (buffer 5 and 6).

graph computes such streams. In comparison to the usual execution model of an audio graph, we want to handle multirate and dynamic restructuring of buffering and to compute the latency introduced by the buffering.

Domains

We define here the domain [GS90] of streams. Defining a domain allows the definition of stream functions by induction without burden using standard tools.

We note $\mathcal{U}$ the set of possible samples. The set of timestamps $\mathcal{T}$ is countable, with $\mathcal{T} \subset \mathbb{Q}^+$. It is ordered with the canonical numerical order $<$ on $\mathbb{Q}^+$. We consider $\mathcal{T}$ as a flat domain $\mathcal{T}_\perp$ where $\perp$ is the minimal element [Mos90] : all elements are incomparable except $\perp$ for the domain order $\prec$. We also assume the following property on $\mathcal{T}$:

$$\exists \epsilon \in \mathbb{Q}^+, \epsilon > 0, \forall t_1, t_2 \in \mathcal{T}, t_1 \neq t_2 \Rightarrow \epsilon < |t_1 - t_2|$$

This property ensures that time advances, *i.e.* does not get stuck (non density).

A *buffer* $\boldsymbol{b}$ is a triplet (μ, p, b) of $B = \mathcal{L} \times \mathcal{P} \times \mathcal{U}^\star$ where $\mu \in \mathcal{L}\mathbb{Q}$ a latency, $p \in \mathcal{P}$ is the *sample-period*, with $\mathcal{P} = \mathbb{Q}^+ \setminus \{0\}$, and b is a finite sequence $b_1, \ldots, b_n$ of $\mathcal{U}$, which will be called a *buffer of samples*. We note $\ell(\boldsymbol{b})$ the *size* of buffer b , $\mu_{\boldsymbol{b}}$ the latency of buffer $\boldsymbol{b}$, and $p_{\boldsymbol{b}}$ the sample-period of buffer $\boldsymbol{b}$. We will also note $b[i]$ the i-th element of buffer of samples b. We also define an infix operator $\oplus : B \times \mathcal{L} \to B$ such that $(\mu_1, p, b) \oplus \mu_2 = (\mu_1 + \mu_2, p, b)$.

A timestamped sequence of buffers, or *stream* of buffers, is a function $s \in S$ with $S = \mathcal{T} \to B$. It associates to a timestamp a buffer of samples with latency and sample-period. A stream of buffers can also be seen as a sequence indexed by elements in $\mathcal{T}$ ordered by the canonical numerical total order on $\mathbb{Q}^+$, giving a meaning to *prefix*, *suffix*, *first* and *last* elements.

For a stream s, if $\mathrm{dom}(s)$ is finite, we say that s is finite. Similarly, if $\mathrm{dom}(s)$ is infinite, stream s is said to be infinite. We note ϵ the empty stream (*i.e.* $\mathrm{dom}(\epsilon) = \emptyset$).

We can represent a function s in S as a subset of $\mathrm{dom}(s) \times \mathrm{codom}(s) \subset (\mathcal{T} \times B)^\star \cup (\mathcal{T} \times B)^\omega$. To ensure the causality of the operations on streams defined in that way, we use the *prefix order* on $(\mathcal{T} \times B)^\star \cup (\mathcal{T} \times B)^\omega$, instead of the usual Scott order. The idea is that with the passing of time, we gain information and the known prefix (*i.e.* initial non-$\perp$ element) increases.

Definition 1 (first)**.** *Given $s \in S \setminus \epsilon$, we define* $\mathrm{first}(s)$ *as:*

$$\mathrm{first}(s) = \min\left(\mathrm{dom}(s)\right)$$

We also note $\text{last}(s) = \max(\text{dom}(s))$ if s is finite. Note that last is undefined for an infinite stream. For an ordered set E, we note $E^< = E \setminus \{\max(E)\}$ if E is finite and $E^< = E$ if E is infinite. For a set A, we note $\bar{A} = A \cup \{+\infty\}$.

Definition 2 (next)**.** *Given $s \in S$ and $t \in \text{dom}(s)^<$, we define* $\text{next}(s,t)$ *as:*

$$\text{next}(s,t) = \min\left\{t' \in \text{dom}(s) \,\middle|\, t' > t\right\}$$

Definition 3 (tail)**.** *Given $s \in S$, we define* $\text{tail}(s)$ *as:*

$$\text{tail}(s) = \begin{cases} \text{dom}(s) \setminus \{\text{first}(s)\} \text{ if } \text{dom}(s) \neq \emptyset \\ \emptyset \text{ if } \text{dom}(s) = \emptyset \end{cases}$$

We note $\ell(s) = \text{card}\,(\text{dom}(s)) \in \bar{\mathbb{N}}$ the length of stream s, potentially infinite.

We also define two operators prec and succ returning a subset of $\text{dom}(s)$ with, for $s \in S$ and $t \in \text{dom}(s)$:

$$\text{prec}(s,t) = \left\{t' \in \text{dom}(s) \,\middle|\, t' \leq t\right\} \tag{3.1}$$

$$\text{succ}(s,t) = \left\{t' \in \text{dom}(s) \,\middle|\, t' \geq t\right\} \tag{3.2}$$

Definition 4 (interleave)**.** *Given two streams s_1 and s_2, we define $s = \text{interleave}(s_1, s_2)$ as follows:*

$$\text{dom}(s) = \text{dom}(s_1) \cup \text{dom}(s_2)$$

$$\forall t \in \text{dom}(s),\ s(t) = \begin{cases} s_1(t), \text{ if } t \in \text{dom}(s_1) \\ s_2(t), \text{ if } t \notin \text{dom}(s_1) \end{cases}$$

Operator interleave yields a stream s resulting from the temporal asymmetric interleaving of two streams s_1 and s_2. If there are samples at the same timestamp in each stream, as s must be a function, we need to associate to it only one value. We could parametrize interleave with a function that would combine the value of s_1 and s_2 at that timestamp, as it is done in ReactiveML [MP05] with the `gather` construct. We choose here the value of s_1 at that timestamp for the sake of simplicity.

Definition 5 (concat)**.** *We define* concat *from* interleave *with an additional condition on the timestamps of the last and first bufers in the two streams to concatenate. Given $s, s' \in S \setminus \epsilon$ with* $\text{last}(s) < \text{first}(s')$*, we define* concat $=$ interleave*. We will also denote* $\text{concat}(s, s')$ *as $s \odot s'$.*

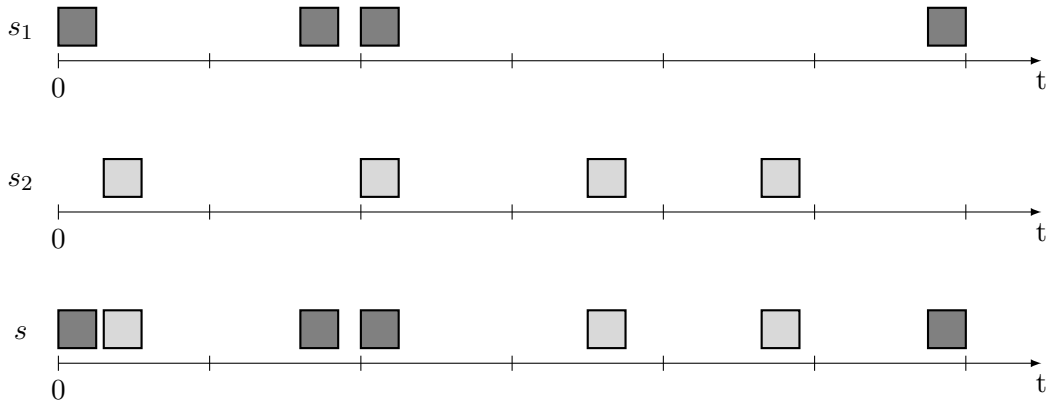


Figure 3.8.: Operator interleave on two aperiodic 1-sample buffer streams s_1 and s_2. Samples in s are greyed in accordance to the stream they take their value from. Remark that s_1 and s_2 share a timestamp, and that we keep in s the value of s_1 at that timestamp.

Notation of s as a sequence. We have a stream $s \in S \setminus \epsilon$; $\mathrm{dom}(s)$ as an ordered set can be written $\{t_1, t_2, \dots\}$ such that $s = ((t_1, \boldsymbol{b}_1), (t_2, \boldsymbol{b}_2), \dots)$. We denote $t_1 = \mathrm{first}(s)$. Given $i \in \{1, \dots, \ell(s)\}$, if t_i is an element of $\mathrm{dom}(s)^<$, then $t_{i+1} = \mathrm{next}(s, t_i)$ and more generally, for $1 \leq k \leq \ell(s) - i, t_{i+k} = \mathrm{next}(s, \cdot)^k(t_i)$. We note $I_s = \{1, \dots, \ell(s)\}$, which we call set of indices of s. We similarly note for $i \in I_s$, $\boldsymbol{b}_i = s(t_i)$ and $s_i = (t_i, \boldsymbol{b}_i)$ and finally, $s = ((t_i, \boldsymbol{b}_i))_{i \in I_s}$.

If all buffers in the stream have size 1, we call it a *stream of samples*. We note $A = \{s \in S \,|\, \forall t \in \mathrm{dom}(s), \ell(s(t)) = 1\}$ the set of *streams of samples*. If all buffers of a stream have all the same size, we call the stream a *homogeneous stream*. If the stream has only one buffer, we call it a *singleton stream*.

Definition 6 (Substream)**.** *Let $s \in S$ a stream. $s' \in S$ is the substream of s starting at $t \in \mathcal{T}$, and ending $t' \in \bar{\mathcal{T}}$ such that:*

$$\mathrm{dom}(s') = \{t'' \in \mathrm{dom}(s) \,|\, t \leq t \leq t'\}$$
$$\forall t'' \in \mathrm{dom}(s'),\ s'(t'') = s(t'')$$

We will note $s' = \mathrm{substream}(s, t, t')$.

Definition 7 (Canonical flattening of a stream of buffers into a stream of samples)**.** *Any stream can be converted into a stream of samples, using the conversion function $\phi : S \to A$.*

Let $s = ((t_i, \boldsymbol{b}_i))_{I_s}$ be a stream. $s' = \phi(s)$ is defined by $s' = ((t'_i, \boldsymbol{b}'_i))_{i \in I_{s'}}$

where:

$$\ell(s') = \sum_{i=1}^{\ell(s)} \ell(\boldsymbol{b}_i)$$

and

$$\forall i \in I_s, \forall j \in \{1, \dots, \ell(\boldsymbol{b}_i)\}, \begin{cases} t'_{j+\sum_{k=1}^{i-1} \ell(\boldsymbol{b}_k)} = t_i + p_{\boldsymbol{b}_i} \times (j-1) \\ b'_{j+\sum_{k=1}^{i-1} \ell(\boldsymbol{b}_k)}[1] = b_i[j] \\ p_{\boldsymbol{b}'_{j+\sum_{k=1}^{i-1} \ell(\boldsymbol{b}_k)}} = p_{\boldsymbol{b}_i} \\ \mu_{\boldsymbol{b}'_{j+\sum_{k=1}^{i-1} \ell(\boldsymbol{b}_k)}} = \mu_{\boldsymbol{b}_i} \end{cases} \tag{3.3}$$

Definition 8 (Equivalence between streams)**.** *Given two streams s and s', we say they are equivalent if $\phi(s) = \phi(s')$. We will note it $s \equiv s'$.*

Definition 9 (Buffer-periodic stream of buffers)**.** *A stream $s \in S$ is* buffer-periodic *if and only if:*

$$\exists \pi_s \in \mathcal{P}, \forall i \in I_{\mathrm{dom}(s)<}, t_{i+1} - t_i = \pi_s$$

We call this π_s the stream-period*.*

Note that the *sample-period* p_b of a buffer b, and the *buffer-period* π_s of a stream s are usually not the same periods. For instance, a typical audio stream will have a sample-period of 1/44100 s, whereas the buffer period will usually be between $\frac{32}{44100}$ s and $\frac{1024}{44100}$ s.

We give a simpler expression of the canonical representation $\phi(s)$ of a stream of buffers $s \in S$ when s is *homogeneous* with buffers of size n. With $s = ((t_i, \boldsymbol{b}_i))_{i \in I_s}$ and $s' = ((t'_i, \boldsymbol{b}'_i))_{i \in I_{s'}}$ where $s' = \phi(s)$ and we have:

$$\forall i \in I_{s'}, \begin{cases} t'_i = t_{1+\left\lfloor \frac{i-1}{n} \right\rfloor} + ((i-1) \bmod n) \times p_{\boldsymbol{b}_{1+\left\lfloor \frac{i-1}{n} \right\rfloor}} \\ b'_i[1] = b_{1+\left\lfloor \frac{i-1}{n} \right\rfloor}[1 + (i-1) \bmod n] \\ p_{\boldsymbol{b}'_i} = p_{\boldsymbol{b}_{1+\left\lfloor \frac{i-1}{n} \right\rfloor}} \\ \mu_{\boldsymbol{b}'_i} = \mu_{\boldsymbol{b}_{1+\left\lfloor \frac{i-1}{n} \right\rfloor}} \end{cases} \tag{3.4}$$

and $\ell(s') = n \times \ell(s)$.

Proof. We prove here that the equation 3.4 in the case of a stream of buffers of the same size is equivalent to the Definition 7.

Let s a homogeneous stream with buffers of same size n and $s' = \phi(s)$.

1. From Equation 3.4 to Equation 3.3 of Definition 7.

   Let $i \in I_s$ and $j \in \{1, \dots, n\}$.

   $$\begin{aligned}
   t'_{j+\sum_{k=1}^{i-1} \ell(\boldsymbol{b}_k)} &= t'_{j+n\times(i-1)} \text{ as all buffers } \boldsymbol{b}_k \text{ have the same size} \\
   &= t_{1+\left\lfloor \frac{j+n\times(i-1)}{n} \right\rfloor} + ((j+n\times(i-1)-1) \bmod n) \times p_{\boldsymbol{b}_{1+\left\lfloor \frac{j+n\times(i-1)}{n} \right\rfloor}} \\
   &\quad \text{using Equation 3.4} \\
   &= t_{1+\left\lfloor \frac{j-1}{n}+i-1 \right\rfloor} + ((j+n\times(i-1)-1) \bmod n) \times p_{\boldsymbol{b}_{1+\left\lfloor \frac{j-1}{n}+i-1 \right\rfloor}} \\
   &= t_i + ((j-1) \bmod n) \times p_{\boldsymbol{b}_i} \\
   &= t_i + (j-1) \times p_{\boldsymbol{b}_i} \text{ as } j \in \{1, \dots, n\}
   \end{aligned}$$

   Similarly, we have, using the same arguments on indices:

   $$\begin{aligned}
   b'_{j+\sum_{k=1}^{i-1} \ell(\boldsymbol{b}_k)}[1] &= b_i[j] \\
   \ell(s') &= \sum_{i=1}^{n_s} \ell(\boldsymbol{b}_i) \\
   p_{\boldsymbol{b}'_i} &= p_{\boldsymbol{b}_{1+\left\lfloor \frac{i-1}{n} \right\rfloor}} \\
   \mu_{\boldsymbol{b}'_i} &= \mu_{\boldsymbol{b}_{1+\left\lfloor \frac{i-1}{n} \right\rfloor}}
   \end{aligned}$$

   So Definition 7 holds.

2. From Definition 7 to Equation 3.4. We have:

   $$\forall i \in I_s, \forall j \in \{1, \dots, \ell(\boldsymbol{b}_i)\}, \begin{cases} t'_{j+\sum_{k=1}^{i-1} \ell(\boldsymbol{b}_k)} = t_i + p_{\boldsymbol{b}_i} \times (j-1) \\ b'_{j+\sum_{k=1}^{i-1} \ell(\boldsymbol{b}_k)}[1] = b_i[j] \end{cases}$$

   All buffers in stream s have same size n, therefore, $\ell(s') = n \times \ell(s)$.

   Let $i \in I_{s'}$. We can write:

   $$t'_i = t'_{1+(i-1) \bmod n+n\left(1+\left\lfloor \frac{i-1}{n} \right\rfloor -1\right)}$$

   We set $j = 1 + (i-1) \bmod n$ and $i' = 1 + \left\lfloor \frac{i-1}{n} \right\rfloor$ to rewrite it as:

   $$t'_i = t'_{j+n\times(i'-1)}$$

We remark that for $x \in \mathbb{N}$ and $y \in \mathbb{N} \setminus \{0\}$, the following property holds:

$$\left\lfloor \frac{x}{y} \right\rfloor = \frac{x - x \bmod y}{y} \tag{3.5}$$

If we write the Euclidian division of x by y, we have $x = y \times q + r$ with $0 \leq r < y$ and $q \in \mathbb{N}$.

$$\begin{aligned} \frac{x - x \bmod y}{y} &= \frac{y \times q + r - r}{y} \\ &= q \end{aligned}$$

and

$$\begin{aligned} \left\lfloor \frac{x}{y} \right\rfloor &= \left\lfloor \frac{y \times q + r}{y} \right\rfloor \\ &= \left\lfloor q + \frac{r}{y} \right\rfloor \\ &= q \text{ as } q \in \mathbb{N} \text{ and } 0 \leq \frac{r}{y} < 1 \end{aligned}$$

It follows from Equation 3.5 that:

$$\begin{aligned} j + n \times (i' - 1) &= 1 + (i-1) \bmod n + n \times \left(1 + \left\lfloor \frac{i-1}{n} \right\rfloor - 1\right) \\ &= 1 + (i-1) \bmod n + n \times \left(1 + \frac{(i-1) - (i-1) \bmod n}{n} - 1\right) \\ &= 1 + (i-1) \bmod n + (i-1) - (i-1) \bmod n \\ &= i \end{aligned}$$

We also have:

$$\begin{aligned} i' = 1 + \left\lfloor \frac{i-1}{n} \right\rfloor &\leq 1 + \left\lfloor \frac{n\ell(s) - 1}{n} \right\rfloor \\ &= 1 + \frac{(n\ell(s) - 1) - (n\ell(s) - 1) \bmod n}{n} \text{ using Equation 3.5} \\ &= \ell(s) + \frac{n - 1 - (n\ell(s) - 1) \bmod n}{n} \\ &= \ell(s) \text{ as } n - 1 - (n(\ell(s) - 1) + n - 1) \bmod n = 0 \end{aligned}$$

Therefore $i' \in \{1, \ldots, \ell(s)\} = I_s$.

As all buffers have same size, we have:

$$n \times (i' - 1) = \sum_{k=1}^{i'-1} \ell(\boldsymbol{b}_k)$$

Hence, we can apply Definition 7 with $i' \in I_s$ and $j \in \{1, \ldots, \ell(\boldsymbol{b}_{i'})\}$ and we get:

$$\begin{aligned}
t'_{j+\sum_{k=1}^{i'-1} \ell(\boldsymbol{b}_k)} &= t_{i'} + p_{\boldsymbol{b}_{i'}} \times (j-1) \\
&= t_{1+\left\lfloor \frac{i-1}{n} \right\rfloor} + p_{\boldsymbol{b}_{1+\left\lfloor \frac{i-1}{n} \right\rfloor}} \times ((i-1) \bmod n) \\
b'_{j+\sum_{k=1}^{i'-1} \ell(\boldsymbol{b}_k)}[1] &= b_{i'}[1 + (i-1) \bmod n] \\
p_{\boldsymbol{b}'_{j+\sum_{k=1}^{i'-1} \ell(\boldsymbol{b}_k)}} &= p_{\boldsymbol{b}_{1+\left\lfloor \frac{i-1}{n} \right\rfloor}} \\
\mu_{\boldsymbol{b}'_{j+\sum_{k=1}^{i'-1} \ell(\boldsymbol{b}_k)}} &= \mu_{\boldsymbol{b}_{1+\left\lfloor \frac{i-1}{n} \right\rfloor}}
\end{aligned}$$

which is Equation 3.4.

□

Property 1 (Flattening of a buffer-periodic stream)**.** *Let s be an homogeneous buffer-periodic stream with period π_s, $s \neq \epsilon$, and $s' = \phi(s)$. We have:*

$$\forall i \in I_{s'}, \begin{cases} t'_i = t_1 + \pi_s \times \left\lfloor \frac{i-1}{n} \right\rfloor + ((i-1) \bmod n) \times p_{\boldsymbol{b}_{1+\left\lfloor \frac{i-1}{n} \right\rfloor}} \\ b'_i[1] = b_{1+\left\lfloor \frac{i-1}{n} \right\rfloor}[1 + (i-1) \bmod n] \end{cases} \tag{3.6}$$

Period and latency are as in Equation 3.4.

Definition 10 (Sample-periodic stream)**.** *A stream $s \in S$ is said* sample-periodic *if $\phi(s)$ is buffer-periodic.*

It implies that all buffers $\boldsymbol{b}$ in s have the same sample-period $p_{\boldsymbol{b}}$. However, a buffer-periodic stream is not necessarily sample-periodic. For instance, buffer-periodic streams with buffers of same sample-periods are not necessarily sample-periodic, as in the counter-example of Figure 3.9. The number of samples in the buffers could also change, as in a non-homogeneous stream. For instance, it can represent a stream which has been resampled at some time intervals.

Another example are streams filtered by a control condition (switch on/off): the absence of a buffer is not the same as a buffer of silent audio. In those cases though, subsequences of the stream are sample-periodic. A 1-buffer stream is sample-periodic, because of the periodicity of its only one buffer.

Audio streams are represented by sample-periodic buffered streams, where the sample period is typically 1/44100 s. Buffer sizes often range from 32 to 4096 samples, in powers of 2; therefore for a buffer size of 256, the buffer period is 256/44100 s.

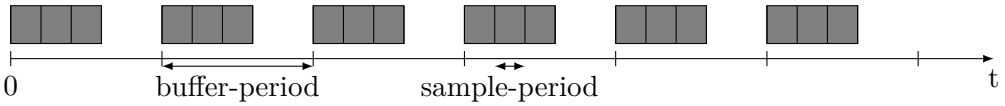


Figure 3.9.: A stream that is *buffer-periodic* and has all buffers with the same *sample-period*. However, the stream is not *sample-periodic*. The time interval between the last sample of a buffer and the first sample of the next buffer is not populated with samples with the same sample-period.

Property 2 (Sample-periodic buffer-periodic streams with same periods)**.** *Let $s \in S$ a buffer-periodic stream with buffer-period π_s where all buffers have the same sample-period p and the same size n,* i.e*, for all $i \in I_s, \ell(s_i) = \ell(s_1)$. Stream s is sample-periodic if and only if:*

$$\forall i \in I_{\mathrm{dom}(s)^<}, p_{s_i} = \frac{t_{i+1} - t_i}{\ell(s_i)}$$

i.e.

$$\pi_s = p \times n$$

Proof. Let $s \in S$ a buffer-periodic stream where all buffers have the same sample-period p and the same size n. We note $s' = \phi(s)$. As s is buffer-periodic with buffer-period π_s and buffers have the same size n, we can apply Equation 3.4. Let $i \in \{1, \ldots, n \times n_s - 1\}$. We have:

$$t'_i = t_{1+\left\lfloor \frac{i-1}{n} \right\rfloor} + ((i-1) \bmod n) \times p$$

Hence:

$$t'_{i+1} - t'_i = t_{1+\left\lfloor \frac{i}{n} \right\rfloor} + (i \bmod n) \times p - t_{1+\left\lfloor \frac{i-1}{n} \right\rfloor} - ((i-1) \bmod n) \times p \quad (3.7)$$

1. We assume that s is sample-periodic. Then s' is buffer-periodic and there exists a period π' such that:

$$\pi' = t_{1+\left\lfloor \frac{i}{n} \right\rfloor} + (i \bmod n) \times p - t_{1+\left\lfloor \frac{i-1}{n} \right\rfloor} - ((i-1) \bmod n) \times p$$

Let i be such that sample $b'_i[1]$ in s' is the last of a buffer of s, and the next sample $b'_{i+1}[1]$ is the first of the next buffer in s. The idea is that even if buffers have all the same sample-period, we need to also have the same time interval as the sample-period between the last sample of a buffer and the first sample of the buffer that follows it.

In that case, i is the last index of a buffer in s, which all have size n, so n divides i. Therefore, there exists $q \in \mathbb{N}$ such that $i = q \times n$ and we have:

$$\begin{aligned}\left\lfloor \frac{i-1}{n} \right\rfloor + 1 &= \frac{(q \times n - 1) - (q \times n - 1) \bmod n}{n} + 1 \text{ using Equation 3.5} \\ &= \frac{q \times n - 1 - (n-1) + n}{n} \\ &= q\end{aligned}$$

and $\left\lfloor \frac{i}{n} \right\rfloor = \left\lfloor \frac{q \times n}{n} \right\rfloor = q$.

Hence, we have $\left\lfloor \frac{i}{n} \right\rfloor = \left\lfloor \frac{i-1}{n} \right\rfloor + 1$. We also have $(i-1) \bmod n = n - 1$. It follows that:

$$\pi' = t_{1+\left\lfloor \frac{i-1}{n} \right\rfloor + 1} - t_{1+\left\lfloor \frac{i-1}{n} \right\rfloor} - (n-1) \times p \tag{3.8}$$

If we consider the case where the two samples are not at the boundaries of a buffer, n does not divide i, hence, we have $\left\lfloor \frac{i}{n} \right\rfloor = \left\lfloor \frac{i-1}{n} \right\rfloor$ and so we get:

$$\pi' = p \times (i \bmod n - (i-1) \bmod n) = p$$

It yields, by replacing π' by p in Equation 3.8:

$$p = t_{1+\left\lfloor \frac{i}{n} \right\rfloor + 1} - t_{1+\left\lfloor \frac{i-1}{n} \right\rfloor} - (n-1) \times p$$

so

$$p = \frac{t_{\left\lfloor \frac{i}{n} \right\rfloor + 2} - t_{1+\left\lfloor \frac{i-1}{n} \right\rfloor}}{n} \text{ } i.e. \text{ } p = \frac{\pi'}{n}$$

We can rewrite it as:

$$p_{b_j} = \frac{t_{j+1} - t_j}{n}$$

where $j = 1 + \left\lfloor \frac{i-1}{n} \right\rfloor$

2. Let us assume that the equation of Property 2 holds.

   Let $i \in \{1, \ldots, \ell(s) - 1\}$. We need to prove that $t'_{i+1} - t'_i$ is the same quantity for all i. Our hypothesis that all buffers have the same sample-period ensures that for an interval between two consecutive samples inside a buffer of s, $t'_{i+1} - t'_i = p$. We need to check that if $t'_{i+1} - t'_i = p$ when i is the index of the last sample of a buffer and $i + 1$ the index of the first sample of the next buffer. In that case, Equation 3.7 gives:

$$\begin{aligned} t'_{i+1} - t'_i &= t_{1+\left\lfloor \frac{i}{n} \right\rfloor} + (i \bmod n) \times p - t_{1+\left\lfloor \frac{i-1}{n} \right\rfloor} - ((i-1) \bmod n) \times p \\ &= t_{1+\left\lfloor \frac{i}{n} \right\rfloor+1} - t_{1+\left\lfloor \frac{i}{n} \right\rfloor} - (n-1) \times p \\ &\quad \text{using the same arguments as for Equation 3.8.} \end{aligned}$$

   The equation of Property 2 states, as all buffers have size n, at index $1 + \left\lfloor \frac{i-1}{n} \right\rfloor$, that:

$$p = \frac{t_{1+\left\lfloor \frac{i-1}{n} \right\rfloor+1} - t_{1+\left\lfloor \frac{i-1}{n} \right\rfloor}}{n}$$

   Therefore:

$$\begin{aligned} t'_{i+1} - t'_i &= t_{1+\left\lfloor \frac{i-1}{n} \right\rfloor+1} - t_{1+\left\lfloor \frac{i-1}{n} \right\rfloor} - (n-1) \times \frac{t_{1+\left\lfloor \frac{i-1}{n} \right\rfloor+1} - t_{1+\left\lfloor \frac{i-1}{n} \right\rfloor}}{n} \\ &= \frac{t_{1+\left\lfloor \frac{i-1}{n} \right\rfloor+1} - t_{1+\left\lfloor \frac{i-1}{n} \right\rfloor}}{n} \\ &= p \end{aligned}$$

   Hence, s' is buffer-periodic and therefore, s is sample-periodic.

□

We summarize the various sets in use, and show how they are linked to expressing control and audio signals of IMSs in Table 3.1.

We can also classify the streams as in Table 3.2. Some streams are called *atypical* and are not usually found in audio systems.

in S	in IMS
$\mathcal{U}$	a sample, a control event
$\mathcal{U}^\star$	a buffer of samples
$\mathcal{L}$	a latency
$\mathcal{P}$	a period
$\boldsymbol{b} \in B$	a buffer
aperiodic $s \in S$ with $\forall t \in \mathrm{dom}(s), \ell(s(t)) = 1$	control events
sample-periodic $s \in S$	an audio stream

Table 3.1.: Signal and control in IMS and representation in S

Buffer-periodicity	Sample-periodicity	Description
Buffer-Periodic	Homogeneous	Same size of all buffers
	Sample-periodic	Same size and period for all buffers and $\pi_s = \ell(s_i) \times p_{s_i}$
	Atypical	Previous conditions do not apply.
Buffer-aperiodic	Stream of samples	$\ell(s_i) = 1$
	Atypical	Previous conditions do not apply.

Table 3.2.: A classification of streams: s refers to a stream here.

4. Syntax and types

In this chapter, we present a syntax of audio graphs, where audio processing nodes are first declared, and then connected together, in Section 4.1. We show a type system on streams flowing from nodes to nodes on the edges of the audio graph in Section 4.2.

4.1. Syntax of nodes and audio graphs

An audio graph is defined first by defining its nodes, and then how the nodes are connected together.

We distinguish between *simple nodes*, which process their input streams buffer per buffer and output streams with the same timestamps as their inputs; and *special nodes*, which can output streams and take input streams with buffers with different sample-period, sizes, or production timestamps (for instance a delay).

An audio graph is specified by a list of nodes and by the connections between these nodes. Nodes are referred to with an identifier $\langle \text{id} \rangle$. The edges of the graph are identified by a variable that materializes the ports. A connection is the association of the input edges and output edges to a node. As mentioned before, ports of a node are explicitly ordered, which means that the input edges and output edges can be ordered.

ϵ represents the empty string and `\n` is a new line.

⟨port⟩ ::= ⟨node⟩.⟨portnumber⟩
⟨edge⟩ ::= ⟨port⟩→⟨port⟩
⟨edgelist⟩ ::= ⟨edge⟩,⟨edgelist⟩ | ⟨edge⟩
⟨signals⟩ ::= ((⟨edgelist⟩ | ϵ))
⟨decl⟩ ::= ⟨id⟩:=⟨node⟩
⟨equation⟩ ::= ⟨signals⟩=⟨id⟩⟨signals⟩
⟨statement⟩ ::= ⟨decl⟩ | ⟨equation⟩
⟨audiograph⟩ ::= ⟨statement⟩\n ⟨audiograph⟩ | ⟨statement⟩

Simple nodes

Here, we give the syntax of ⟨node⟩.

⟨node⟩ ::= `maps(f ; a, b ; a', b')`

where f is a function from samples tuples to samples tuples, not streams to streams, with a input controls and a' output controls, b audio inputs and b' audio outputs. When `v` is the node identifier ⟨id⟩, then $i(\mathtt{v}) = a + a'$ and $o(\mathtt{v}) = b + b'$. The signature of f is:

$$f : \mathcal{U}^a \times \mathcal{U}^b \to \mathcal{U}^{a'} \times \mathcal{U}^{b'}$$

We can also define simple nodes as such:

⟨node⟩ ::= `mapb(f ; a, b ; a', b')`

In that case, `v` will have a input controls and a' output controls, and b audio inputs and b' audio outputs. f is a function from buffers to buffers, such that:

$$f : \mathcal{U}^a \times B^b \to \mathcal{U}^{a'} \times B^{b'}$$

Sources and sinks

⟨node⟩ ::= `in(`m`)`
⟨node⟩ ::= `out(`m`)`

We distinguish sources and sinks from other nodes: `in`(m) defines a source with m outputs and `out`(n) defines a sink with n inputs.

Special nodes

Special nodes are used to explicity convert between two streams with different buffering and sampling characteristics. They can be seen as *coercion* operators.

We define a special node as follows:

$$\langle \text{node} \rangle \quad ::= \quad \texttt{specialnode(params)}$$

We give an overview of the special nodes considered in this work in Table 4.1.

Node	Description
`delay(n, b)`	Gives back its input with a delay of n samples with default value `b`
`window(n, m)`	Sliding window of size n samples, shifted by m samples
`fuse(n)`	Fuse n consecutive buffers of a sample-periodic stream into one large buffer
`split(n)`	Split buffers into n buffers of equal size
`expansion(n)`	Multiply the number of samples in a buffer by n and decrease the sample-period by a factor of n
`decimation(n)`	Only keep $\frac{1}{n}$ of the samples in a buffer and increase the sample-period by a factor of n
`periodicize(p, n)`	Transform an aperiodic stream into a homogeneous periodic stream with sample-period p and buffer size n by copying the data as needed to feed the output

Table 4.1.: Special nodes. All these operators have one input stream and one output stream.

These nodes are representative of the computations happening in audio graphs and we will describe their semantics in the next sections.

For the graph of Figure 3.2, we have the following node declarations, leading to the graph with identifiers of Figure 4.1.

$v_1 := \texttt{in}(1)$
$v_2 := \texttt{maps}(0.999; 0, 0; 1, 0)$
$v_3 := \texttt{maps}(\texttt{out mem}; 0, 1; 0, 1)$
$v_4 := \texttt{maps}(+; 0, 2; 0, 1)$
$v_5 := \texttt{maps}(\times; 1, 1; 0, 1)$
$v_6 := \texttt{out}(1)$
$v_7 := \texttt{maps}(\texttt{in mem}; 0, 1; 0, 1)$

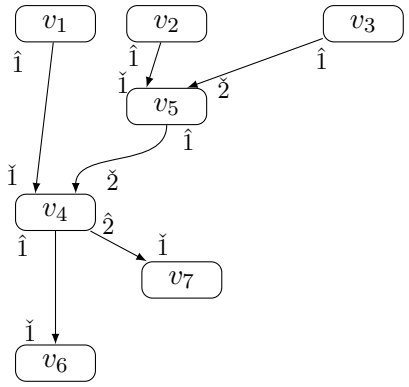


Figure 4.1.: The audio graph from the one pole filter of Figure 3.2 with ports and variable names on the edges.

Node `fork`. We also have a $\texttt{fork}(n)$ with one input and n outputs. It is used when an outgoing port of a node would be connected to n input ports of nodes. Instead, we connect this output to a `fork` node, the outputs of which are connected to the input ports of the outgoing nodes, as shown in Figure 4.2. This node simplifies the expression of the type system and the semantics by making sure an output port is only used once. It makes it possible to adapt the type of each occurrence of a stream depending on which input port an output port is connected to. It supports the idea of implementing an edge as an intermediate memory storage that can adapt to various buffering demands as described in Chapter 6.

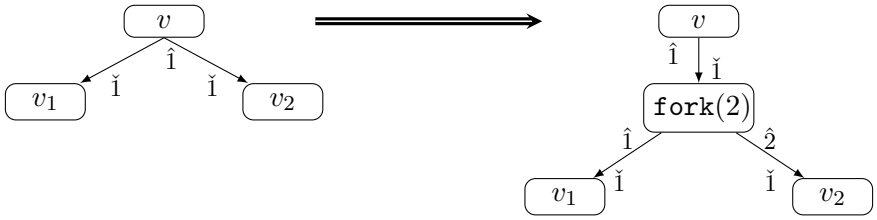


Figure 4.2.: Several edges go out of the same port on the left. In that case, we have to duplicate the output stream, by inserting a fork node, on the right.

Connecting nodes together

We write the execution of the whole graph in the *applicative style*, *i.e.* we write the nodes with their inputs and outputs as a set of unordered equations on variables labelling the inputs and outputs. We could have also used combinators (*functional style* [Del+87]) to describe the shape of the graph, but we thought that the additional positioning information given by the combinators is not useful here as we do not want to generate an actual circuit [Sch87].

An audio graph is written as a set of equations where variables represent connections between the nodes. Sources and sinks are also identifiable in the syntax by looking at the absence of input edges and output edges.

For instance, if we take the graph of Figure 4.1, we get the following equations:

$$
\begin{aligned}
(v_1.\hat{1} \to v_4.\check{1}) &= v_1() \\
(v_2.\hat{1} \to v_5.\check{1}) &= v_2() \\
(v_3.\hat{1} \to v_5.\check{2}) &= v_3() \\
(v_5.\hat{1} \to v_4.\check{2}) &= v_5(v_2.\hat{1} \to v_5.\check{1}, v_3.\hat{1} \to v_5.\check{2}) \\
(v_4.\hat{1} \to v_4.\check{1}, v_4.\hat{2} \to v_7.\check{1}) &= v_4(v_1.\hat{1} \to v_4.\check{1}, v_5.\hat{1} \to v_4.\check{2}) \\
() &= v_6(v_4.\hat{1} \to v_6.\check{1}) \\
() &= v_7(v_4.\hat{2} \to v_7.\check{1})
\end{aligned}
$$

4.2. Types

We associate types to streams, nodes and the whole audio graph. Types describe which nodes can be connected together and which kind of streams they can get as inputs and outputs. Some nodes can only accept some specific

streams, with a given buffer size or periodicity, while other nodes accept generic streams, *i.e.* any buffer size, any periodicity. The types of nodes are the usual types on a function where argument and return types are *streams types*. Thereafter, we will define the semantics only on well-typed audio graphs.

We use type variables $\alpha, \beta, \ldots$ to specify *parametric polymorphism* [PB02].

The graph must be syntactically correct and *well-formed*, *i.e.*, a new equation only involves input edges that have been output ports in the previous equations.

Types of elements in $\mathcal{U}$

Elements of $\mathcal{U}$ can represent control, audio samples, *i.e.* scalar elements, or multidimensional elements, such as images or spectral bins obtained from a spectral analysis.

The syntax of element types is defined by the following grammar:

$$\langle \text{element type} \rangle \quad ::= \quad \langle \mathit{SType} \rangle \mid \alpha$$

where SType (as *scalar type*) is handled as usual ML types, as in [Kah87], such as float, int, array. If we need several type variables for element types, we will note them $\alpha_1, \ldots, \alpha_n$.

For instance, an audio sample could have type real. A pixel in an image could have type $\mathit{pixel} = \mathit{int} \times \mathit{int} \times \mathit{int}$ where each integer is one of the RGB components, and a 64x64 square image, $\mathit{array}(\mathit{pixel}, 64, 64)$.

Type of a buffer

A buffer is similar to an array but its elements are called samples and may be associated to a timestamp (using ϕ, see Definition 7).

$$\langle \text{buf type} \rangle \quad ::= \quad \mathit{buffer}(p, n, \langle \text{element type} \rangle)$$

where $p \in \mathcal{P} \cup \{\gamma\}$, the period of buffers, and $n \in \mathbb{N} \setminus \{0\} \cup \{\delta\}$ the size of the buffer. δ and γ are type variables that indicate that a period or a size within respectively $\mathcal{P}$ and $\mathbb{N} \setminus \{0\}$ can be used for the considered buffer.

Type of a stream

The type of streams indicates the amount of information we have on the type of the samples, the buffer size, and the sample-period or the buffer-period of a

stream.

$$\begin{array}{rcll} \langle \text{period type} \rangle & ::= & \texttt{aperiodic}(\langle \text{element type} \rangle) & | \\ & & \texttt{periodic}(\pi, \langle \text{element type} \rangle) & | \\ & & \texttt{elastic}(p, n, \langle \text{element type} \rangle) & | \\ & & \texttt{buffered}(p, n, \langle \text{element type} \rangle) & | \\ & & \beta_{\texttt{stream}} & \end{array}$$

where $\pi \in \mathcal{P} \cup \{\gamma\}$ a buffer-period, $p \in \mathcal{P} \cup \{\gamma\}$, the sample-period of buffers, and $n \in \mathbb{N} \setminus \{0\} \cup \{\delta\}$ the size of buffers. γ and δ are type variables that indicate that any period or any size within respectively $\mathcal{P}$ and $\mathbb{N} \setminus \{0\}$ can be used for the considered stream.

Each of the four different types for a stream describes less and less precisely the sample-period, buffering and periodicity or not of a stream:

- Type `buffered`(p, n, element type) is the type of sample-periodic buffer-periodic streams, such as audio streams.
- Type `elastic`(p, n, element type) is the type of sample-periodic streams that are not buffer-periodic but are buffered in buffers of at most n samples with sample-period p. These maximum-size buffers can be split in smaller parts.
- Type `periodic`(π, element type) is the type of buffer-periodic streams that are not necessarily sample-periodic.
- Type `aperiodic`(element type) is the type of any stream, non necessarily periodic. It can be used for control streams.

Note that `buffered` and `elastic` types cannot represent a stream which would change of sample rate dynamically during the execution, because its sample-period p remains the same. Type `aperiodic` can be used but it hides the fact that the stream is actually *piecewise sample-periodic*, *i.e.*, given $s \in S$:

$$\begin{array}{l} \exists T \subset \text{dom}(s),\ \text{first}(s) \in T\ \ \wedge \forall t \in T, \exists p \in \mathcal{P}, \exists n \in \mathbb{N} \setminus \{0\}, \\ \quad (\text{substream}(s,t,t) : \texttt{elastic}(p,n,e) \\ \quad \vee \text{substream}(s,t,t) : \texttt{buffered}(p,n,e)) \end{array}$$

where e the common element type of s. Type `periodic` could also be used as the buffering, *i.e.* the buffer-period, would not change with a change in sample rate. Because these types refer to increasing information on their inhabitants, they exhibit a subtyping relationship (see Figure 4.3). These types also characterize the processing capabilities of a node. For example, some nodes are

general enough to handle buffers of any size, others are specific to one buffer size.

However, a change in sample rate is introduced through a `resampler` (or `expansion` or `decimation`) node in our framework. It means streams do not change of sample rates here but rather that we created a new graph which results from the insertion of a new node in the current graph. This new graph has another type, where a given stream has also the same sample rate permanently.

Types of functions for simple nodes

The functions used for `maps` have type ⟨sample function type⟩ and for `mapb`, ⟨buffer function type⟩:

$$\begin{aligned}
\langle\text{sample function type}\rangle &::= \langle\text{element type}\rangle\times\cdots\times\langle\text{element type}\rangle \rightarrow \\
&\quad \langle\text{element type}\rangle\times\cdots\times\langle\text{element type}\rangle \\
\langle\text{buffer function arg}\rangle &::= \langle\text{element type}\rangle \mid \langle\text{buf type}\rangle \\
\langle\text{buffer function}\rangle &::= \langle\text{buffer function arg}\rangle\times\cdots\times\langle\text{buffer function arg}\rangle \rightarrow \\
&\quad \langle\text{buffer function arg}\rangle\times\cdots\times\langle\text{buffer function arg}\rangle
\end{aligned}$$

Type of a node and of an audio graph

A node and an audio graph are seen as functions of streams into streams:

$$\begin{aligned}
\langle\text{node}\rangle &::= \langle\text{stream}\rangle\times\cdots\times\langle\text{stream}\rangle \rightarrow \langle\text{stream}\rangle\times\cdots\times\langle\text{stream}\rangle \\
\langle\text{source}\rangle &::= \rightarrow \langle\text{stream}\rangle\times\cdots\times\langle\text{stream}\rangle \\
\langle\text{sink}\rangle &::= \langle\text{stream}\rangle\times\cdots\times\langle\text{stream}\rangle \rightarrow
\end{aligned}$$

A graph is a function from sources to sinks, where we give the types of the input streams and output streams.

Typing rules

We note Γ a typing environment, which binds type τ to term s, denoted $s : \tau$ or $v : \tau$. $\emptyset$ is the empty context and $\Gamma, s : \tau$ is an augmented context where term s with type τ has been added. E is a set of equations on types. $\Gamma \vdash s : \tau, E$ is the relation between environment Γ, term s, type τ, and set of equations E.

Types of a stream. A buffered type is a subtype of an elastic type, as in Rule (subtyping 1) and an elastic type is a subtype of an aperiodic type, as in Rule (subtyping 2). A buffered type is also a subtype of a periodic type, as in

Rule (subtyping 3), which is a subtype of an aperiodic type, as in Rule subtyping 4. This leads to the subtyping hierarchy of Figure 4.3. We will write $A <: B$ to say that A is a subtype of B.

$$\frac{\Gamma \vdash s : \mathit{buffered}(p, n, \alpha), E}{\Gamma \vdash s : \mathit{elastic}(p, n, \alpha), E} \quad \text{(subtyping 1)}$$

$$\frac{\Gamma \vdash s : \mathit{elastic}(p, n, \alpha)}{\Gamma \vdash s : \mathit{aperiodic}(\alpha)} \quad \text{(subtyping 2)}$$

$$\frac{\Gamma \vdash s : \mathit{buffered}(p, n, \alpha)}{\Gamma \vdash s : \mathit{periodic}(p \times n, \alpha)} \quad \text{(subtyping 3)}$$

$$\frac{\Gamma \vdash s : \mathit{periodic}(\pi, \alpha)}{\Gamma \vdash s : \mathit{aperiodic}(\alpha)} \quad \text{(subtyping 4)}$$

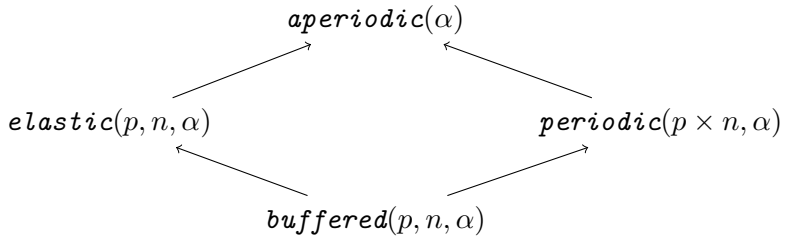


Figure 4.3.: Subtyping hierarchy for streams. A type can be *upgraded* to a type upper in the diagram. This generalization corresponds to losing some temporal and buffering information related to the stream.

Declaration This is similar to a let-binding construct typing.

$$\frac{\Gamma \vdash \texttt{v := node} : \tau_1, E \qquad \Gamma, \texttt{v} : \tau_1 \vdash \texttt{G} : \tau_2, F}{\Gamma \vdash \texttt{v := node} : \tau_1 \texttt{\textbackslash n G} : \tau_2, E \cup F} \quad \text{(decl)}$$

Below, we show the actual types of simple nodes and special nodes, which are functions from streams to streams.

Types of simple nodes. A simple mode built with maps can operate on any buffer size and periodicity of buffers, as in Rules (maps elastic), (maps aperiodic). We show the rule for two inputs and two outputs, each aperiodic and elastic, to ease the writing, but it extends naturally to more inputs and outputs.

$$\frac{\Gamma \vdash \texttt{f} : \alpha_1 \times \alpha_2 \to \alpha_3 \times \alpha_4}{\Gamma \vdash \texttt{maps}(f; 1, 1; 1, 1) : \textit{aperiodic}(\alpha_1) \times \textit{elastic}(\gamma_1, \delta_1, \alpha_2) \to \textit{aperiodic}(\alpha_3) \times \textit{elastic}(\gamma_1, \delta_1, \alpha_4)} \text{(maps elastic)}$$

In Rule (maps elastic), $\alpha_1, \alpha_2, \alpha_3, \alpha_4$ are sample types as f is a sample function with which we build a node with `maps`. Node `maps` has two inputs and two outputs. The first input and the first output of `maps` have respectively type $\textit{aperiodic}(\alpha_1)$ and type $\textit{aperiodic}(\alpha_2)$ and represent control, whereas the second input and output of `maps` have respectively type $\textit{elastic}(\gamma_1, \delta_1, \alpha_2)$ and $\textit{elastic}(\gamma_2, \delta_2, \alpha_4)$ and represent audio streams.

$$\frac{\Gamma \vdash \texttt{f} : \alpha_1 \times \alpha_2 \to \alpha_3 \times \alpha_4}{\Gamma \vdash \texttt{maps}(f; 1, 1; 1, 1) : \textit{aperiodic}(\alpha_1) \times \textit{aperiodic}(\alpha_2) \to \textit{aperiodic}(\alpha_3) \times \textit{aperiodic}(\alpha_4)} \text{(maps aperiodic)}$$

A node built with `mapb` is typed similarly as with `maps`. The difference lies in how audio signals are typed: if the function on buffers handles buffers with a period and size, then audio signals are typed as $\textit{buffered}$, in Rule (mapb buffered). They can also be typed as $\textit{periodic}$ as in Rule (mapb periodic) if the node handles buffer-periodic but not sample-periodic buffers, such as a FFT node that operates on overlapping windows of the signal.

$$\frac{\Gamma \vdash \texttt{f} : \alpha_1 \times \textit{buffer}(\gamma_1, \delta_1, \alpha_2) \to \alpha_3 \times \textit{buffer}(\gamma_2, \delta_2, \alpha_4)}{\Gamma \vdash \texttt{mapb}(f; 1, 1; 1, 1) : \textit{aperiodic}(\alpha_1) \times \textit{periodic}(\gamma_1 \times \delta_1, \alpha_2) \to \textit{aperiodic}(\alpha_3) \times \textit{periodic}(\gamma_2 \times \delta_2, \alpha_4)} \text{(mapb periodic)}$$

$$\frac{\Gamma \vdash \texttt{f} : \alpha_1, \textit{buffer}(\gamma_1, \delta_1, \alpha_2) \to \alpha_3, \textit{buffer}(\gamma_2, \delta_2, \alpha_4)}{\Gamma \vdash \texttt{mapb}(f; 1, 1; 1, 1) : \textit{aperiodic}(\alpha_1) \times \textit{buffered}(\gamma_1, \delta_1, \alpha_2) \to \textit{aperiodic}(\alpha_3) \times \textit{buffered}(\gamma_2, \delta_2, \alpha_4)} \text{(mapb buffered)}$$

Types of special nodes on a stream. We show the declarations of special nodes here. Some special nodes preserve sample-periodicity or create sample-periodic streams, and are typed with the $\textit{buffered}$ or $\textit{elastic}$ type.

The typing rules have the following general shape:

$$\frac{}{\Gamma, \texttt{v} : \tau_1, \ldots, \tau_n \to T_{\text{node}}(\tau_1, \ldots, \tau_n) \vdash \texttt{v} : \tau_1, \ldots, \tau_n \to T_{\text{node}}(\tau_1, \ldots, \tau_n), \emptyset} \text{(node)}$$

where T_{node} is a function from types to types that depends on the kind of considered node. We now show several possible T_{node}.

For instance, fuse(m) take m consecutive buffers and fuse them into one buffer and can only do that if consecutive buffers have the same sample-periodicity, as in (fuse).

$$T_{\texttt{fuse(m)}} : \begin{cases} \textit{elastic}(p, n, \alpha) \to \textit{elastic}(p, m \times n, \alpha) \\ \textit{buffered}(p, n, \alpha) \to \textit{buffered}(p, m \times n, \alpha) \end{cases} \tag{fuse}$$

Rule (split) refers to a node `split`(m) that split buffers in a stream into m buffers.

$$T_{\texttt{split(m)}} : \begin{cases} \textit{buffered}(p, n, \alpha) \to \textit{buffered}(p, \frac{n}{m}, \alpha) \text{ if } n \bmod m = 0 \\ \textit{elastic}(p, n, \alpha) \to \textit{elastic}(p, \frac{n}{m}, \alpha) \text{ if } n \bmod m = 0 \\ \textit{periodic}(\pi, \alpha) \to \textit{periodic}(\frac{\pi}{m}, \alpha) \\ \textit{aperiodic}(\alpha) \to \textit{aperiodic}(\alpha) \end{cases} \tag{split}$$

A `delay(k)` node delays by k samples and so has only a meaning for sample-periodic streams, *i.e.* with ***buffered*** and ***elastic*** types.

$$T_{\texttt{delay(k)}} : \begin{cases} \textit{buffered}(p, n, \alpha) \to \textit{buffered}(p, n, \alpha) \\ \textit{elastic}(p, n, \alpha) \to \textit{elastic}(p, n, \alpha) \end{cases} \tag{delay}$$

The `window(k, m)` node creates a sliding window of buffers of size k shifted by m which overlap, the overlap being of $k - m$ samples, from a sample-periodic stream. The result is not sample-periodic any more due to the overlap (see Property 2) but is buffer-periodic.

$$T_{\texttt{window(k, m)}} : \begin{cases} \textit{buffered}(p, n, \alpha) \to \textit{periodic}(p \times m, \alpha) \\ \textit{elastic}(p, \alpha) \to \textit{periodic}(p \times m, \alpha) \end{cases} \tag{window}$$

Nodes `expansion(k)` and `decimation(k)` are used to resample the stream and oversample or undersample it by $k > 0$. Period and buffer size change but the stream remains sample-periodic if it was beforehand. It also technically applies to buffer-periodic but not sample-periodic streams, and aperiodic streams, even those which would be composed of buffers of strictly positive size, but the signal processing meaning of it is less obvious.

$$T_{\texttt{expansion(k)}} : \begin{cases} \textit{buffered}(p, n, \alpha) \to \textit{buffered}(\frac{p}{k}, k \times n, \alpha) \\ \textit{elastic}(p, n, \alpha) \to \textit{elastic}(\frac{p}{k}, k \times n, \alpha) \\ \textit{periodic}(\pi, \alpha) \to \textit{periodic}(\pi, \alpha) \\ \textit{aperiodic}(\alpha) \to \textit{aperiodic}(\alpha) \end{cases} \quad \text{if } k \neq 0 \qquad \text{(expansion)}$$

For `decimation`, we give Rule (decimation).

$$T_{\texttt{decimation(k)}} : \begin{cases} \textit{buffered}(p, n, \alpha) \to \textit{buffered}(k \times p, \frac{n}{k}, \alpha) \\ \textit{elastic}(p, n, \alpha) \to \textit{elastic}(k \times p, \frac{n}{k}, \alpha) \\ \textit{periodic}(\pi, \alpha) \to \textit{periodic}(\pi, \alpha) \\ \textit{aperiodic}(\alpha) \to \textit{aperiodic}(\alpha) \end{cases} \qquad \text{(decimation)}$$

Node `periodicize(p, n)` takes an *aperiodic* stream and transforms it into a sample-periodic stream of *elastic* type with sample-period p and buffer size n. We can also use a combination of `fuse`, `split`, `expansion` and `decimation` can be used to obtain a sample-periodic stream of desired period and buffer size from a sample-periodic stream with given period and buffer size, sintead of using `periodicize`.

$$T_{\texttt{periodicize(p, n)}}(\textit{aperiodic}(\alpha)) = \textit{elastic}(p, n, \alpha) \qquad \text{(periodicize)}$$

Multiple rates. The typing rules illustrate that we handle multirate on two levels:

- *multirate* at the sample rate level. For that, we use resampler nodes, such as `expansion` and `decimation`
- *multirate* at the buffer rate level. Nodes `fuse` and `split` modify the buffer-period $p \times n$ at a constant sample-period p, changing only buffer size n.

Type of an audio graph. The type of an audio graph is determined by looking at the set of equations connecting the nodes together.

$$\frac{\Gamma \vdash x_1 : \mu_1, E_1 \quad \dots \quad \Gamma \vdash x'_1 : \mu'_1, E'_1 \quad \dots \quad \Gamma \vdash v : \tau_1 \times \cdots \to T_{\mathrm{v}}(\tau_1, \dots)}{\Gamma \vdash (x'_1, \dots) = v(x_1, \dots) : \mu_1 \cdots \to \mu'_1 \times \dots, E \cup \cdots \cup \{\mu_1 <: \tau_1\} \cup \cdots \cup E'_1 \cup \{\mu'_1 <: \tau'_1\} \cup \{\mu'_1, \cdots = T_{\mathrm{v}}(\mu_1, \dots)\}} \text{(equation)}$$

$$\frac{\Gamma \vdash e : \mu_1 \times \cdots \to \mu'_1 \times \dots, E \qquad \Gamma \vdash G : \tau}{\Gamma \vdash e\texttt{\textbackslash n}\ G : \{e\mu_1 \times \cdots \to \mu'_1 \times \dots\} \cup \tau, E \cup E'} \text{(audio graph)}$$

To type the audio graph, we want to type all of the edges of the graphs, which appear in the equation defining the connections of the graph. For that, we have used the Hindley algorithm, by building a set of equations on types. The equations are then solved using the unification algorithm of Robinson [DL06].

Example of a simple graph. We show the graph declaration and types for the audio graph of Figure 4.4.

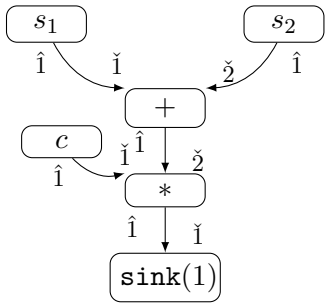


Figure 4.4.: A simple graph which mixes two sources and then applies a gain c to the result, before outputting it to a sink. For simple nodes, we just write the function used for `maps` as node label.

A textual syntax associated to the graph is the following:

```
s1 := source(1)
s2 := source(2)
c := source(1)

v1 := maps(+;0,2 ; 0,1)
v2 := maps(*; 1,1; 0,1)

s := sink(1)
```

The equations associated to the graph are:

$$
\begin{aligned}
e_{s1.\hat{\imath}\to v1.\check{\imath}} &= s1() \\
e_{s2.\hat{\imath}\to v1.\check{2}} &= s2() \\
e_{v1.\hat{\imath}\to v2.\check{2}} &= v1(e_{s1.\hat{\imath}\to v1.\check{\imath}}, e_{s2.\hat{\imath}\to v1.\check{2}}) \\
e_{c.\hat{\imath}\to v2.\check{\imath}} &= c() \\
e_{v2.\hat{\imath}\to s.\check{\imath}} &= v2(e_{c.\hat{\imath}\to v2.\check{\imath}}, e_{v1.\hat{\imath}\to v2.\check{2}}) \\
() &= s(e_{v2.\hat{\imath}\to s.\check{\imath}})
\end{aligned}
$$

5. Semantics

We present a denotational semantics that resolves to isochronous streams, *i.e.* control is *periodicized* and sent to the boundaries of buffers. We first give some operators that are used to transform streams, and then we present the semantics, based on the idea that nodes operate on *isosynchronous* streams, *i.e.* nodes output something when they are fed elements with the same timestamps.

5.1. Stream transformations

Several operators are used to transform stream timings but not the inner elements (in $\mathcal{U}$). Operator map (Definition 11) applies a buffer operator to each buffer in a stream. Operator fuse fuses contiguous buffers in a stream, as shown in Figure 5.1; split splits a buffer in a stream into several ones, as in Figure 5.2. Operator bufferize transforms a finite stream into a one-buffer stream, as in Figure 5.3. Operator snap snaps a stream to the timestamps of another stream, as in Figure 5.4. Operator periodicize transforms any stream into a sample-periodic stream, as shown in Figure 5.5. Operator window (see Figure 5.6) transforms a stream into a stream of potentially overlapping windows of the signal. Table 5.1 summarizes the operators with their domains and codomains.

Causality of operators Informally, an operator on s is causal if when it modifies the buffer at timestamp t, it only needs the buffers at timestamps $\text{prec}(s,t) \setminus \{t\}$. We will define only causal operators.

Definition 11 (map)**.** *Let $s \in S$ a stream and $\mathrm{o} : B \to B$. We define* $\text{map}(s, \mathtt{o})$ *as:*

$$\text{map}(s, \mathtt{o}) = ((t, \mathtt{o}(s(t))))_{t \in \text{dom}(s)}$$

If `o` *requires arguments* `args` *in addition to the buffer b to process, we note it in a curried way as* `o(args)`(b)*.*

We also define similarly a series of map(i) transformations that can apply an operator $\mathtt{o} : B^i \to B^j$ to i input buffers. The operator does not apply a

function to elements at the same index in the input sequences as in an usual map(i)[1] but two elements with the same timestamp. It also means that when some elements in an input stream have timestamps not present in another input stream, they are discarded.[2]

$$\text{map(i)}(s_1, \ldots, s_i, \mathsf{o}) = ((t, \mathsf{o}(s_1(t), \ldots, s_i(t))))_{t \in \bigcap_{k=1}^{i} \text{dom}(s_k)} \tag{5.1}$$

In addition, we update latencies in the following way. For each output stream $s'_{k'}$ with $k' \in \{1, \ldots, j\}$, we have:

$$\mu_{s'_{k'}}(t) = \max_{k \in \{1,\ldots,i\}} \{\mu_{s_k}(t)\}$$

We additionally define a flatmap operator for functions that generate streams, *i.e.* $\mathrm{o} : B \to S$, such that:

$$\text{flatmap}(s, \mathsf{o}) = \bigodot_{t \in \text{dom}(s)} \mathsf{o}(s(t)) \tag{5.2}$$

We also define mapsubstreams that applies a function to consecutive substreams of period p of the input stream. We write it here recursively, but we could write it as well by intension as the previous map operators.

$$\begin{aligned} \text{mapsubstreams}(s, \mathsf{o}, p) = [\text{fix}(\lambda f.\lambda s'.\mathsf{o}(\text{substream}(s', t_1, t_1 + p)) \\ \odot f(\text{substream}(s', \text{next}(s', t_1 + p), \text{last}(s'))))] \, (s) \end{aligned} \tag{5.3}$$

Definition 12 (fuse)**.** *We consider a* finite *sample-periodic stream* $s = ((t_i, \boldsymbol{b}_i))_{I_s} \in S$*. We define* fuse *with:*

$$\text{fuse}(s) = \begin{cases} \text{fuse}\left(\text{fuse2}\left(\text{first}(s), s(\text{first}(s)), s(\text{next}(s, \text{first}(s)))\right) \odot s_{\restriction \text{tail}(\text{tail}(s))}\right) & \text{if } \ell(s) \geq 2 \\ s \text{ if } \ell(s) < 2 \end{cases}$$

where $s_{\restriction \text{tail}(\text{tail}(s))}$ *is the restriction of* s *to* $\text{tail}(\text{tail}(s))$*, and where* $\text{fuse2}(t, \boldsymbol{b}_1, \boldsymbol{b}_2) = ((t', \boldsymbol{b}'))$*, such that:*

$$\begin{aligned} t' &= t \\ \boldsymbol{b}' &= (\mu_1 + \mu_2, p_s, b_1[1] \cdots b_1[\ell(b_1)] \cdot b_2[1] \cdots b_2[\ell(b_2)]) \end{aligned}$$

with p_s *the common sample-period,* μ_1, μ_2 *respectively the latency of* $\boldsymbol{b}_1, \boldsymbol{b}_2$*.*

Operator fuse *fuses all the consecutive buffers of a finite sample-periodic stream into a singleton stream, with only one larger buffer.*

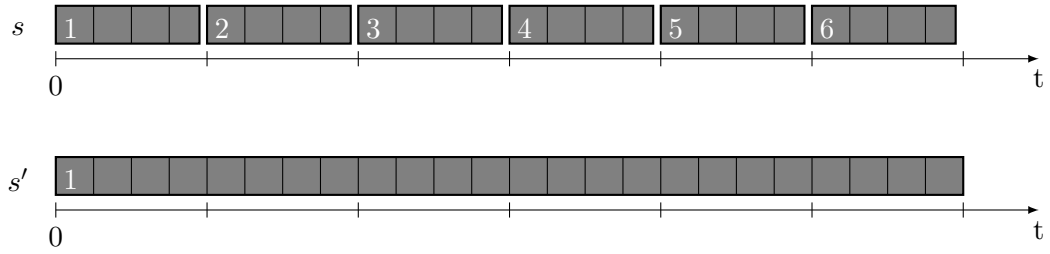


Figure 5.1.: The fuse operator: $s' = \text{fuse}(s)$

Operator fuse also applies on any finite sample-periodic substream of a non-sample-periodic stream, as shown on Figure 5.1.

Definition 13 (split)**.** *Given a singleton stream $s = ((t, \boldsymbol{b}))$, an integer $m \in \mathbb{N} \setminus \{0\}$, a stream $s' = ((t'_i, \boldsymbol{b}'_i))_{i\in\{1,2\}} = \text{split}(s, m)$ is obtained by:*

$$t'_1 = t_1 \text{ and } \boldsymbol{b}'_1 = (\mu_{\boldsymbol{b}} - p_{\boldsymbol{b}} \times (\ell(\boldsymbol{b}) - m), p_{\boldsymbol{b}_1}, b[1] \cdots b[m]) \tag{5.4}$$

$$t'_2 = t_1 + m \times p_{\boldsymbol{b}_1} \text{ and } \boldsymbol{b}'_2 = (\mu_{\boldsymbol{b}}, p_{\boldsymbol{b}}, b[m+1] \cdots b[\ell(\boldsymbol{b})] \tag{5.5}$$

Operator split splits the buffer in the stream into two consecutive buffers, as shown in Figure 5.2. The first one is defined by Equation 5.4 and the second one, by Equation 5.5.

We also define operator splite(s, m) that splits a one-buffer stream $s = ((t, b))$ into a m-buffer buffer-periodic sample-periodic stream, if $\ell(b) \bmod m = 0$, such that:

$$\text{splite}(s, m) = s_1 \odot \text{splite}(s_2, m - 1)$$

where $s_1 \odot s_2 = \text{split}(s, \frac{\ell(b)}{m})$

Extending split or splite to a stream s with $\ell(s) > 1$ is easily done with flatmap. For instance, $\lambda s.\text{flatmap}(s, \text{splite}(2))$ will split all buffers of s into two buffers of equal size and generate a stream with those new buffers.

Definition 14 (bufferize)**.** *We consider a finite stream of samples $s \in S$ (where all buffers have size 1), and a period $p \in \mathcal{P}$. We define $s' = \text{bufferize}(s, p)$*

[1] For instance in OCaml, `List.map2`.

[2] This is analogous to an usual map on input sequences with different sizes but in our case, absent timestamps can be anywhere in the streams, whereas for sequences, absent indices are at the end.

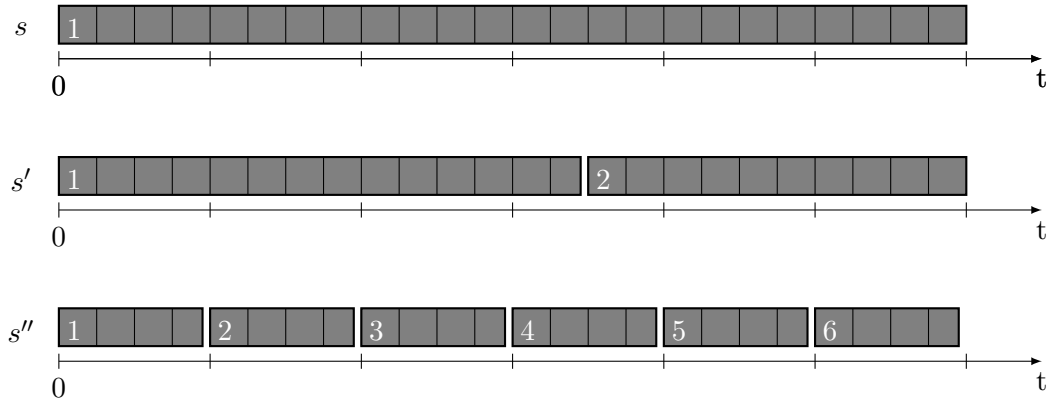


Figure 5.2.: The split operator on a 24-sample one-buffer stream. We show $s' = \text{split}(s, 14)$, which yields a two-buffer stream where the two buffers do not have the same size, and $s'' = \text{splite}(s, 4)$, which yields a 6-buffer stream with buffers of size 4.

with $s' = ((t', \boldsymbol{b}'))$ *a singleton stream, such that:*

$$\begin{aligned} t' &= \text{first}(s) \text{ and} \\ \ell(\boldsymbol{b}') &= 1 + \left\lfloor \frac{\text{last}(s) - \text{first}(s)}{p} \right\rfloor \\ \boldsymbol{b}' &= \left(\lceil \text{last}(s) - \text{first}(s) \rceil, p, (b'[1] \cdots b'[\ell(\boldsymbol{b}')])\right) \\ \forall i \in \{1, \ldots, \ell(\boldsymbol{b}')\},\ & b'[i] = b_{s(t'_i)}[1] \end{aligned}$$

where $t'_i = \max(\text{prec}(s, \text{first}(s) + (i-1) \times p))$.

Operator bufferize transforms a timestamped finite stream into a stream with only one buffer, as shown in Figure 5.3. It picks the sample whose timestamp is closest to the given multiple of a period in the past. It is causal: it is enough to know all the timestamps and elements of the stream before the current timestamp. It is usually applied on a substream. For a general stream $s \in S$ with arbitrary buffer sizes, we apply bufferize on $\phi(s)$. As the first sample has to wait for the last sample to be processed, latency added by bufferize is $(\ell(\boldsymbol{b}') - 1) \times p = \lceil \text{last}(s) - \text{first}(s) \rceil$. Typically, bufferize is used to periodically sample aperiodic control. We also use it to group into buffers a sample-periodic stream where samples are not grouped beforehand, which models the behaviour of a `source` for instance.

Definition 15 (snap)**.** *We consider a stream* $s = ((t_i, \boldsymbol{b}_i))_{i \in I_s}$*, a set* $T \subset \mathcal{T}$ *of*

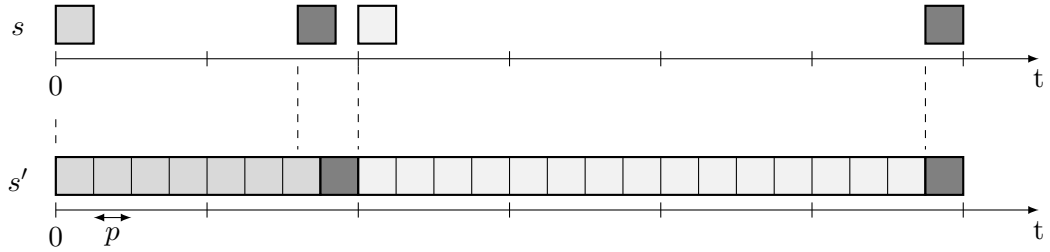


Figure 5.3.: The bufferize on an aperiodic 1-sample buffer stream. We want to transform it into a stream with one buffer of period p. Samples in s' are greyed in accordance to the sample in s they take their value from.

timestamps and an element $\boldsymbol{b} \in B$*. We define* $s' = \text{snap}(s, T, \boldsymbol{b})$ *by:*

$$\begin{aligned} \text{dom}(s') &= T \\ s' &= ((t'_i, \boldsymbol{b}'_i))_{i \in I_{s'}} \end{aligned}$$

with:

$$\forall t' \in \text{dom}(s'), s'(t') = \begin{cases} s(\max \text{prec}(s, t')) \oplus (t' - \max \text{prec}(s, t')) \,, & \textit{if } \text{prec}(s, t') \neq \emptyset \\ \boldsymbol{b}, & \textit{if } \text{prec}(s, t') = \emptyset \end{cases}$$

The snap operator[3] *binds* stream buffers to the timestamps of another stream. As shown in Figure 5.4, it chooses the value of the timestamp that is the closest in the past to the current timestamp of the target timestamps and as such, it is causal. If there is no such element, we use a default buffer $\boldsymbol{b} \in B$. We could write bufferize as the composition of snap to a periodic stream, followed by fusing the elements in the stream. Typically, snap can be used to snap control samples to the boundaries of audio buffers. The latency is equal to the difference between the new and the old timestamp.

We can also use a stream s'' as argument for snap instead of the set of target timestamps T. In that case, we take $T = \text{dom}(s'')$.

Property 3 (Timestamp preservation)**.** *Operators* substream*,* fuse *and* split *preserve the timestamps of the samples,* i.e.*, for a stream* $s \in S$ *and an operator* `o` *among those operators:*

$$\text{dom}(\phi(s)) = \text{dom}(\phi(\text{o}(s))$$

[3]The name of the operator is inspired by the *snap to grid* function available in some DAWs to quantize MIDI notes.

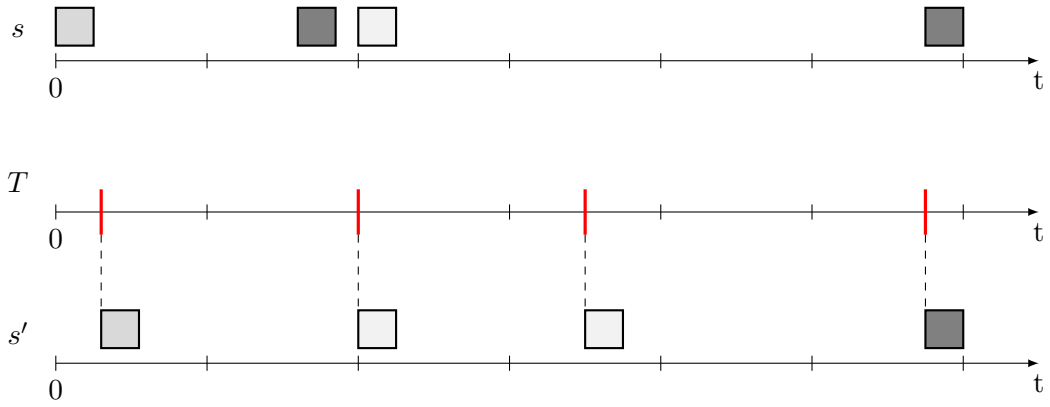


Figure 5.4.: The snap on an aperiodic 1-sample buffer stream. We want to use new timestamps for the samples. Samples in s' are greyed in accordance to the sample in s they take their value from. We do not need a default buffer here as the first timestamp of T is higher than the first timestamp of s.

Property 4 (Sample-periodicity preservation)**.** *Operators* substream, bufferize, fuse *and* split *preserve sample-periodicity.*

Definition 16 (periodicize)**.** *Let $s \in S$ a stream, $n \in \mathbb{N} \setminus \{0\}$ and $p \in \mathcal{P}$. We define $s' =$* periodicize(s, p, n) *by:*

$$\forall t' \in \mathrm{dom}(s'),\ s'(t') = \mathrm{bufferize}(\mathrm{substream}(\mathrm{snap}(\phi(s), T), t', t' + (n-1) \times p), p)$$

where

$$\begin{aligned} \mathrm{dom}(s') &= \{t \leq L(s) \,|\, \exists k \in \mathbb{N}, t = \mathrm{first}(s) + k \times n \times p\} \\ T &= \{t \leq L(s) \,|\, \exists k \in \mathbb{N}, t = \mathrm{first}(s) + k \times p\} \\ L(s) &= \begin{cases} +\infty, & \textit{if } s \textit{ is infinite} \\ \mathrm{last}(s) + p_{s(\mathrm{last}(s))} \times \ell(s(\mathrm{last}(s))), & \textit{if } s \textit{ is finite} \end{cases} \end{aligned}$$

Property 5 (periodicize sample-periodizes)**.** *Given $s \in S$, $n \in \mathbb{N} \setminus \{0\}$ and $p \in \mathcal{P}$,* periodicize(s, p, n) *is sample-periodic and* homogeneous*.*

Proof. Let $s \in S$ a stream, $n \in \mathbb{N}\setminus\{0\}$ and $p \in \mathcal{P}$. We note $s' =$ periodicize(s, p, n).

For each timestamp $t \in \mathrm{dom}(s')$, bufferize is applied using period p on a substream of duration $(n-1) \times p$ and so all the buffers in the stream have

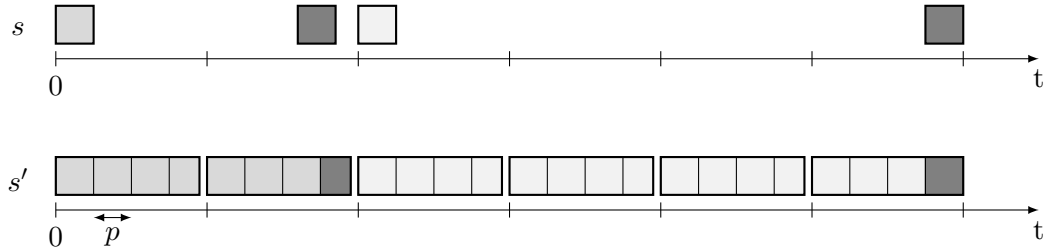


Figure 5.5.: periodicize(s, p, 4) operator on an aperiodic 1-sample buffer stream s. It transforms here s into a stream with 4-sample buffers of sample-period p. Samples in s' are greyed in accordance to the sample in s they take their value from.

the same size n and the same period p. s' is also clearly buffer-periodic per definition of $\mathrm{dom}(s')$.

Let $i \in I_{\mathrm{dom}(s)^<}$.

$$\frac{t_{i+1} - t_i}{\ell(s_i)} = \frac{(t_1 + (i+1) \times n \times p) - (t_1 + i \times n \times p)}{n} \text{ using the definition of } \mathrm{dom}(s')$$
$$= p$$

So we can apply Property 2 to s'. Therefore, s' is sample-periodic. □

Definition 17 (window)**.** *Let $s \in S$ a sample-periodic stream with sample-period p. Let $n \in \mathbb{N} \setminus \{0\}$ and $m \in \mathbb{N}$. We define $s' = \mathrm{window}(n, m)$, with $s'' = \phi(s)$, by:*

$$\mathrm{dom}(s') = \begin{cases} D \cap [0, \mathrm{last}(s)] & \textit{if } s \textit{ is finite} \\ D & \textit{if } s \textit{ is infinite} \end{cases}$$
$$\textit{where } D = \{\mathrm{first}(s) + m \times p \times k \mid k \in \mathbb{N}\}$$
$$\forall t' \in \mathrm{dom}(s'),\ s'(t') = \mathrm{fuse}(\mathrm{substream}(s'', t', t' + p \times (n-1)))(t')$$

A window of size n is shifted repeatedly by m samples to yield $\mathrm{window}(n, m)$, as shown in Figure 5.6. Note that s' is not sample-periodic, because of the overlap between buffers.

Resampling on a buffer

Buffer operators transform a buffer into a buffer. Buffer operators `decimation` and `expansion` are used to change the number of samples in a buffer, as shown

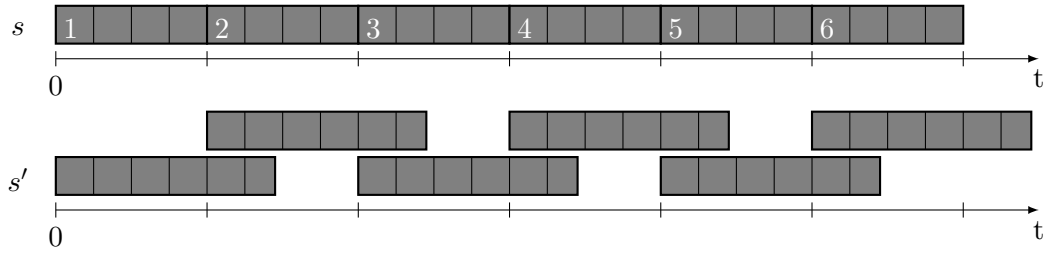


Figure 5.6.: The window operator: $s' = \text{window}(6, 4)(s)$

in Figures 5.7 and 5.8. We do not call them *downsampling* and *upsampling* as these operations usually require additional filtering that would be performed by an audio processing node (see Section 7.2).

Definition 18 (`decimation`)**.** *On a sample-periodic buffer $\boldsymbol{b} \in B$ with latency $\mu \in \mathcal{L}$, period $p \in \mathcal{P}$, given $m \in \mathbb{N} \setminus \{0\}$, we define $\boldsymbol{b}' = \texttt{decimation}(\boldsymbol{b}, m)$ with latency μ' and period p' such that:*

$$\mu' = \mu$$
$$p' = p \times m$$
$$\ell(\boldsymbol{b}') = \left\lfloor \frac{\ell(\boldsymbol{b})}{m} \right\rfloor$$
$$\forall i \in \{1, \ldots, \ell(\boldsymbol{b}')\},\ b'[i] = b[(i-1) \times m + 1]$$

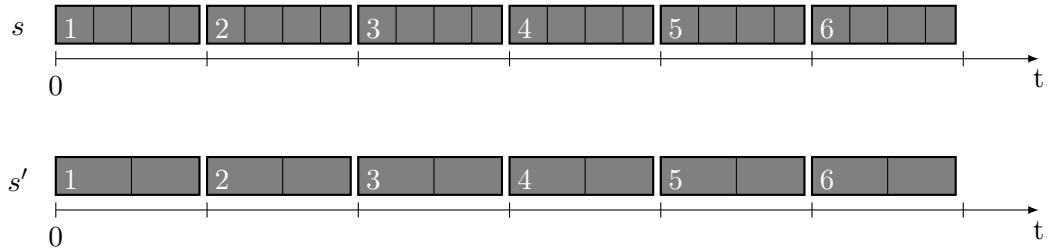


Figure 5.7.: The `decimation` operator, where $s' = \text{map}(s, \texttt{decimation}(2))$.

Definition 19 (expansion)**.** *On a sample-periodic buffer $b \in B$ with latency $\mu \in \mathcal{L}$, period $p \in \mathcal{P}$, given $m \in \mathbb{N} \setminus \{0\}$, we define $\boldsymbol{b}' = \texttt{expansion}(\boldsymbol{b}, m)$ with*

latency μ' and period p' such that:

$$\mu' = \mu$$
$$p' = \frac{p}{m}$$
$$\ell(\boldsymbol{b}') = \ell(\boldsymbol{b}) \times m$$
$$\forall i \in \{1, \ldots, \ell(\boldsymbol{b}')\},\ b'[i] = b[\left\lfloor \frac{i-1}{m} \right\rfloor + 1]$$

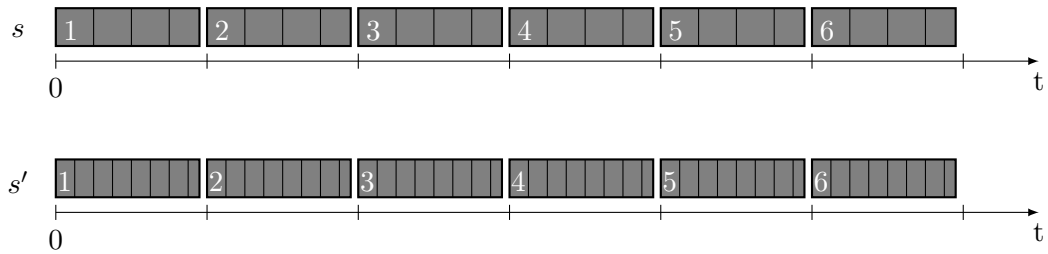


Figure 5.8.: The `expansion` operator, where $s' = \text{map}(s, \texttt{expansion}(2))$.

We also define operator `resampling`, such that for $\boldsymbol{b}$ a buffer and $r = \frac{n}{m} \in \mathbb{Q}^+ \backslash \{0\}, n, m \in \mathbb{N}^+ \backslash \{0\}$, $\texttt{resampling}(\boldsymbol{b}, r) = \texttt{decimation}(\texttt{expansion}(\boldsymbol{b}, n), m)$.

We usually want to apply resampling on a sample-periodic stream. Any stream s can be made sample-periodic by applying periodicize (see Property 5). Then, we just have to apply map with `resampling` as the buffer operation.

Property 6 (Snapping and resampling)**.** *Given $s \in S$ and $n \in \mathbb{N} \setminus \{0\}$, we have:*

$$\phi(\text{map}(s, \text{decimation}(n))) = \text{snap}(\phi(s), T)$$

with $T = \{t_{1+i} \,|\, i \equiv 0 \mod n\}$.

And we also have:

$$\phi(\text{map}(s, \text{expansion}(n))) = \text{snap}(\phi(s), T')$$

with $T' = \left\{ t_i + j \times \frac{t_{i+1} - t_i}{n} \,\middle|\, i \in I_s \wedge j \in \{0, \ldots, n-1\} \right\}$

Resampling buffers on a stream s and then applying ϕ is the same as choosing an adequate set of timestamps to which to snap timestamps of $\phi(s)$, *i.e.* snap is a more general version of $\text{map}(\cdot, \texttt{resampling}(\cdot, r))$ with $r \in \mathbb{Q}^+ \setminus \{0\}$.

Definition 20 (apply)**.** *Given a one-sample buffer* $\boldsymbol{e} \in B$ *and a buffer* $\boldsymbol{b} \in B$*, and a function on samples* $f : \mathcal{U}^2 \to \mathcal{U}$*, we define* apply *as a buffer:*

$$\text{apply}(f, \boldsymbol{e}, \boldsymbol{b}) = (\mu_{\boldsymbol{b}}, p_{\boldsymbol{b}}, f(e[1], b[1]) \dots f(e[1], b[\ell(\boldsymbol{b})]))$$

Operator apply is used to apply a sample function on a buffer with a control value. We generalize apply to any number of one-sample buffers and any number of buffers with more than one sample with *the same size*.

Audio graphs and nodes

To a node $v \in V$ with p inputs and q outputs, we associate a function $f_v : S^p \to S^q$. Similarly, a graph G with n sources and m sinks is denoted by a function $g_G : S^n \to S^m$.

Definition 21 (Isochronous streams)**.** *We say that streams* s *and* s' *are* isochronous *if and only if* $\text{dom}(s) = \text{dom}(s')$*. We note* $s \doteq s'$*.*

If we have any given streams s and s', then $s_{\restriction \text{dom}(s) \cap \text{dom}(s')} \doteq s'_{\restriction \text{dom}(s) \cap \text{dom}(s')}$, where $s_{\restriction A}$ is the restriction of the definition domain of s to A. Given two isochronous streams s and s', if s is buffer-periodic, then s' is buffer-periodic with the same buffer-period.

Property 7 (map(i) and isochrony)**.** *Let* $i \in \mathbb{N} \setminus \{0\}$ *and* $s_1, \dots, s_i$ *i streams and an operator* $\text{operator} : B^i \to B^j$*. We have:*

$$\text{map(i)}(s_1, \dots, s_i, \text{operator}) = \text{map(i)}(s_{1 \restriction \cap_{k=1}^{i} \text{dom}(s_k)}, \dots, s_{i \restriction \cap_{k=1}^{i} \text{dom}(s_k)}, \text{operator}) \tag{5.6}$$

and if we note $s'_1, \dots, s'_j \in S^j$ *the result, then:*

$$\forall k' \in \{1, \dots, j\},\ \text{dom}(s'_{k'}) = \cap_{k=1}^{i} \text{dom}(s_k) \tag{5.7}$$

meaning $s'_1 \doteq \dots \doteq s'_j$*.*

As we will see in the execution semantics of an audio graph in Section 5.2, to execute an audio node, we will make its input streams *isochronous*.

5.2. Semantics

We present the semantics of an audio graph on streams, where we are concerned both on how the signals are transformed and when they are transformed. Our semantics is a *denotational* semantics that can split a buffer unevenly for more precision. We can split buffers at the time of the control (*sample-accuracy*), as in Rule 5.19, or snap control to the buffer boundaries (*block-accuracy*), as in Rule 5.18.

Operator	Domain	Codomain
first	S	$\mathcal{T}$
last	S	$\bar{\mathcal{T}}$
next	$S \times \mathcal{T}$	$\mathcal{T}$
tail	S	$\mathcal{T}$
substream	$S \times \mathcal{T} \times \mathcal{T}$	S
fuse	P	P
split	$S \times \mathbb{N} \setminus \{0\}$	S
bufferize	$S \times \mathcal{P}$	P
snap	$S \times \mathcal{T} \times B$	S
interleave	$S \times S$	S
periodicize	$S \times \mathcal{P} \times \mathbb{N} \setminus \{0\}$	P
map	$S \times (B \to B)$	S
decimation	$B \times \mathbb{N} \setminus \{0\}$	B
expansion	$B \times \mathbb{N} \setminus \{0\}$	B

Table 5.1.: Operators on streams and on buffers. S is the set of streams, B is the set of buffers; $\mathcal{T}$ is the set of timestamps; $\mathcal{P}$ is the set of periods. We note P the set of sample-periodic streams.

Type of a stream

An audio graph transforms its input streams into its output streams. Its semantics depends on its typing, as we will not process buffers the same way for *buffered* and *elastic*. We give types for the constants in the audio graphs, *i.e.* the sources and sinks. We define a function $\mathcal{M}$ that given a type associates streams $s \in S$.

- $\mathcal{M}(\textit{buffered}(p, n, e))$ is the set of sample-periodic streams s with sample-period p, buffer-periodic with buffer-period $n \times p$, with sample type e.
- $\mathcal{M}(\textit{elastic}(p, n, e))$ is the set of streams that are sample-periodic and have a sample-period p, buffer-periodic with buffer-period $n \times p$, with sample type e.
- $\mathcal{M}(\textit{periodic}(\pi, e))$ is the set of streams that are buffer-periodic with buffer-period π, with sample type e.
- $\mathcal{M}(\textit{aperiodic}(e))$ is the set of streams whose samples have type e and where we assume no additional properties.

Types ***buffered***, ***periodic*** and ***aperiodic*** correspond to increasingly lenient constraints on the shape of the stream (sample-period, buffer-period). Type ***elastic*** adds additional information on how the stream is processed by the node. Type ***aperiodic*** will typically correspond to some control.

For the reciprocal function $\mathcal{M}^{-1}$, we choose that a sample-periodic stream with sample-period p, buffer-periodic with buffer-period π, with types of samples e has type ***buffered***$(p, \frac{\pi}{p}, e)$ (not ***elastic***). It is not an issue due to the subtyping rules.

Periodic execution

All nodes of graph G must be executed within a periodic audio tick of period T, by an audio callback that is fed by and feeds the soundcard.

With periodic execution, a periodic schedule is built from the periodic types. We compute a base tick τ, which can be different from the audio callback tick T, such that:

$$\tau = \gcd(\pi_1, \ldots, \pi_m) \tag{5.8}$$

where $\pi_1, \ldots, \pi_m$ are the buffer-periods of all the stream periodic types appearing in the audio graph, *i.e.* ***periodic***(π, e), and ***buffered***(p_i, n_i, e_i) and ***elastic***(p_i, n_i, e_i) where $\pi_i = p_i \times n_i$. If all the sample-periods are the same and there are only ***buffered*** or ***elastic*** types, as it is in an audio graph with only one samplerate, τ becomes the greatest common divisor of the buffer sizes in the types.

The denotation function $[\![v]\!]$ for a node $v \in V$ is a function $S^{i(v)} \to S^{o(v)}$.

Special nodes. We give the semantics of special nodes on streams. Let x be a stream.

$$[\![\texttt{fuse}(n)]\!](x) = \text{mapsubstreams}(x, \text{fuse}, n \times p_{\text{first}(x)} \times \ell(x(\text{first}(x)))) \tag{5.9}$$

$$[\![\texttt{split}(n)]\!](x) = \text{flatmap}(x, \text{splite}(n)) \tag{5.10}$$

$$[\![\texttt{decimation}(n)]\!](x) = \text{map}(x, \text{decimation}(n)) \tag{5.11}$$

$$[\![\texttt{expansion}(n)]\!](x) = \text{map}(x, \text{expansion}(n)) \tag{5.12}$$

$$[\![\texttt{periodicize}(p,n)]\!](x) = \text{periodicize}(x,p,n) \tag{5.13}$$

$$[\![\texttt{delay}(n,\boldsymbol{b})]\!](x) = \lambda t. \begin{cases} x(t - p_{x(\text{first}(x))} \times n) \text{ if } t \geq p_{x(\text{first}(x))} \times n \\ \boldsymbol{b} \text{ if } t < p_{x(\text{first}(x))} \times n \end{cases} \tag{5.14}$$

$$[\![\texttt{window}(n,m)]\!](x) = \text{window}(n,m)(x) \tag{5.15}$$

Simple nodes. We give here the equations for simple nodes. To simplify the equations, we only write these equations for the case $i(v) = 2$. They can easily be extended to nodes with an arbitrary number of inputs. Thanks to the subtyping rules and by permutating input types, among the 16 possible input types, we consider only 6 cases. Other cases are described in Table 5.2.

	buffered	*elastic*	*periodic*	*aperiodic*
buffered	5.16	5.16	5.17	5.18
elastic	5.16	5.16	5.17	5.19
periodic	5.17	5.17	5.17	5.18
aperiodic	5.18	5.19	5.18	5.20 or 5.21

Table 5.2.: The rules to apply for a given combination of types for a two =input node.

Let v a simple node.

1. $v := \texttt{mapb}(f;0,2;a',b')$ or $v := \texttt{maps}(f;0,2;a',b')$ and $v : \textit{buffered}(p,n,\alpha) \times \textit{buffered}(p,n,\beta) \to z_1 \times \cdots \times z_{a'+b'}$

   Both input streams are buffered with same period and buffer size.

$$\begin{aligned} [\![\texttt{mapb}(f;0,2;a',b')]\!](x,y) &= \text{map}(2)(x,y,f) \\ [\![\texttt{maps}(f;0,2;a',b')]\!](x,y) &= \text{map}(2)(x,y,\text{apply}(f)) \end{aligned} \tag{5.16}$$

2. $v := \texttt{mapb}(f;0,2;a',b')$ or $v := \texttt{maps}(f;0,2;a',b')$ and $v : \textit{elastic}(p,n,\alpha) \times \textit{elastic}(p,n,\beta) \to z_1 \times \cdots \times z_{a'+b'}$

   Both input streams are elastic with same period and buffer size. We use Rule 5.16.

3. $v := \texttt{mapb}(f; 0, 2; a', b')$ and $v : \textit{periodic}(\pi, \alpha) \times \textit{periodic}(\pi', \beta) \to z_1 \times \cdots \times z_{a'+b'}$
where $\pi = \pi'$.

$$[\![\texttt{mapb}(f; 0, 2; a', b')]\!](x, y) = \text{map}(2)(x, y, f) \tag{5.17}$$

4. $v := \texttt{mapb}(f; 1, 1; a', b')$ or $v := \texttt{maps}(f; 1, 1; a', b')$ and
$v : \textit{aperiodic}(\alpha) \times \textit{buffered}(p, n, \beta) \to z_1 \times \cdots \times z_{a'+b'}$

We have one aperiodic input stream (control) and one buffered input stream (audio). In that case we snap the aperiodic event to the periodic buffers.

$$\begin{aligned}[\![\texttt{mapb}(f; 1, 1; a', b')]\!](x, y) &= \text{map}(2)(\text{snap}(x, \text{dom}(y), \boldsymbol{b}_x), y, f) \\ [\![\texttt{maps}(f; 1, 1; a', b')]\!](x, y) &= \text{map}(2)(\text{snap}(x, \text{dom}(y), \boldsymbol{b}_x), y, \text{apply}(f))\end{aligned} \tag{5.18}$$

where $\boldsymbol{b}_x : \textit{buffer}(p, n, \alpha)$ is a default buffer. Typically, we choose the default buffer to represent silence, *i.e.*, samples inside are zeroes, and latency is 0. Note that snap will keep only one control value per time interval between two timestamps of x and discard the other ones.

5. $v := \texttt{maps}(f; 1, 1; a', b')$ and
$v : \textit{aperiodic}(\alpha) \times \textit{elastic}(p, n, \beta) \to z_1 \times \cdots \times z_{a'+b'}$

One input is *aperiodic* and the other is *elastic*. The *elastic* type allows us to split buffers to accommodate control. It makes it possible to have better precision (less latency) than with Rule 5.18.

$$\begin{aligned}[\![\texttt{maps}(f; 1, 1; a', b')]\!](x, y) = \text{map}(2)(&\\ &\text{interleave}(y, \text{snap}(x, T(x), \boldsymbol{b}_x)),\\ &\text{mapsubstreams}(2)(x, y, \text{fix}\,(\text{splitting})\,, p \times n),\\ &\text{apply}(f))\end{aligned} \tag{5.19}$$

where $\boldsymbol{b}_x : \textit{buffer}(p, n, \alpha)$ is a default value. Operator mapsubstreams grabs all control values of y in a buffer-period $p \times n$ of x and gives them to splitting which splits a buffer of the audio stream at each timestamp of the control stream and is defined as follows:

$$\begin{aligned}&\text{splitting(f)}(s, s') = \\ &\qquad \text{let } s'' = \text{split}(s', \left\lceil \frac{\text{first}(s) - \text{first}(s')}{p} \right\rceil) \text{ in} \\ &\qquad s''(\text{first}(s'')) \odot f(s''(\text{next}(\text{first}(s''))), s_{\restriction \text{tail}(s)})\end{aligned}$$

interleave$(y, \text{snap}(x, T(x), b_x))$ builds the new set of timestamps representing the new buffers, *i.e.* the timestamps of the audio streams and the timestamps of the control stream snapped at the level of samples, where $T(s) = \left\{p \times \left\lceil \frac{t}{p} \right\rceil \,\middle|\, t \in \text{dom}(s)\right\}$. Note that $\boldsymbol{b}_x$ is actually never used by snap as $\text{first}(T(x)) \geq \text{first}(x)$.

6. $v := \texttt{mapb}(f; 2, 0; a', b')$ ou $v := \texttt{mapb}(f; 2, 0; a', b')$ and $v : \textit{aperiodic}(\alpha) \times \textit{aperiodic}(\beta) \to z_1 \times \cdots \times z_{a'+b'}$

Both input streams are aperiodic. Two cases arise depending on whether there are audio outputs or not.

a) $b' > 0$ *i.e.* one of the outputs is an audio stream, with type $\textit{buffered}(p, n, \gamma)$ or $\textit{elastic}(p, n, \gamma)$.

The node must be executed so that it yields a buffer of its periodic stream at each activation. Therefore, we snap the two aperiodic inputs to the required buffer-period, with the set of timestamps $T = \{\min\{\text{first}(x), \text{first}(y)\} + p \times n \times k < \max\{\text{last}(x), \text{last}(y)\} \mid k \in \mathbb{N}\}$.

$$\begin{aligned}
[\![\texttt{mapb}(f; 2, 0; a', 1)]\!](x, y) &= \text{map}(2)(\text{snap}(x, T, \boldsymbol{b}_x), \text{snap}(y, T, \boldsymbol{b}_y), f) \\
[\![\texttt{maps}(f; 2, 0; a', 1)]\!](x, y) &= \text{map}(2)(\text{snap}(x, T, \boldsymbol{b}_x), \text{snap}(y, T, \boldsymbol{b}_y), \text{apply}(f))
\end{aligned} \tag{5.20}$$

where $\boldsymbol{b}_x : \textit{buffer}(p, 1, \alpha)$ and $\boldsymbol{b}_y : \textit{buffer}(p, 1, \beta)$ are default values.

b) $b' = 0$ *i.e.* none of the outputs is periodic.

We want to execute the node each time there is a new value either in x or in y.

$$\begin{aligned}
[\![\texttt{mapb}(f; 2, 0; a', 0)]\!](x, y) &= \text{map}(2)(\text{snap}(x, \text{interleave}(x, y), \boldsymbol{b}_x), \\
&\qquad \text{snap}(y, \text{interleave}(x, y), \boldsymbol{b}_y), f) \\
[\![\texttt{maps}(f; 2, 0; a', 0)]\!](x, y) &= \text{map}(2)(\text{snap}(x, \text{interleave}(x, y), \boldsymbol{b}_x), \\
&\qquad \text{snap}(y, \text{interleave}(x, y), \boldsymbol{b}_y), \text{apply}(f))
\end{aligned} \tag{5.21}$$

where $\boldsymbol{b}_x : \textit{buffer}(p, 1, \alpha)$ and $\boldsymbol{b}_y : \textit{buffer}(p, 1, \beta)$ are default values.

Note that in a well-typed audio graph, the following input types do not appear, for a node $v : z_1 \times z_2 \to w_1 \times \cdots \times z_{o(v)}$:

- $z_1 : \textit{buffered}(p, n, \alpha)$ and $z_2 : \textit{buffered}(p', n', \beta)$ where $p \times n \neq p' \times n'$. It means we need to explicitly insert `fuse` and `split` node if we want to

adapt between different buffer sizes, and use resampler nodes if we want to resample. It is what we conceptually do in the audio architecture in Chapter 6 by inserting `fuse` and `split` nodes with type variables between all processing nodes and let the actual types for `fuse` and `split` nodes be inferred.

- $\boxed{z_1 : \texttt{elastic}(p, n, \alpha) \text{ and } z_2 : \texttt{elastic}(p', n', \beta)}$ where $p \times n \neq p' \times n'$.
- $\boxed{z_1 : \texttt{buffered}(p, n, \alpha) \text{ and } z_2 : \texttt{periodic}(\pi', \beta)}$ where $p \times n \neq \pi'$.
- $\boxed{z_1 : \texttt{elastic}(p, n, \alpha) \text{ and } z_2 : \texttt{periodic}(\pi', \beta)}$ where $p \times n \neq \pi'$.
- Permutations of the previous types.

To extend to more than two audio inputs, we consider the subtyping rules and then apply rules where the audio inputs have all the same type. When there are several aperiodic inputs, we merge recursively the aperiodic inputs similarly to what we do in Rule 5.21 or Rule 5.20 depending on whether there are also audio inputs or not.

Executing the whole graph. Let G a graph with nodes in V with sources $v_1^i, \ldots, v_n^i$ and sinks $v_1^o, \ldots, v_m^o$.

The denotation function $[\![G]\!]$ for G is a function $S^n \to S^m$. We have:

$$[\![G]\!](v_1^i, \ldots, v_m^i) = (\text{fix}.F)_{\restriction\{y_1,\ldots,y_m\}}(v_1^i, \ldots, v_n^i) \tag{5.22}$$

where $x_{y_1}, \ldots, x_{y_m}$ is the permutation of the subtuple of $(c_1, \ldots, c_\nu)$ that corresponds to the sinks nodes $v_1^o, \ldots, \ldots v_m^o$. We restrict the tuple resulting from the fixpoint iteration of F to the indices $y_1, \ldots, y_m$. Finally, F is defined as follows:

$$F(s_1, \ldots, s_\nu) = [\![v_1]\!](\tau_1) + + \ldots + + [\![v_n]\!](\tau_n) \tag{5.23}$$

where τ_i is the tuple of streams from the streams of source nodes $v_1^i, \ldots, v_n^i$ and $++ : S^g \times S^h \to S^{g+h}$ is the concatenation on tuples.

Examples. We show here two examples of execution of the semantics on a typical audio node, with typical audio streams. We consider an adder, $+ : S \times S \to S$. $+$ can add any streams, periodic or aperiodic, together. It has two input streams, x and y. We note $t'' = \text{first}(x + y)$. In case we need a default value for the streams, we choose a one-sample buffer with a zero sample inside x(0). We will show two examples where what changes is the input streams. The streams in time are represented on Figures 5.9 and 5.10.

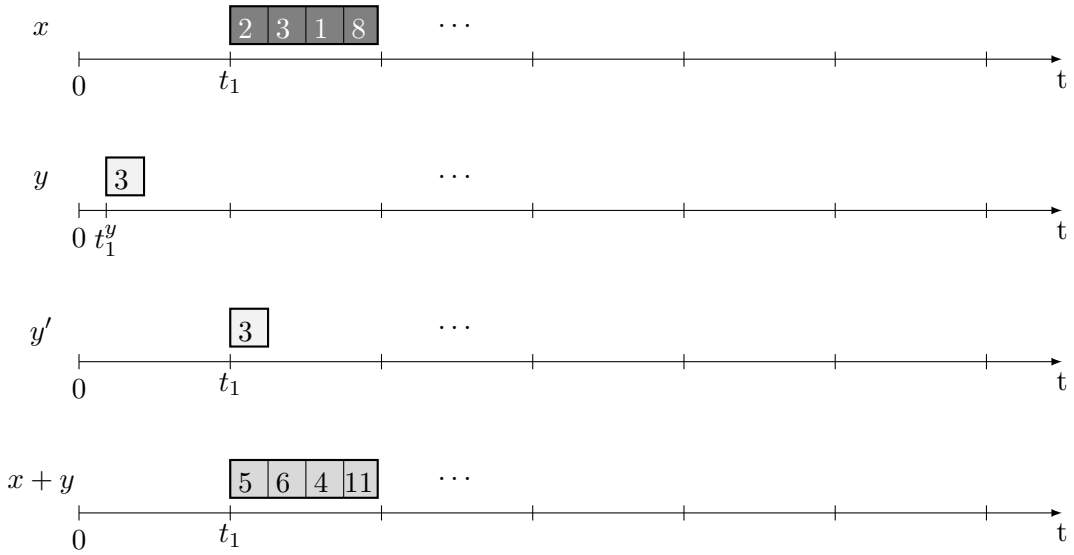


Figure 5.9.: An audio periodic generic stream and a control stream are inputs of a node. We only show the first buffers of the stream. The aperiodic control buffer is snapped to the periodic buffer boundaries, yielding stream y'.

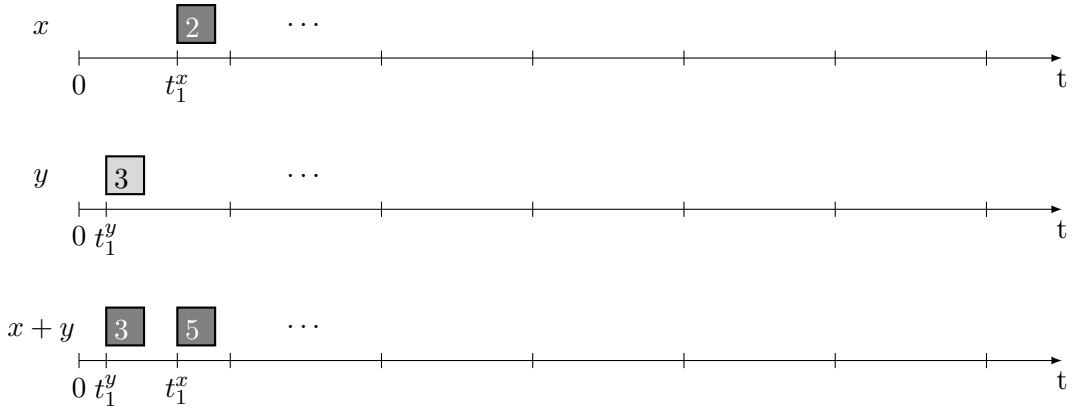


Figure 5.10.: Two aperiodic streams. We only show the first buffers of each stream. For each timestamp in one aperiodic stream, we generate a timestamp for the other periodic stream, where its value is its previous value or a default one.

- Figure 5.9. x is an audio stream with period p and y is a control aperiodic stream.

  We note $t_1 = \text{first}(x)$ and $t_1^y = \text{first}(y)$. We suppose that $b_{x(t_1)} = (2, 3, 1, 8)$ and $b_{y(t_1^y)} = (3)$. Rule 5.18 applies, as we cannot split the buffer in the periodic stream here.

  We have $b_{(x+y)(t_1)} = (5, 6, 4, 11)$, and $x + y$ has only one element at timestamp t_1.

- Figure 5.10. x and y are two control aperiodic streams and we generate control.

  Rule 5.20 applies. We note $\text{first}(x) = t_1^x$ and $\text{first}(t_1^y)$. We suppose that $b_{x(t_1^x)} = (2)$ and $b_{y(t_1^y)} = (3)$ and that $\text{first}(x) > \text{first}(y)$.

  $x + y$ has two elements, at timestamps t_1^y and t_1^x, and $b_{(x+y)(t_1^y)} = (2)$ and $b_{(x+y)(t_1^x)} = (5)$.

Soundness. We show the soundness of the semantics by showing the following preservation property: a well-typed term and a semantic rule yield a well-typed value that has the right type according to the typing rules. The composition of functions preserving types obviously preserves types, so it is enough to show the preservation for each predefined node. We show here the preservation on node `fuse`.

Let s a sample-periodic stream with sample-period p, buffer size n and element type e. Using $\mathcal{M}$ from Section 5.2, we get:

$$s : \texttt{buffered}(p, n, e) \text{ or } s : \texttt{elastic}(p, n, e)$$

Let $m \in \mathbb{N} \setminus \{0\}$.

Node `fuse`. The semantics rule for `fuse` gives:

$$s' = [\![\texttt{fuse}(m)]\!](s) = \text{mapsubstreams}(s, \text{fuse}, m \times p_{\text{first}(s)} \times \ell(s(\text{first}(s)))) \quad (5.24)$$

It means that a node `fuse` fuses slices of m contiguous buffers of size $\ell(s(\text{first}(s)))$. As s is buffer-periodic, $\ell(s(\text{first}(s))) = n$.

The resulting buffer of operator fuse has the same sample-period p as the sample-period of its input stream $\text{substream}(s, k \times m \times p_{\text{first}(s)} \times \ell(s(\text{first}(s))), k \times m \times p_{\text{first}(s)} \times \ell(s(\text{first}(s))))$, which is the $k + 1$-th slice of stream s, by Definition 12 and has length $m \times n$. Operator mapsubstreams builds a buffer-periodic stream with buffer-period $\pi = m \times n \times p$. We can apply Property 2, as the

buffer period $\pi = m \times n \times p = p \times (m \times n)$, so s' is sample-periodic. Besides, fuse does not touch the sample itself so we preserve the sample type.

As s' is sample-periodic with period p, buffer-periodic with buffer size $m \times n$, we can conclude that:

$$s' : \texttt{buffered}(p, m \times n, e) \text{ respectively } s' : \texttt{elastic}(p, m \times n, e)$$

This is the expected conclusion of typing rule (fuse).

5.3. Related work and comparison with the formalization

Here, we compare our type system (Chapter 4) and our semantics (Chapter 5) with what is done in IMSs and for more general-purpose signal processing languages, using the *dataflow model* and the *reactive synchronous model*. Chapter 2 provides a more general description of the various paradigms and languages.

The comparison focuses on the following points:

- how they handle control and audio;
- how buffering is modelled (or not);
- how multiple rates are handled.

Dataflow model

The *dataflow model* (see Section 2.4.1)) involves connected nodes, which communicate using fixed numbers of tokens, in the synchronous dataflow model [LM87]. When a node has received enough tokens, it can be fired and generate tokens. In this model, audio samples are represented by tokens and consuming or producing several tokens at the same time is a way of grouping them, as in *buffers*. The model allows different token consumption and production rates. The consistency of the rates is checked statically using the *balance equations*: the existence of a non-zero solution indicates that the rates are consistent. However, the multiple rates do not represent any timings in the classical dataflow models. We also think that using a type system to check the consistency of the rates is more user-friendly as it makes it possible to state precisely where in the audio graph the incompatible rates lie.

However, there are models based on the dataflow model that add timing information, such as the time-triggered dataflow model [AA09], which is well

suited for audio architectures with callbacks. In this model, nodes are divided into three types: *inputs*, *outputs* and *untimed* nodes. Inputs are time-triggered, *i.e.* started by the time-triggered callback, and outputs are time-restricted, *i.e.* they have a deadline and correspond to the end of the callback function. Not all sources are *inputs*, but all *inputs* are sources. Similarly, not all sinks are *outputs*, but all *outputs* are sinks, to represent that some sources and sinks drive the audio processing, such as a microphone or a speaker that needs buffer at a precise instant, whereas other ones, such as synthesizers, can adapt. In our own model, we can also model that by giving precise periodic types to the microphone and speaker nodes, and *generic* types (with type variables) to the synthesizer node. The time-triggered dataflow is also constrained by the callback activations, and does not describe precisely how control is intertwined with the audio processing.

Synchronous languages

Synchronous languages (Section 2.4.2) define a model for real-time execution of tasks. They also use a type system where types are called *clock types* to describe the consecutive values in time of a stream. Compared to clock types of synchronous languages which defines a sequence but not timings, our model allows for any timing and explain how an event with an arbitrary timing can be processed with other ones. A synchronous language would hide in the implementation how an event that can arrive at any time is actually bound to some clock. It is particularly crucial to keep track of these timings to be able to assess the precision and the latency in the audio graph. Besides, those languages are also used to define the processing nodes, contrary to ours where we take a higher-level view and see nodes as blackboxes with annotations.

Another difference lies on how we compute simultaneously on two streams: in synchronous languages, computations on several streams at the same time generate elements that are on the intersection of their two clock types or cannot be performed if the two clock types are different, depending on the language. In our semantics, nodes with several audio inputs compute on the intersection of the timestamps of their inputs, but control inputs (type `aperiodic`) behave differently. Those ones are snapped to the timestamps of audio inputs if there are audio inputs. If there are several control inputs, the computations operate on the union of the timestamps.

Our model has buffers as first-class citizens: a buffer is a grouping of consecutive samples that are processed at the same time, even though it represents a sequence of samples where each sample could have its own timestamp.

In usual synchronous languages, elements of a stream are dealt with independently, but recent works have studied how to group those consecutive elements into buffers, with type systems that describe the possible groupings.

A synchronous functional language with integer clocks [Gua16] This work describes a synchronous reactive language similar to Lustre, with more complex clock types, that can represent the grouping of several values in the same time instant. The usual clock type is a boolean clock type that represents the presence or absence of a value. On the contrary, an *integer clock* represents the number of values in a time instant. The clock types can also be hierarchical and provide a *local time scale*, in which time steps from a subprogram are hidden.

The model also involves operators on streams: `unpack`, which unpack segments (*i.e.* buffers for us) into one stream, is similar to our ϕ. However, as for the synchronous languages, the model does not embed timestamps; it encodes only the length of each segment. It is as if all streams in our model were sample-periodic, with a period fixed outside of the model, at implementation time.

Audio signal processing languages

Faust. A model for multirate Faust is presented in [OJ16]. The semantics use a function from periodic time domains, which can be seen as a set of periodic timestamps in our semantics, to multidimensional samples. It has upsampling and downsampling operators similar to the ones we presented: they decimate or expand by copying values and it is the task of the programmer to add filters afterwards. It also adds multidimensional operators `v` and `s` that vectorizes contiguous samples into a vector of samples, respectively serializes a vector of samples to individual samples. Although those operators can be used to describe a buffer-per-buffer semantics of Faust, they are mostly used to describe processing operations that cannot be effectively done sample-by-sample, such as a FFT. Indeed, the idealized semantics of Faust is to process signals sample-per-sample, although the implementation is buffer-per-buffer. In our semantics, we capture this buffer-per-buffer approach, which enables us to describe more precisely how audio signals and control signals interact.

The type system encodes multiple rates, bounds on the set of values and the dimension of the samples. Sample types (including the size of the vectors) and rates are independent except for the rule for the serialize operator. As such, the inference of sample types is performed first, followed by the inference of rates. Expressions can be *rate-scalable*, *i.e.* their rate can be adjusted to

their context, as simple nodes can with *buffered* or *elastic* types when their sample-period or buffer size is a type variable.

Again for Faust, in [JO11], dependent vector types are introduced to achieve the same vectorization and serialization operations.

A preliminary implementation[4] was undertaken in a multirate Faust interpreter called Faustine [BWJ14]. A difficulty that appears with the multidimensional operators of Faust is the absence of higher-order functions that forces to implement operations using macros and access vectors element per element, leading to huge Faust expressions after macro expansion.

Another limitation is that the multidimensional model for Faust does not handle arbitrary *overlapping vectors*. Our domains can represent it and our type system can also address it using the *periodic*(π, α) type. It seems it is not satisfactory as we lose both sample-period and buffer size, but overlapping windowed streams are usually *directly* consumed by a function that will produce a value. For instance in the case of the FFT, a window will lead to *one sample*, which is not an audio sample but an array of frequency bins, for which one can deduce the sample-period, *i.e.* π, and the buffer size, *i.e.* 1.

Kronos. Kronos [NL09; Nor15] is a musical programming language that lets the compiler decide the signal rates dealt with by the signal processors, as the signal processors are rate-polymorphic (with constraints). The type system is based on System F_ω. It handles event streams and can also represent multidimensional signals, including overlapping windows. As Faust, it aims at "implement[ing] the bottom of the signal-processing stack well", not to "replace high-level composition systems". Our model rather targets higher abstractions by embedding audio effects coded outside with some type annotation (*buffered*, *elastic* and so on) to connect them effectively, and that is why a stream of buffers and not a stream of samples is our main abstraction of a signal.

Arrp. In Arrp [Leb16], signals are streams of multidimensional arrays. They are also considered as a multidimensional array, one dimension of which is infinite. Computations are written as recurrence equations on the signals. Array size is part of the array type. Indexing is restricted to quasi-affine expression, which allows upsampling and downsampling, as well as defining an overlapping window. Type checking and inference use the polyhedral model [Fea91].

[4] It does not process signals in real time, tackles a subset of Faust expressions and performs only dynamic type checking.

Marsyas Marsyas processes data as chunks called *slices* [BPT06]. A slice is a two-dimensional piece of data characterized by its number of samples, its number of observations and its sampling rate. The samples are the usual division in time of the signal, while observations refer to the dimensionality of the sample: there can be several observations per sample. This is similar to having unidimensional arrays as a possible sample type. Control updates are slice-synchronous but slices can have any size, and different sizes within the same stream.

Max/MSP and *PureData* In these two IMSs, the semantics of when a control is taken into account for a node that processes only control depends on the *temperature* of the inputs ports, which can be *hot* or *cold*. Our semantics is less versatile: each arrival of a control activates a computation, *i.e.* the timestamps of the activations are the union of the timestamps of the input streams.

Part II.

Implementation and optimization of audio graphs

6. Proof of concept of an architecture for extensible, dynamic, heterogeneous audio plugins

> First, the user experiments to develop (or appropriate) a signal-processing or synthesis routine to use as a building block. Next, one uses the plug-in definition facility to define the routine's inputs, outputs, and parameters (and then shares the plug-in with others, if desired).
>
> *(David Zicarelli)*

In this chapter, we apply the formal model of Part I to an actual IMS, Antescofo. Antescofo is a score follower and a music programming language (see Chapter 2 for a more detailed description). We have extended Antescofo, which is embedded in Max/MSP or Puredata as an external and did not process itself audio signals, with an audio extension. This paves the way to a standalone Antescofo, especially on embedded platforms. Our audio extension makes it possible to define an audio graph, and to reconfigure it during execution, as an Antescofo action, to embed Faust effects, a subset of OpenCV functions, and custom audio processing units. It uses the type system defined in Section 4 to help connecting together heterogeneous audio effects and has a partial[1] implementation of the semantics described in Chapter 5. This work has been previously described in my articles [Don+16; DG17].

we will give two examples of applications using this audio extension in Section 6.4.

[1] For instance, buffers are not split at a control timestamp in this implementation.

6.1. Audio plugins

Audio plugins are special third-party components that can be loaded into an audio system[2] to add functionalities [Izh17; GM03], especially in Digital Audio Workstations (DAWs) where they are inserted on the tracks to modify the sound. The audio system is called the *host*. Audio plugins usually aim at complying to a standardized interface that describes how they can be used in a specific audio system, or even better, a range of audio systems. Among the requirements of audio plugins, we can list the following important ones:

- Do not crash the host;
- Expose as many functionalities to the host as possible;
- Display a user interface to control the plugin;
- Provide functionalities to save and load commonly used parameters, *i.e. presets*.

A number of standards for audio plugins have emerged, each targetting a specific platform first, as described in Table 6.1.

Several kinds of audio plugins. Audio plugins can be divided into several categories:

- *Instrument*, where the plugin generates a sound per note, and that sound can be controlled through control parameters;
- *Effect*, that processes the input signal and output the resulting signal;
- *Analyzer and meter*, that help to understand the input signal and can display a spectrograph or a loudness meter, for instance.

Usually, the notes are transmitted to the plugin using MIDI [Loy85]. Some plugin specifications, such as the VST one or LV2,[3] also allows the plugin to modify MIDI information, *e.g.* by transposing or arpeggiating MIDI notes. Here we focus on plugins that generate or process audio, not MIDI.

Inputs and outputs. Some plugin formats support only a fixed number of inputs and outputs, whereas others such as VST 3 can dynamically change their inputs and outputs, as well as their category.

[2] They were first introduced with the release of Pro Tools III, as early as 1994, and VST, in Cubase version3.02 in 1996.

[3] `https://x42-plugins.com/x42/x42-midifilter`

Standard	Platforms	Characteristics
VST (Virtual Studio Technology)	*Windows*, macOS, Linux	
AU (AudioUnits)	macOS	
LADSPA (Linux Audio Developer's Simple Plugin API)	Linux	
DSSI (Disposable Soft Synth Interface)	Linux	add note events to LADSPA
LV2 (LADSPA Version 2)	Linux	
AAX (Avid Audio eXtension)	ProTools 10 and later	
RTAS	ProTools 10 and earlier	
Web audio plugin	web browsers	
MAX/MSP externals	Max	Similar to Puredata externals

Table 6.1.: The main standards of audio plugins, with the platforms on which they can be run. If there are several ones, there is often one which is the first and main target, and we emphasize it.

Control in audio plugins. Plugin capabilities for control parameters differ in how and when they handle the control parameters. When controls are modified unpredictably during a live performance, they are usually taken into account at the next buffer. However, audio plugins are often used in non-live contexts, where control is known in advance. It is called *automation* and recorded or drawn as a curve with various interpolations. In that case, *sample-accurate* automation becomes possible and is handled by some audio plugin formats, such as VST 3 or LV2.

Resampling, vector size. DAWs usually have a fixed sample rate and vector size (buffer size), which cannot be changed during playback or at different positions in the effect chain, and even sometimes require to restart the DAW is such a change is needed. In VST 3 for instance, the sample rate and the maximum vector size cannot change during audio processing.

Graphical interface. The audio plugin can expose its controllable parameters and some visualisations to the host, which will be in charge of generating an interface using generic graphical elements, or use a custom interface, as shown in Figure 6.1. In some plugin formats, such as VST 3, control parameters can be grouped semantically by category.

Capabilities of a plugin. The audio plugin informs the host of its capabilities: which control parameters it has, how many inputs and outputs and so on. It can also declare which extension or version of the specification it can handle. Some plugin formats deal with it directly in the binary code, by providing functions in the API that indicate the presence or absence of a functionality, as for the VST format. In LV2, the textual human-readable format turtle is used.

In the next sections, we describe an audio extension for Antescofo where we connect audio blackbox nodes together. These nodes can be Faust [LFO13] code, or can be custom-coded in C++ or any other coupled programming language. For instance, we developed an FFT node, in order to do spectral processing which is difficult to undertake in pure Faust. The *automation* in Antescofo is represented by the `Curve` instruction. In the cases when the control curve is known beforehands, the audio extension of Antescofo is able to do *sample-accurate automation.* Our extension does not only handle audio but can also handle other multimedia streams, such as video, thanks to its multirate

Figure 6.1.: Two possible graphical interfaces for the *amSynth* plugin[4] in the Carla host. On the left, the generic interface; on the right, the custom interface. Image from Linux Magazine Issue 175, June 2015.

capabilities. We do not aim at providing a graphical interface[5] but a textual interface to use in the Antescofo programming language. We describe the annotations we give to the nodes of an audio graph when defining them, which exposes what they can do, and we will explain later the audio architecture of the host, *i.e.* Antescofo, as well as the API that the nodes have to follow when programmed in C++and in Faust. Antescofo programs can dynamically change the audio graph during execution, which typical hosts cannot usually do.

6.2. Audio extension syntax

In Antescofo, signals flow through processing nodes, called *effects*, which transform samples, connected through audio *channels*. We will explicitly represent

[4] *amSynth* is an analog modelling synthesizer using subtractive synthesis. `https://amsynth.github.io/`

[5] Though it could be derived from the annotations we give to the node.

the channels. Effects can also be controlled with *control parameters* and generate *control* themselves. It is a partial implementation of the formal semantics of Part I.[6] Audio channels transport sample-periodic signals, and control parameters are represented by aperiodic signals. In the implementation, buffers are not split yet when controls happen, and control is only implicitly timestamped at the granularity of a buffer.

In the formal semantics, we explicitly defined nodes that would aggregate several buffers or rather return slices of buffers, *i.e.* operators fuse and split. Here, channels semantically act as fuse or split to adapt the streams between two nodes of incompatible types. Moreover, only types ***buffered*** for sample-periodic audio signals and ***aperiodic*** for control parameters are used here.

We do not give here a full formal syntax but some representative examples to get the gist of how to write an Antescofo program that natively processes audio signals.

6.2.1. Declaration of effects and channels

Effects and *channels* are declared in the score and are instantiated at parsing time, whereas the connections between *effects* can be changed all along the performance, using a *patch* action. The declaration

```
@dsp_def my_effect : type := dsp::F(arg1, ..., argn)
```

introduces an instance my_effect of a DSP node dsp::F with the optional type specification type. Giving an explicit type to a DSP node makes it possible to add further constraint to the type inference system, for instance to impose some constraints induced by the environment. The arguments in the right hand side are instantiation parameters (*e.g.* the size of a FFT window), which cannot be changed during execution. The effect dsp::F can be a builtin effect or can be defined in another DSP processing language, such as Faust [Don+16], for which effect can be defined with a @faust_def, as shown in Section 6.2.4.

Channels are declared only to specify the identifiers that can be used in a patch:

```
@dsp_channel $$my_channel
```

6.2.2. Type annotations

Explicit type annotations can be added by the programmer when defining an effect or instantiating it. We draw a distinction between control values and audio signal values by using sigils: control value types start by one `$` whereas

[6] The full semantics of the audio extension was actually written afterwards.

signal value types start by `$$`. Scalar types can be any Antescofo types: a float, a map, an array for instance. For signal types, we can precise the sampling rate, the buffer size and the element type, or use the wildcard `*` to add new type variables that will be inferred later. If we do not specify anything to the signal type, then it assumes a type variable by default. By default, it uses float as element type. We also handle int and multidimensional arrays of float or int as element type. For isochronous effects, where the sample rate is supposed to be the same for all input and outputs, it is indicated only once, as shown in Code 6.1. We have chosen to specify the sample rate and not the period in the actual syntax, in contrast to the formal semantics of Chapter 5, as it is more intuitive for audio programmers to speak about sample rate (44 100 Hz instead of 2.27×10^{-5} s).

```
$$, $ -[44100] ⟶ $$(256),$
```

Code 6.1: A type annotation that describes an effect with two inputs and two outputs. There is one audio input and one control input, and one audio output and one control output. The node is *isochronous* and uses a sampling rate of 44 100 Hz. We also impose the buffer size of the output audio signal to be 256 samples.

6.2.3. Connecting effects

Effects are connected together to create a dataflow graph, that typically takes an audio signal from the soundcard or the host environment, and sends back a transformed signal. In Antescofo, connecting *effects* is an elementary action in the score, called a *patch* action. Patches describe the dataflow graph in a functional style: it lists a number of equations with the outputs on the left-hand side, and the digital signal processor and its inputs on the right-hand side.

In Code 6.2, a builtin sampler that plays a wav sound file is connected to the audio output. The type specifies that the sampler takes one control input (to trigger the playback) and outputs two data: a signal (from the sound file) and a control value that indicates the end of the playback. The whenever construction defines a reaction which is performed each time its condition $end_sample is set to true. The boolean control variable $play_sample triggers the playback. Notice the intersection between the control variable in the program and the control variable in the patch. The patch plugs the sampler through control variables and links. When $play_sample is set to true, the sampler starts

```
$play_sample := false
$end_sample := false

@dsp_channel  $$out
@dsp_def dsp::my_sampler : $ -[88200]-> $$, $
    := dsp::sampler("sample.wav")

whenever($end_sample) { print "Playing␣Done" }

patch{
    $$out, $end_sample := dsp::my_sampler($play_sample)
    dsp::output[0]($$out)
}
; ...
$play_sample := true
```

Code 6.2: An Antescofo score where a sampler is connected to the soundcard output. The sampler used a 88200 sample rate, has one scalar control input to indicate when the sample must be played, one audio signal output and one scalar output to say when the sample has finished playing.

its playback. Once the playback has finished, the output `$end_sample` is set to true by the effect, which triggers the reaction, making it easy to loop a sample for instance.

As an Antescofo action, a `patch` action can be played after detecting some musical event, waiting for some delay, and can be synchronized with the usual synchronization strategies [Con+12] of Antescofo.

The graph held by a `patch` action is dynamic: subsequent `patch` actions modify the graph. This is a much more convenient and lightweight way than the huge connection matrices that are usually found in typical hosts such as Max/MSP, which require to precisely know the number of plugins in use.

6.2.4. Defining effects

Effects can be defined in two ways in the audio extension:

- with the Faust language, where audio variables in Faust represent audio channels for Antescofo, and interface variables in Faust represent control variables in Antescofo.
- with C++ using a dedicated API, which we use it to code nodes that cannot be programmed in Faust, such as nodes with multirate inputs.

With the Faust language

The @faust_def instruction starts a Faust processing node declaration, as in Code 6.3. In Antescofo, scalar variable names start[7] with a $. Variables carrying signals are distinguished from the scalar variables, by starting with $$.[8] The Faust definition of an effect shows the scalar or signal type of each input arguments. The signal outputs are inferred from the Faust code, by looking at the final process instruction. To bind variable names in Antescofo and control in the Faust code, we do not use Faust audio signals but the graphical interface elements: hslider for instance represents a GUI element in Faust and is considered as a control parameter in Antescofo. Similarly, Faust code can expose output controls to Antescofo with the vbargraph and hbargraph Faust instructions, that are typically used as GUI elements to display a meter showing the level of a signal.

```
@faust_def faust::Pitch($$audioIn, $hr1, $pitch, $psout)
{
    import("music.lib");
    w=2048;
    x=100;
    hr1=hslider("hr1", 0, -20000, 20000, 0.1);
    psout=hslider("psout", 0.5, 0, 1, 0.1);
    pitch=hslider("pitch", 0.5, 0, 20000, 0.1);

    ratio = (pitch+hr1)/pitch;
    semitones = 12*log(ratio)/log(2);

    transpose1 (sig) =
    fdelay1s(d,sig)*fmin(d/x,1) + fdelay1s(d+w,sig)*(1-fmin(d/x,1))
    with{
        i = 1 - ratio;
        d = i : (+ : +(w) : fmod(_,w)) ~ _;
    };

    process = _<:(transpose1):>_*psout;
}
```

Code 6.3: A pitch shifter programmed in Faust, declared as an effect in Antescofo. It has one audio input, $$audioIn, and three control parameters, $hr1, $hr2, and $psout, and one audio output. In the Faust code, the inputs and outputs are represented implicitly by the underscores (in the last line).

[7] It is reminiscent of sigils in Perl, and due to the birth of Antescofo as a language sending and receiving Max messages that can use arbitrary characters.

[8] The same symbols are used for the type declarations of nodes.

Faust effects are compiled at definition time using the Faust on-the-fly compiler in *libfaust* [LFO13], based on LLVM [LA04]. The *libfaust* library generates LLVM IR code from Faust code, which is then compiled by LLVM, taking advantage of the many optimization passes available in the compiler. Compiling at definition time in the score, *i.e.* when the score is first loaded in Antescofo, increases load times, but makes it easier to distribute the score on various architectures, without having to pre-compile the Faust code for several architectures.

With the C++ API

Faust cannot natively deal with multirate streams and also does not deal with signals carrying array values, which can be used to represent images of a video, for instance. In that case, we can use a dedicated C++ API to program native signal processing nodes.

Such an effect declares its activation period as it has to be *isochronous*, as seen in Code 6.4. The activation period can be defined as *generic*, which corresponds to a `periodic` type with period α in the type system. The types of the inputs and outputs, and the instantiation parameters, are also specified. To process the input signals using the provided controls, a signal processing node has to implement the `compute` method (see Code 6.5). Input and output buffers as well as control parameters are retrieved using methods of the parent class `DspNode`.

Antescofo variables used as input or output controls are accessed through the member variables `in_control_variables` and `out_control_variables` and can be manipulated as any other Antescofo variables.[9]

6.3. Audio architecture

We present the audio architecture of the host of the audio effects, *i.e.* Antescofo. We show how the audio graphs of Part I are represented, especially, how they pass audio through audio buffers. We detail the type checking and type inference. We also explain how Antescofo schedules the audio tasks.

The lifetime of an audio graph starts with a `patch` action; then we perform type checking and type inference, using a subset of the rules of Chapter 4, *i.e.* Rules for types `buffered` and `aperiodic`; then we compute the scheduling

[9] As it happens in a code typically called in an audio thread, whereas Antescofo control code operates in another thread, some codes are used to make sure that no synchronization problem as well as no blocking of the audio thread occur.

```
MyNode::MyNode(pre_antescofo*antesc, string id,
    DspPeriod* period,
vector<pair<string, DspStreamType*>*>& ins,
vector<pair<string, DspStreamType*>*>& outs,
const vector<Value>& params,
location* loc) : DspNode(antesc, id, *period, loc)


bool MyNode::infer_and_check_connections(int
    nb_inchannels, int nb_incontrolvars, int
    nb_outchannels, int nb_outcontrolvars, DspPeriod
    *period, vector<pair<string, DspStreamType *> *>
    &ins, vector<pair<string, DspStreamType *> *> &
    outs)
```

Code 6.4: All native signal processing nodes inherit from the DspNode class. DspPeriod refers to the activation period, and DspStreamType, to the types of the inputs and outputs. The parameters to give when instantiating are stored in params. Types for a given node are checked within the infer_and_check_connections method.

```
void DspCamera::compute()
{
    if(!is_valid)
        return;

    assert(sources.size() == 0);
    assert(destinations.size() == 1);
    // Process the outputs
    buffer outbuf = destinations[0] ⟶
        get_input_buffer();
    assert(outbuf.is_valid());
    // And write results in the output buffers
    copy_n(data, size, (unsigned char*) (outbuf.
        begin().get_pointer()));
}
```

Code 6.5: A typical compute method, that needs to be implemented by all signal processing effects.

of the nodes, and then the graph is executed, until a new patch action or the performance stops, as summed in Figure 6.2.

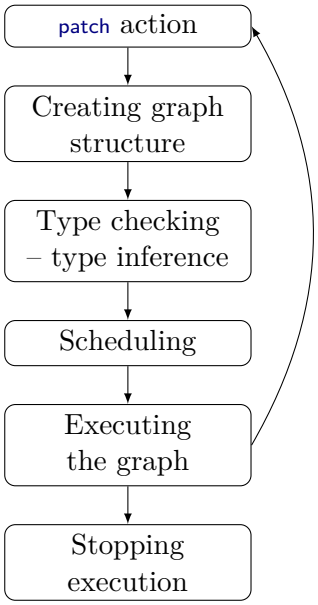


Figure 6.2.: Summary of the lifetime of a DSP graph.

6.3.1. Audio graph internal representation

The DSP graph is internally represented as a *bipartite graph*, alternating audio effect nodes and channel nodes, with channels storing internal audio buffers. Channels are also used to implement the `fork` node as described in Chapter 3, as shown in Figure 6.3. It means that they have one input but have several outputs to distribute the signal coming from one port to several ports. The DSP graph is created when a patch action in the Antescofo score is executed, after some specified event in the score. The effects as well as the channels are allocated before, at instantiation time.

A channel has an internal *circular buffer* that is used to adapt to the various rates. An *effect* connected to the input of a *channel* writes in the buffer, and nodes connected to the outputs read the buffer. We use virtual memory functionalities, *i.e.* the *mmap* sytem call on Linux and macOS, to remap the memory addresses after the end of the buffer to the start of the buffer itself. It ensures that we can directly give a pointer to the internal buffer without having to copy buffers when buffers span the end and the beginning of the

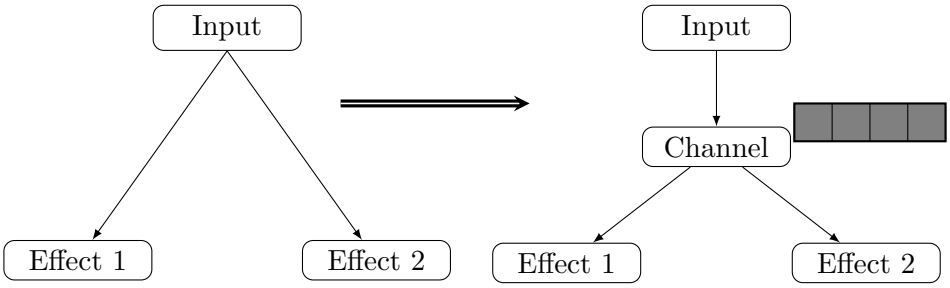


Figure 6.3.: DSP graph (left) and the associated *bipartite graphs* (right). Channels nodes hold buffers. We use one channel node per output port of a node.

circular buffer, thus optimizing for less copying. It also entails that we can only allocate memory on multiple of a page size, typically, 4 KiB. For a graph with 10 effects and 20 links, the memory consumption will be roughly 80 KiB. This is quite small for modern computers, even for small card boards such as Raspberry Pi.[10]

If the graph has no *sinks* or no output control parameters set to an Antescofo variable, we say it is not active and we do not execute it. It means that we assume that nodes do not have side effects.

If a DSP node or a link channel is not used in an active *patch*, the link and the related DSP nodes are disabled, as shown in Figure 6.4: removing a channel (resp. a node) from the audio graph also removes the subtree rooted by the channel (resp. the node). All links and nodes that are not connected to an output channel are also disabled.

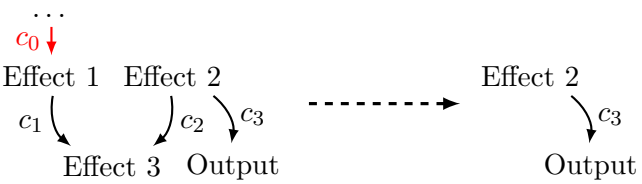


Figure 6.4.: Removing channel c_0 in the DSP graph. As Effect 1 and Effect 3 need buffers in channel c_0; Effect 1, Effect 3 and channel c_0, c_1, c_2 are removed from the graph. The incoming effects to Effect 1 that do not have any other outcoming path to the Output are also removed from the Dsp graph.

[10] The Raspberry 3 has 1 GiB RAM.

6.3.2. Type checking and type inference

Usually in an audio graph, most nodes will have a generic type, except *sources* and *sinks*, and embedded legacy effects that can only work with a precise buffer size or frequency. For instance, Faust effects are sets of equations on samples. The Faust compiler generates a compute function that is parametrized by the length of the processed buffers, hence accepts any buffer size.

Each time a patch action is executed in the Antescofo score, it defines a new graph, for which we need to apply type checking and type inference. We infer the actual types of the generic types, and we check that the already defined types are coherent. In a working DSP graph, *i.e.* a graph with at least one *sink*, at least one node has a *non-generic* type: the *sink*. A case where already defined types are not coherent happens when a *source* and a *sink* do not use the same sample rate and there are no resamplers along the paths in-between.

Channels are used as "impedance" adaptors and fuse and split the buffers of their incoming streams in order to adapt between types with different buffer sizes (but same sample-period), while maintaining the relation:

$$\forall j \in \{1, \dots, m\},\ \frac{\pi_{\text{input}}}{n_{\text{input}}} = \frac{\pi^j_{\text{output}}}{n^j_{\text{output}}} \tag{6.1}$$

where n_{input} is the buffer size and π_{input} the buffer-period of the input of the channel, respectively n^j_{output}, π^j_{output}, of an output of channel j, where the channel has m outputs. Conceptually, it is as if we inserted `fuse` and `split` nodes with type variables for the buffer size between all nodes of the graph.

Type inference and type checking are performed using a fixpoint algorithm: typing rules can be seen as functions that take premises and yield the conclusion. A fixpoint algorithm finds the solution by substitution, as described in Algorithm 1. The types of the nodes are then used for the *scheduling* of the audio graph.

Type checking and type inference are usually fast enough to be performed in real time, when audio graphs are small enough (hundreds of nodes as an order of magnitude).

6.3.3. Scheduling

The order of execution of nodes is computed using a *topological sort* of the graph.[11] The execution of the DSP graph is driven by a period called *DSP tick*. Every DSP tick, some nodes are activated; they consume some data

[11] We had assumed the graph is acyclic, see Chapter 3.1.

Algorithm 1 Fixpoint algorithm for type inference and type checking. If the fixpoint algorithm stabilizes, iterations are bounded by the diameter of the graph, so we compute our number of iterations with respect to that diameter. We perform successively period then buffer-size, then element-type inference. It means that we assume that we managed to compute all periods before computing buffer sizes. UPDATE functions of types use Equation 6.1 to propagate buffer sizes in addition to the typing rules of Chapter 4. We do not perform fixpoint iterations for the element type, as we do not allow type variables for it in the implementation. Variable nbNodes is the number of active nodes in the graph, and variable nodes is the list of active nodes in the graph. The order of nodes in the list depends on the order in which the nodes have been declared in the score.

```
changes ←false
maxIter ←2× nbNodes
repeat
    for all node in nodes do
        changes ←updatePeriod(node) ∨ changes
    end for
    maxIter ←maxIter - 1
until not changes ∨ (maxIter = 0)
changes ←false
maxIter ←2× nbNodes
repeat
    for all node in nodes do
        changes ←updateBuffersize(node) ∨ changes
    end for
    maxIter ←maxIter - 1
until not changes ∨ (maxIter = 0)
for all node in nodes do
    updateElementType(node)
end for
```

available in their input channels, perform some computations, and produce some data in the buffers of the output channels. The DSP tick is computed as the greatest common divisor (GCD) of all the periods, that is to say, the smallest tick that divides all the periods of all the nodes. A node is activated when its reaching the right number of DSP ticks, as shown in Figure 6.5.

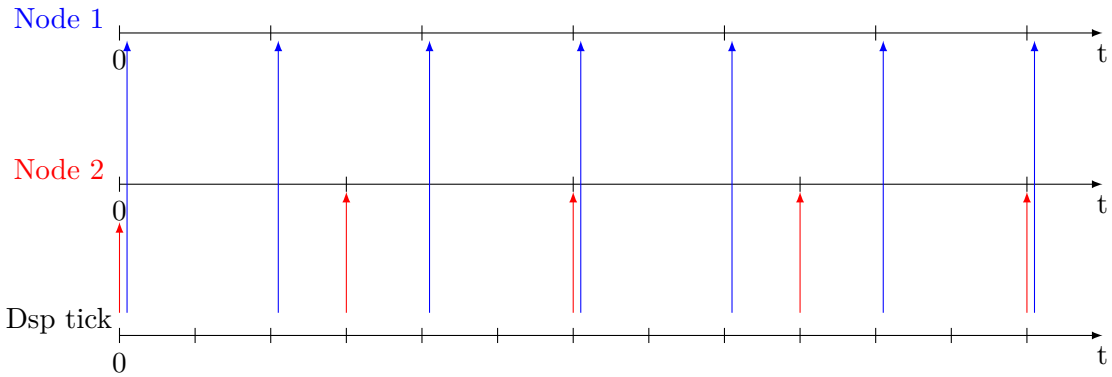


Figure 6.5.: Scheduling two audio nodes with different periods with activations on a DSP tick.

A particular period actually drives the audio computations: it is the *audio callback period*. At each audio callback activation, we advance the DSP tick the number of time required to cover all its duration and execute nodes accordingly, as computed in Algorithm 2 below. It means that we use the audio clock to schedule all nodes, including nodes that would process other kinds of signals, such as video signals. As the callback period is always among the periods, the DSP tick is always lower or equal to this callback period.

6.3.4. Sample-accurate control

Control computations, which are supposed to happen instantly, are not always taken into account right at the moment where they are computed. They are applied to a sample of the audio signal, which entails at least *sample-accuracy*. Due to limited computational resources available, samples are grouped into buffers. Hence, controls may be delayed until the end of the audio computations, *i.e.* DSP tick or audio callback period, thus decreasing the temporal accuracy of the system to *buffer-accuracy*. We described more precisely temporal precision in Chapter 5, and how we can increase temporal precision by

Algorithm 2 Computing timings of the next DSP tick. The audio callback is called repeatedly on buffers of n samples at a sample rate f so we can deduce its period $\frac{n}{f}$. The change in sample rate or buffer size in the callback does not change the timings either. `tickNum` is used to determine which node to execute during the period. `timeRemaining` is the time remaining before the end of the callback activation and `callbackPeriod` is the duration between two periodic calls of the audio callback. The `wait` instruction is useful if we want to have controls taken into account in the right tick, and not recompute the ticks if a timestamped control arrives after its associated DSP tick has been computed. If we just aim at callback period accuracy, we can remove this `wait`. Variable DSPTickPeriod is the duration of the dsp tick for the audio graph. PerformTick executes the nodes in the order of the schedule for all the samples for one DSPTickPeriod.

```
tickNum ←0                                          ▷ Global variable here
timeRemaining ←0                                    ▷ Global variable here
function AudioCallback(buffers, size, samplerate)
    callbackDuration ←size / samplerate
     ▷ timeRemaining is the time remaining at the end of the previous DSP
tick started in the previous callback invocation and is negative or zero.
    timeRemaining ←callBackPeriod + timeRemaining
    while timeRemaining > 0 do
        wait(callbackPeriod - timeRemaining)
                ▷ If we want DSP tick accuracy instead of callback accuracy.
        tickNum ←tickNum + 1
        PerformTick(tickNum)
        timeRemaining ←timeRemaining - DSPTickPeriod
    end while
end function
```

cutting buffers (which is not fully implemented here[12]).

The following outcomes with respect to *sample-accuracy* can occur, as summed up in Figure 6.6 for Antescofo:

- Audio computations are independent of any control parameter so the computations are *sample-accurate* by definition.
- Audio computations happen at buffer boundaries: *sample-accuracy*, as the buffer timestamp is also the timestamp of the first buffer here.
- Musical events are detected by the listening machine and signaled to the reactive engine (see Chapter 2), which triggers some control computations. We can achieve only *buffer-accuracy* at best as we cannot locate a musical event more precisely than at the buffer granularity. The event detection is handled with a spectral analysis on a buffer in the listening machine, which works on 4096-sample sliding windows at 44.10 kHz, with an overlap of 512 samples.
- Control computations are triggered by a delay or by an external event signaled by the environment, such as a keyboard event. It is again *buffer-accurate*, plus the latency of the system.
- Starting or reorganizing the audio computations (for instance, a `patch` action) can only happen at buffer boundaries. They are *buffer-accurate*.
- Control computations are driven by symbolic continuous data or discrete data known beforehands. Discrete data are read at buffer boundaries and are assumed constant during the next buffer computation, whereas symbolic continuous data are updated before each sample computation as their evolution in time is known *a priori*. In both cases, we achieve *sample-accuracy*. Note that this case is similar to what VST 3 or LV2 can do with automation curves.

A focus on *continuous* control variables. Control variables managed within the reactive engine can be taken into account during audio processing at the level of sample-accuracy, when they are tagged "continuous", which is denoted by starting their identifier with `$$`. Continuous variable can be used as ordinary Antescofo control variables. However, when their updates can be anticipated, because for instance they are used to sample a symbolic *curve* construct, this

[12] Control can happen at the middle of a buffer, but the timing of when this control happens is imprecise in the current implementation.

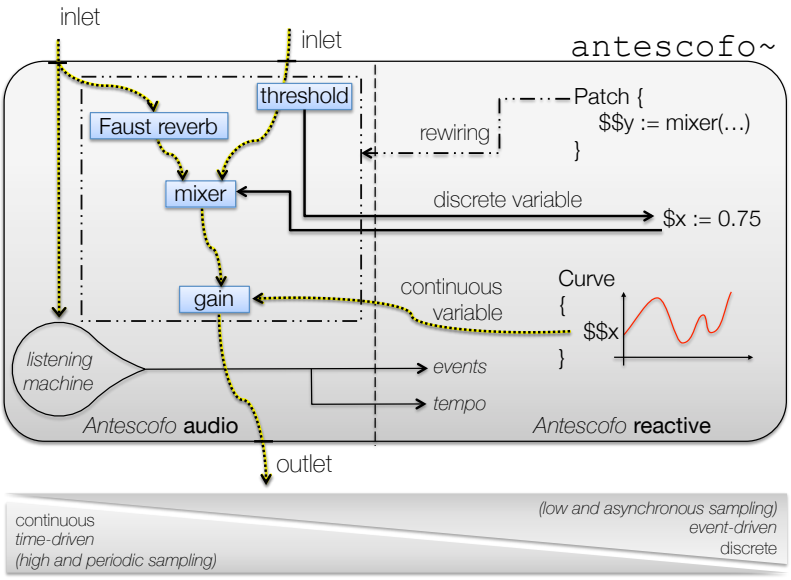


Figure 6.6.: Possible interactions between audio processing and reactive computations, *i.e.* control, in Antescofo.

knowledge is used to achieve sample accuracy in the corresponding audio processing. Figure 6.7 illustrates the difference; the top plots draw the values of the variable `$y` in relative and absolute time in the program:

```
Curve @grain 0.2s { $y { {0} 6 {6} } }
```

This curve construct specifies a linear ramp in time relative to the musician tempo. For the implementation, the control variable `$y` samples the curve every 0.20 s (notice that the sampling rate is here specified in absolute time) going from 0 to 6 in 6 beats. There are 3 changes in the tempo during the scan of the curve, which can be seen as slight changes in the curve slope in the right plots (these changes do not appear in relative time). The bottom plots figure the value of the continuous variable `$$y` (the same changes in the tempo are applied) defined by:

```
Curve @grain 0.2s { $$y { {0} 6 {6} } }
```

Despite the specification of the curve sampling rate (used within the reactive engine), the continuous control variable samples the line every 1/44100 = 0.02 ms during audio processing.

Sub-sample accuracy. If the date when the control happens is between two sampling dates, we can only take into account the control at the next sample.

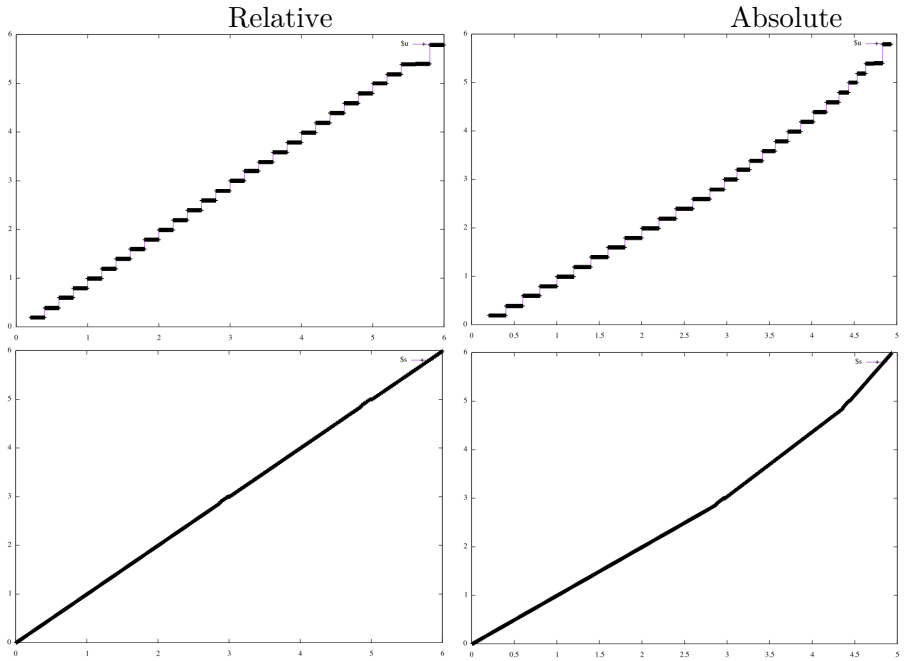


Figure 6.7.: *Top:* plot of the values of the variable `$y` in `Curve @grain 0.2s $y 0 6 6`,in relative and absolute times. There are 3 changes in the tempo during a linear ramp. *Bottom:* plot of the value of the continuous variable `$$y` in `Curve @grain 0.2s $$y 0 6 6`. The same changes in the tempo are applied.

To reach sub-sample accuracy, we would need to choose a value for the signal between the two samples, for instance, using sinc interpolation or keeping the last value. It means we assume properties about the considered signal because *in fine*, we are conceptually oversampling the signal compared to its nominal sampling rate. After getting this new value, we would also have to keep track that this sample is not evenly spaced compared to the other samples of a buffer, which we do not handle in our audio extension.

6.4. Applications

Here, we describe a piece and a proof of concept that use the audio extension of Antescofo, embedded into *PureData* [Puc02b] and an Udoo small board.[13]

6.4.1. Anthèmes II by Pierre Boulez

We present the beginning of the rendition of *Anthèmes II* (1997) by Pierre Boulez using the audio extension of Antescofo. This piece, which has entered the repertoire,[14] is for violin and live electronics. It has been implemented on multiple platforms, including Max and *PureData*. We started from an implementation of the piece that used Antescofo and *PureData* as described in the Antescofo tutorial [CG14].

Programming of Interactive Music pieces starts by a specification of the interactions, computing processes and relations between each other and with the physical world in form of an *Augmented Music Score*. Figure 6.8 (left) shows the beginning few bars of *"Anthèmes II"*, Section 1. The top staff, upper line, is the violin section for the human performer and in human-readable traditional Western musical notation; and the lower staves correspond to computer processes either for real-time processing of live violin sound (four *harmonizers* and *frequency shifter*), sound synthesis (two *samplers*), and spatial acoustics (artificial reverberation *IR*, and live spatialization of violin or effect sounds around the audience). Computer actions in Figure 6.8 are ordered and triggered either upon a previous action with a delay or onto an event from a human performer. Computer processes can also be chained (one sampler's output going into a reverb for example) and their activation is *dynamic* and depends on the human performer's interpretation.

[13] `www.udoo.org`

[14] Multiple performances can be watched on the Internet, for instance `https://www.youtube.com/watch?v=MzawnjOiccM`, played by Francesco d'Orazio in 2013 at the Biennale di Venezia, with Serge Lemouton from Ircam for the electronics.

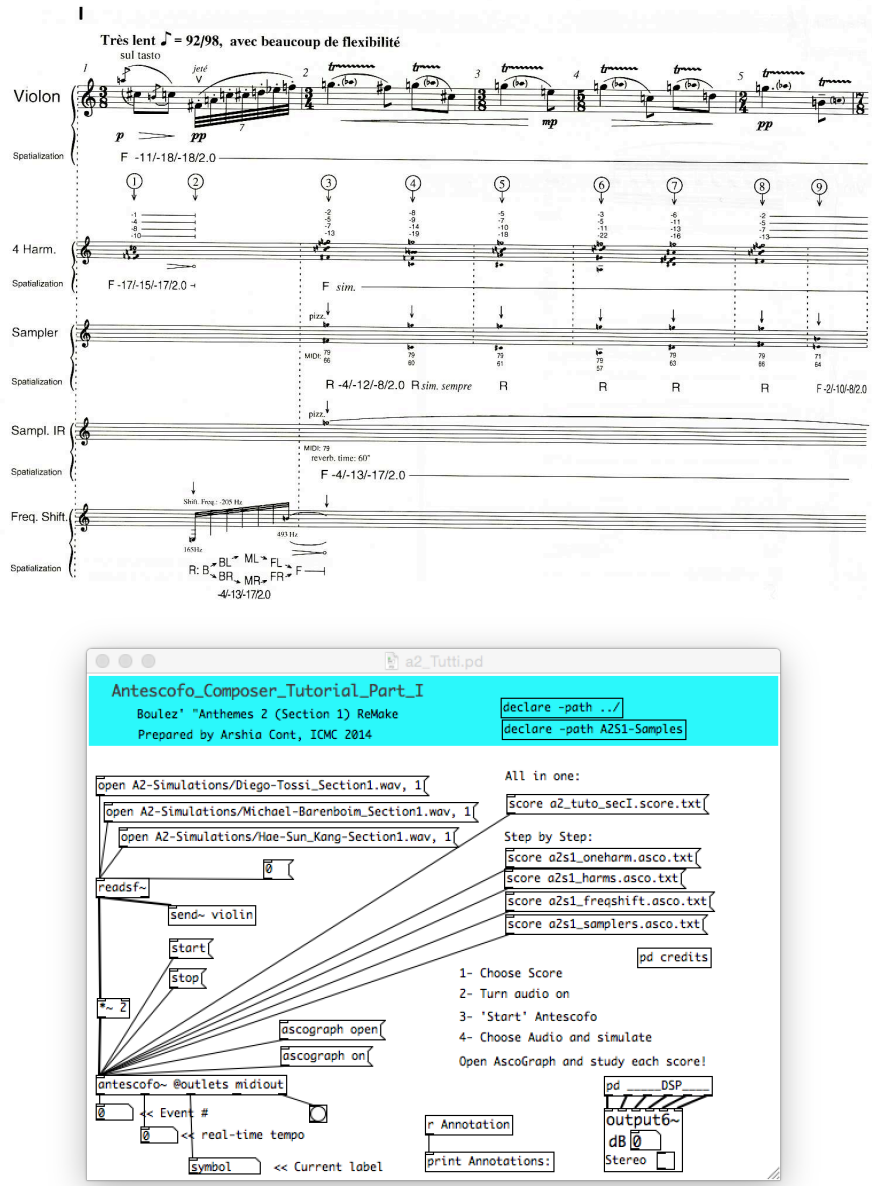


Figure 6.8.: *Top:* Composer's score excerpt of *Anthèmes 2* (Section 1) for Violin and Live Electronics (1997). *Bottom:* Main *PureData* patcher for *Anthèmes 2* (Section 1) from *Antescofo Composer Tutorial*.

Figure 6.8 (right) shows the main patcher window implementation of the electronic processes of the augmented score in *PureData*. The patch contains high-level processing modules, *Harmonizers*, *Samplers*, *Reverb*, *Frequency Shifting* and *Spatial Panning*, as sub-patchers. The temporal ordering of the audio processes is implicitly specified by a data-driven evaluation strategy for the dataflow graph. For example, the real-time scheduling mechanism in *PureData* is mostly based on a combination of control and signal processing in a Round-Robin fashion [RT08], where, during a scheduling tick, time-stamped actions, then DSP tasks, MIDI events and GUI events are executed in that order, as shown in Figure 6.9. Scheduling in *PureData* is thus block-synchronous, meaning that controls occur at the boundaries of audio processing. Furthermore, as in data-flow oriented languages, the audio processes activation, their control and most importantly their interaction with respect to the physical world (human violinist) cannot be specified nor controlled at the program level.

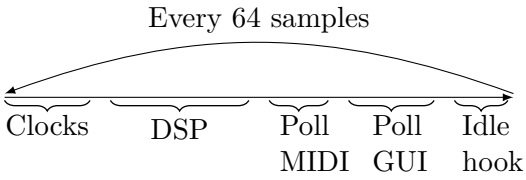


Figure 6.9.: Scheduling cycle in *PureData* (polling scheduler)

Using the audio extension of Antescofo instead of the digital signal processing capabilities of Antescofo makes it possible to control more finely the processes and to adapt the execution of the audio processing to what we know about the music scenario, whereas *PureData* merely received messages to control the effects. For instance, the symbolic curves of Antescofo that control some effects allow for sample-accurate control. It also leverages more efficient audio-processing nodes coded in Faust thanks to the embedded just-in-time Faust compiler. The audio graph of effects and links of the beginning of *Anthèmes II* is represented in Figure 6.10 and corresponds to the patch action of Code 6.6.

In Code 6.7, a level control (fs−out−db) and a DSP parameter (frequency shift value fd1_freq) are sent as messages to the *PureData* patch. Code 6.8 modifies directly Antescofo variables (resp. $psout and $freq) that were specified as controlling the pitch shifter in the previous patch declaration. Variable $psout is controlled as a curve in that case.

On a MacBook Pro, time-profiling the prototype of the embedded audio version with Faust against the message-passing implementation in *PureData*

```
patch{
    $$linkRev2 := faust::adaptor1($$audioIn)
    $$linkRev := faust::ir1($$linkRev2,$damping,$roomsize,$wet,$irout)
    $$linkFS := faust::Pitch($$audioIn, $freq, $aux, $psout)
    $$linkOutPanFS1,
    $$linkOutPanFS2,
    $$linkOutPanFS3,
    $$linkOutPanFS4,
    $$linkOutPanFS5,
    $$linkOutPanFS6 := faust::pannerFS($$linkFS, $s1FS, $s2FS, $s3FS,
                $s4FS, $s5FS, $s6FS)
    $$linkHarm := faust::Harms($$audioIn,$h1,$h2,$h3,$h4,$hrout)
    $$linkSampl, $endSample := dsp::sampler($play)

    $$linkOutPanSampler1,
    $$linkOutPanSampler2,
    $$linkOutPanSampler3,
    $$linkOutPanSampler4,
    $$linkOutPanSampler5,
    $$linkOutPanSampler6 := faust::pannerSampl($$linkSampl, $s1S, $s2S,
                $s3S, $s4S, $s5S, $s6S, $sampl2out)

    ;;;; Output audio:
    $$audioOut1,
    $$audioOut2,
    $$audioOut3,
    $$audioOut4,
    $$audioOut5,
    $$audioOut6 := faust::megaMixer($$audioIn, $$linkRev,
                $$linkHarm, $$linkOutPanFS1,
                $$linkOutPanFS2, $$linkOutPanFS3,
                    $$linkOutPanFS4,
                $$linkOutPanFS5, $$linkOutPanFS6,
                    $$linkOutPanSampler1,
                $$linkOutPanSampler2, $$linkOutPanSampler3,
                $$linkOutPanSampler4, $$linkOutPanSampler5,
                $$linkOutPanSampler6)
}
```

Code 6.6: The patch action for the beginning of *Anthèmes II* by Pierre Boulez. Figure 6.10 shows a more human-understandable visualization of the audio graph with the audio channels.

```
TRILL ( 8100 8200 ) 7/3   Q25
; bring level up to 0db in 25ms
fs-out-db 0.0 25
; frequency shift value
fd1_fre -205.0
```

Code 6.7: *Anthèmes II* score: message passing (old style)

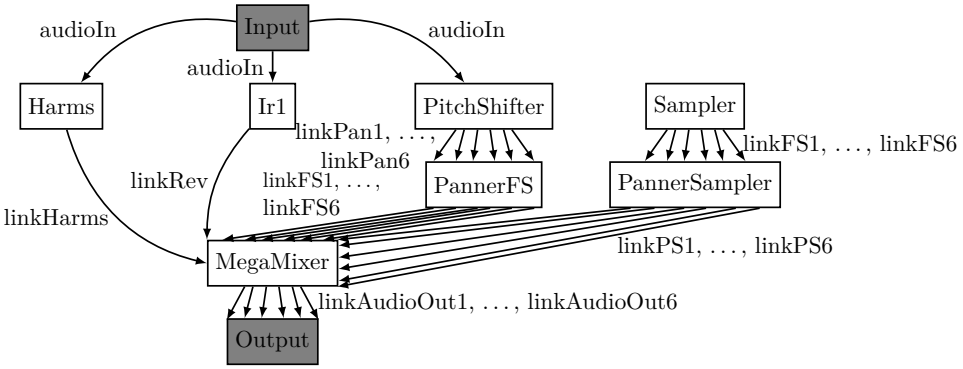


Figure 6.10.: Audio graph at the beginning of *Anthèmes 2* by Pierre Boulez, with the audio channels. The audio signal flows from *Input* to *Output*. We do not show the input and output controls here.

```
TRILL ( 8100 8200 ) 7/3   Q25
Curve c3   @grain := 1ms
{; bring level up to 0db in 25ms
    $psout
    {
        {0}
        25ms {1}
    }
}
$freq := -205   ; freq. shift value
```

Code 6.8: *Anthèmes II* score: embedded audio (new style)

shows a 12% system usage improvement in favour of the new implementation. It is due to the optimization of the DSP nodes thanks to the just-in-time Faust compiler but also to less overhead of controls, due the control Curve approach. However, these results are only one example and it is difficult to accurately evaluate the improvement, especially to instrument the program to measure its real-time behaviour.

6.4.2. Speed tracking and control of a synthesizer

In this example, we show how the audio extension of Antescofo can deal with streams with different sample rates, *i.e.* audio rate and video rate, and perform analysis on live video to control synthesis. The proof of concept does speed tracking of the largest foreground object in a video to control an audio effect. It can be used to roughly track the speed of a waving arm, for instance. The video input has typically a rate of an order of magnitude of 10 Hz, for example, 29.97 frames per second, whereas the audio output usually requires a 44.10 kHz sample rate to keep all human-audible frequencies. This shows that our architecture can accommodate both rates.

In Puredata [Puc+96] with GEM [Dan97], although Puredata makes it possible to change the sample rate in a subpatch using a `block~` object, it is difficult to have several rates live together in the same patch, as mixing video and audio would require. In GEM, a `gemHead` object creates [Zmö04] a state that can store images, and a pointer to this state is carried through the inlets and oulets in a Puredata `atom`, as a `gemList`, *i.e.* the frames are not carried as signals, only pointers to them. In Chuck [WCS16], results of unit analyzers are stored in an object called a `UAnaBlob` [WFC07], which contains a timestamp indicating when it was computed. In contrast, in Antescofo, spectral bins resulting from a FFT for instance would also be represented as a signal, but with a different rate depending on the parameters of the FFT.

Our type system ensures that frames can be carried safely and in a general way within the DSP graph: a video stream with a framerate "fps", seen as a stream of images of given width and height, will have a type such as

$$\mathit{periodic}(\frac{1}{\text{fps}}, 1, \mathit{Image}(\texttt{width}, \texttt{height}))$$

as shown in Figure 6.11. $\mathit{Image}(\texttt{width}, \texttt{height})$ is an alias for $\mathit{Array}(\mathit{int} \times \mathit{int} \times \mathit{int}, \texttt{width}, \texttt{height})$. The output of the speed tracking node is a control variable, which is updated for each frame. It means that we can further process this control variable, by detecting when it changes, with a whenever, as shown on Code 6.9. In Antescofo, the whenever control instruction watches a condition on

```
BPM 120

$speed := 0.
$max_speed := 15
$pitch_freq := 0;
$c0 := 16.35
$c7 := 2093.00

@faust_def faust::SimpleSynth($frequency)
{
    import("stdfaust.lib");

    freq = hslider("frequency", 16.35, 16.35, 2093.0, 0.01) : si.smoo ;

    process = os.osc(freq)  : re.mono_freeverb(0.5,0.5,0.5,23);
}

@dsp_def dsp::webcam := dsp::camera(0) % Select camera 0
@dsp_def dsp::tracking := dsp::speedtracking()

@dsp_def dsp::synth := dsp::SimpleSynth()

@dsp_def dsp::audioOut := dsp::output(0) % Output 0 of the soundcard

@dsp_channel $$video
@dsp_channel $$out

whenever($speed)
{
    print "Speed update"
    $pitch_freq := $c0 + @min($max_speed, $speed) * ($c7 - $c0) /
        $max_speed
}
% Wait for 6 beats
6 print Start

patch{
    $$video := dsp::webcam()
    $speed := dsp::tracking($$video)
    $$out:= dsp::synth($pitch_freq)
    := dsp::audioOut($$out)
}
40s print DONE DONE
```

Code 6.9: An Antescofo score that uses speed tracking of an arm to control a synthesizer.

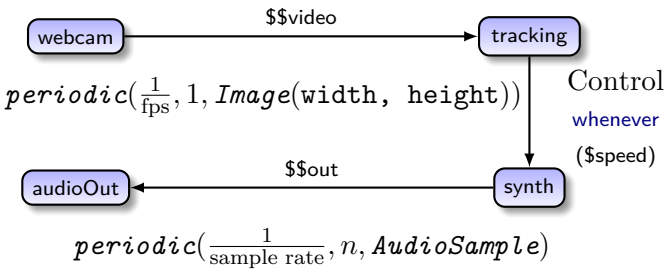


Figure 6.11.: The DSP graph is made of four main nodes: a input node connected to a video source (video camera or video file), a node that does speed tracking, a node that plays a sound, and an audio output, to the soundcard.

variables of the score and computes something when the values of the variables change and the condition evaluates to true. The code associated to that whenever computes here a frequency from the speed. After that, the frequency is used to drive a synthesizer which is coded in Faust.

Speed tracking. To track the speed of a foreground object, we embedded a subset of the OpenCV library [Bra00] in Antescofo. The speedtracking effect is a builtin effect coded in OpenCV. We extract the foreground using the Substractor Background MOG2 [ZV06], eroding and deleting the result to get rid of noise, and then detecting the contours and keeping the largest one with respect to its area, as shown on Figure 6.12. The speed is computed by measuring the displacement of its mass center, and smoothed. When the detected contour changes are higher than a given threshold, the speed is reset.

Synthesizer. The Faust effect is embedded in Antescofo as described in [Don+16]. The input frequency is smoothed and then used to drive a simple oscillator, to which we add some reverb using freeverb, an opensource implementation of a Schroeder/Moorer reverb model [Sch70].

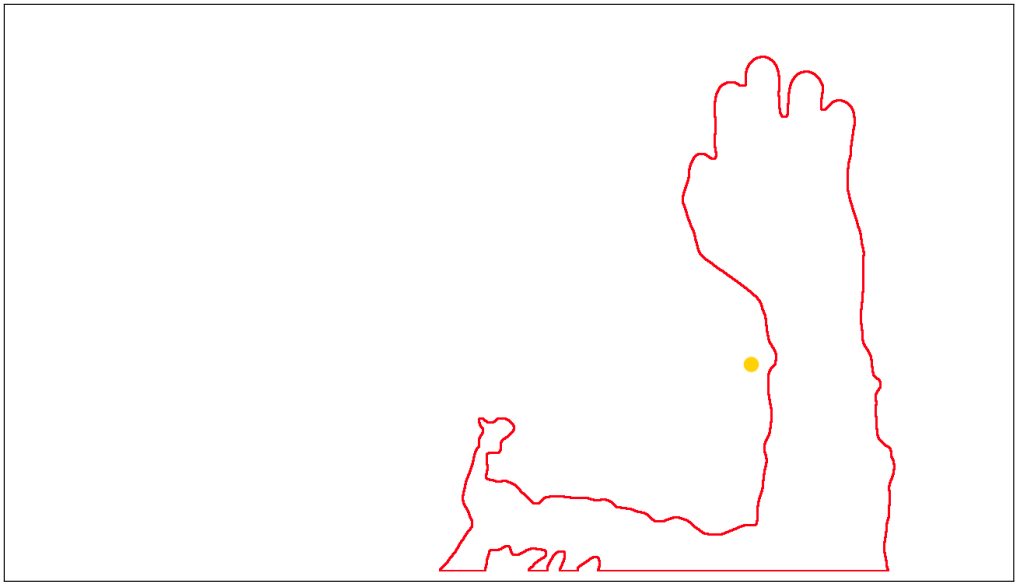


Figure 6.12.: Detection of waving arm and hand in a video using OpenCV. The centroid of the contour is the yellow point left to the wrist.

7. Offline optimization of audio graphs

> You do not want your audio to glitch. Period.
>
> *(Ross Bencina)*

IMSs are real-time systems (see Chapter 2.2) and missing a deadline entails glitches and cracks in the audio output. As in video streaming where the resolution of the stream is reduced if too many frames are dropped, or in audio compression with the mp3 format with respect to file sizes, we study a way of decreasing the execution time of an audio graph while not deteriorating too much the quality. We consider it as an optimization problem where given an audio graph as input, for instance a *PureData* patch, we want to output one or several degraded versions. The optimization will be performed through resampling parts of the audio graph. We will show how we find the nodes in the graph to resample. The degraded versions can be later used in a real-time context, as described in Chapter 8.

We start by presenting the *approximate computing paradigm* in Section 7.1. In Section 7.2, we describe in more details how resampling works and how we select nodes to degrade. In Section 7.3, we present a quality model of nodes and of graphs and, in Section 7.4, an execution time model. In Section 7.5, we describe our experiments and discuss them.

This work originates from our article [DGJ19].

7.1. Approximate computing

Approximate computing [Ven+15] is a paradigm of computation that allows some errors in computations in order to improve performance. It relaxes the concept of correctness, to a correctness with a quantified error. It is best suitable when *intrinsic application resilience* criteria apply.

Intrinsic application resilience consists of:

1. Not having an unique answer, but considering that a range of answers is acceptable;
2. Users who have got used to accepting good-enough results;
3. Noisy input data, and algorithms built to deal with this noise;
4. Using computation patterns that decrease approximations.

Approximate computing can be introduced at various layers of the computing stack: circuits, architecture, software, but also in methodology and tools, and as a cross-layer optimization. The goal is to design systems with a favourable quality versus performance or energy trade-off. It often needs at first a profiling or training executions step and depends on an application-dependent quality measure.

In [Zhu+12], a graph is used to represent a *map-reduce* program, with map nodes which compute and reduce nodes which aggregate data. Accuracy-aware transformations are separated into two classes.

Substitution transformations: they replace one implementation with another implementation. Functions have a propagation, a resource-consumption (energy, time, cost) and an accuracy specification.

Sampling transformations: they randomly subsample the input of a reduction node. They are characterized by a sampling rate.

The method is to randomly choose transformations to ensure a chosen trade-off between accuracy and resource consumption. It is dedicated exclusively to *map-reduce* applications, and does not natively embeds time constraints[1]. The transformations are chosen without taking into account any coherent sampling rate on a path. It also requires a preliminary phase of profiling, and hence cannot tackle dynamic graphs.

7.2. Optimization by resampling

We present here a way of optimizing audio graphs with the *approximate computing* paradigm, by *resampling* parts of the graph. The optimization requires to both choose parts of the graph suitable to be resampled, and to resample them. Resampling consists of changing the number of samples we use per second to represent an audio signal. If we downsample a part of the graph, the

[1] Though it may be possible to design an aggregate error metric that also takes time into account.

audio processing nodes operate on less samples and so the audio graph is executed more quickly. The downsampling will also degrade the quality of the signal. To select the best compromise between execution time and quality, we can enumerate all the possible degraded versions, or use heuristics.

7.2.1. Resampling as a degradation

Here we briefly present some signal processing theory results to understand better the impact of resampling on an audio signal.

The sampling theorem

A continuous-time signal is digitally represented as a discrete-time signal [OS14]. Given a continuous signal $s(t)$, the sampled signal using *sampling period* T is $s(nT)$ for $n \in \mathbb{N}$. The sampling rate f_s is the number of samples per second, $f_s = \frac{1}{T}$. *Sampling* is the process of converting $s(t)$ into $(s(nT))_{n\in\mathbb{N}}$.

Theorem 1 (Shannon's sampling theorem [Sha49])**.** *Given a sampled signal s with sampling rate f_s, with maximum frequency f_{max}, the following condition must be respected in order to reconstruct the continuous signal s:*

$$f_s \geq 2 \times f_{max} \tag{7.1}$$

f_{max} is called the Nyquist frequency*.*

Given a discrete-time signal $x[n]$, we can get the continuous-time signal $x(t)$ using the *Whittaker-Shannon interpolation formula*:

$$x(t) = \sum_{n=-\infty}^{+\infty} x[n] \ \operatorname{sinc}\left(\frac{t - nT}{T}\right) \tag{7.2}$$

where sinc is the *normalized cardinal sine function*, defined as follows:

$$\operatorname{sinc}(x) = \begin{cases} \frac{\sin(\pi x)}{\pi x} \text{ if } x \neq 0 \\ \lim_{x\to 0} \frac{\sin(\pi x)}{\pi x} = 1 \text{ if } x = 0 \end{cases} \tag{7.3}$$

Definition 22 (Band-limited signal)**.** A band-limited signal *is a signal whose frequency content has a bounded frequency content. f_{max} of Theorem 1 is also called* band limit*.*

Actual signals have finite duration and their frequency content does not usually have an upper bound.

Perception by the human ear. Above some frequency threshold, no quality improvement is perceived by human beings, according to psychoacoustics studies [RH11]. The human auditory system cannot perceive frequency above 20 kHz. Shannon's theorem implies that the sampling rate must be at least double of the maximum frequency, so about the sampling rate of audio CDs, at 44.10 kHz. However, oversampling makes it possible to better handle rounding errors that could occur during signal processing and that is why higher sampling rates than 44.10 kHz are often used.

Yet, the signal perceived by the human ear can be approximated to a bandlimited signal in practice.

How to resample

Resampling consists of modifying the discrete-time signal to use a different sampling rate. Thus, the sampling period is different and new samples need to be computed from the previous ones. Two equivalent approaches are possible: interpolating the old samples to get the new ones, or using filters. The second approach [Smi19b] is the one mainly in use in the field of digital audio processing. The quality of the resampling will depend on the quality of the filters, *e.g.*, no filter, a linear filter, or a windowed sinc filter.

Downsampling by an integer factor M. As the signal is downsampled, the band limit will be smaller and so we need to get rid of high frequencies to prevent *aliasing* (see 7.2.1). Thus, the signal first goes through a *lowpass filter*, the cutoff frequency of which is $\frac{f_s}{M}$. Then, the filtered signal is *decimated* by M, *i.e.*, only one sample every M^{th} is kept. Both steps can be computed conjointly if the filtering phase is done using a finite impulse response (FIR) filter. If $x[n]$ is the original signal, $y[n]$ is the resampled signal, and h is the impulse response with length K, we have:

$$y[n] = \sum_{k=0}^{K-1} x[nM - k] \cdot h[k] \tag{7.4}$$

Upsampling by an integer factor N. Two steps are enough to implement upsampling . First, the original signal is expanded by inserting $N - 1$ zeros between the original samples; then, the discontinuities entailed by the zeroes are smoothed using a *lowpass filter*, with cutoff frequency $\frac{f_s}{N}$. As for downsampling, the two steps can be combined using a FIR filter h of length K, where x is the original signal and y the upsampled signal:

$$
\begin{aligned}
y[n] = &\sum_{k=0}^{K} x[p-k] \cdot h[r+kN] \\
&\text{where } n = pN + r \\
&\text{and } r \equiv n \pmod{N}, r \in \{0, \ldots, N-1\}
\end{aligned}
\tag{7.5}
$$

Resampling by a rational factor $\frac{N}{M}$. The original signal is first *upsampled* by N and then *downsampled* by M. Both steps require a lowpass filter, so only one filtering with a cutoff frequency the lowest of the two is necessary.

Interpolation filters [Smi19a]. The input signal is convoluted with the filter. Several kinds of filters are used:

Zero-order hold: the filter keeps the last sample as the value for the current sample. It is similar to do a piecewise constant interpolation of the signal.

First-order hold: it performs a linear interpolation of the signal.

Windowed sync interpolation: the filter is, for L the window size and w a symmetric window such as Hamming, Blackman or Kaiser, and $\alpha < 1$:

$$
h_\Delta(n) = \begin{cases} w(n-\Delta)\ \mathrm{sinc}(\alpha(n-\Delta)), 0 \leq n \leq L-1 \\ 0 \text{ otherwise} \end{cases}
\tag{7.6}
$$

Degradations in the signal

Resampling can lead to degradations in the signal, due to *aliasing* and to *interpolation filter noise*.

Aliasing. Aliasing occurs when the high-frequency content of the original signal becomes undistinguishable from frequencies lower than the Nyquist frequency in the downsampled signal. Let us illustrate it on two sinusoid signals, $x_1(t) = \cos(2\pi(kf_e + f_0)t + \phi_0)$ and $x_2(t) = \cos(2\pi(kf_e - f_0)t - \phi_0)$ and let sample them at frequency f_e. It means the sampling period is $T = \frac{1}{f_e}$ and so we get the samples, for n such that $t = nT = \frac{n}{f_e}$:

$$
\begin{aligned}
x_1(t) &= \cos\left(\frac{2\pi n(kf_e + f_0)}{f_e} + \phi_0\right) \\
&= \cos\left(2nk\pi + 2\pi\frac{nf_0}{f_e} + \phi_0\right) \\
&= \cos\left(\frac{2\pi f_0 n}{f_e} + \phi_0\right)
\end{aligned}
$$

Similarly:

$$\begin{aligned} x_2(t) &= \cos\left(2nk\pi - 2\pi\frac{nf_0}{f_e} - \phi_0\right) \\ &= \cos\left(\frac{2\pi f_0 n}{f_e} + \phi_0\right) \end{aligned}$$

The samples from x_1 and from x_2 are indistinguishable. Hence, signals fold around the Nyquist frequency after downsampling: a signal with frequency f larger than $\frac{f_e}{2}$ is folded into $f_e - f$.

Interpolation/filter noise. To prevent aliasing from happening, frequencies above $\frac{f_2}{2}$ are disposed of by filtering them, but filters are not perfect and only attenuate the signal with an increasing slope. Just after $\frac{f_e}{2}$, the slope is small, but the signals there are folded into a non-audible part of the spectrum. When the signals are close to f_e, they are folded into audible parts, but the slope of the filters are higher. A windowed sinc filter is better at removing high frequencies than a linear one.

Resamplers as audio nodes

A resampler with resampler factor $r \in \mathbb{Q}$ is an audio node v with one input port and one output port. The buffer sizes of the stream on the input port n_i and the stream on the output port n_o are linked (see Chapter 4) such that $n_i = r \times n_o$.

Resampling is agnostic of the actual computation node semantics as it operates on the input signal itself, contrary to replacing a node by another degraded version, which requires to know the semantics of the computations in the node.

Definition 23 (Signal distance on a graph)**.** *Let G and G' be two audiographs with the same numbers of sources and sinks, n and m, that act on streams of S where elements are in $\mathcal{U}$, which we suppose is equipped with a norm $\|\cdot\|$. In practice for audio with elements in $\mathbb{R}$, we can choose the absolute value, or the Euclidian distance if we need continuity.*

We define the signal distance $d_s([\![G]\!], [\![G']\!])$ between $[\![G]\!]$ and $[\![G']\!]$ as:

$$d_s([\![G]\!], [\![G']\!]) = \max_{(s_1,\ldots,s_n)\in S^n} \|[\![G]\!](s_1,\ldots,s_n) - [\![G']\!](s_1,\ldots,s_n)\|$$

Proof. $d_s([\![G]\!], [\![G']\!])$ is actually the distance associated to the L^∞*norm.* □

Note that d_s is a distance on the signal of the graph, not on the structure of the graph. Indeed, $d_s([\![G]\!], [\![G']\!]) = 0$ does not entail necessarily that $G = G'$.

If we add an identity node in a graph, we will get the same signal but not the same structure.

7.2.2. An optimization problem under constraints

Given an audio graph, we want to find some *equivalent* degraded versions of it that maximize some criterion while respecting some constraints. The degradation is achieved by inserting resamplers on the edges of the graph, as shown on Figure 7.1, in order to resample parts of it. The premise here is that a node that receives less samples per activation will take less time to be executed.

When a downsampler node is inserted on a path, all the following nodes operate on a downsampled signal. We need to insert an upsampling node if there is a node on the path that enforces a specific sample rate, for instance, a sink to the soundcard.

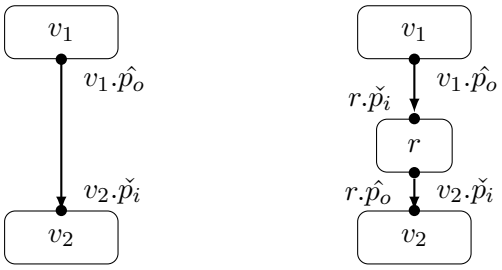


Figure 7.1.: Inserting a downsampling node r between nodes v_1 and v_2. v_1 has an output port $v_1.\hat{p_o}$, v_2 has an input port $v_2.\check{p_i}$ and r has an input port $r.\check{p_i}$ and an output port $r.\hat{p_o}$.

Insertion of a resampler

We note G the non-degraded audio graph.

Definition 24 (Resampling of the audio graph)**.** *A* resampling $\xi_r(G) = (V_{\xi_r(G)}, P_{\xi_r(G)}, E_{\xi_r(G)})$ *at resampling factor* $r \in \mathbb{Q} \setminus \{0\}$ *of an audio graph* $G = (V, P, E)$ *is a function* $\mathcal{G} \to \mathcal{G}$ *such that:*

Insertion of resamplers: $\exists e \in E, \exists e_1, e_2 \in E_{\xi_r(G)}, e = v.\hat{p_o} \to v'.\check{p_i}, e_1 = v.\hat{p_o} \to R(r).\check{p_i}, e_2 = R(r).\hat{p_o} \to v'.\check{p_i}$ *and* $R(r)$ *is a resampler node with resampling factor* r*. We insert it only on an edge between two audio ports (*`buffered` *or* `elastic` *stream types).*

Structure preservation: $\forall e \in E_{\xi_r(G)}, e \in E \vee \exists v \in V, e = v.\hat{p_o} \rightarrow R(r).\check{p_i} \vee e = R(r).\hat{p_o} \rightarrow v.\check{p_i}$ *where* $R(r)$ *is a resampler node with resampling factor* r.

Incoming path resampling: v *is a node of* G *and* $i(v) = n$*, we note the incoming paths to* v *in* $\xi_r(G)$*,* $v_1 \rightsquigarrow \cdots \rightsquigarrow v$ *up to* $v_n \rightsquigarrow \cdots \rightsquigarrow v$*. Those paths are the paths whose last edge is going to* v*. If* $v.\check{k}$ *has an audio type with sample-period* p' *in* $\xi_r(G)$*, whereas it has sample-period* $p = r \times p'$*, then all input audio ports of* v *are also resampled.*

Outcoming path resampling: *If* $v_1 \rightsquigarrow \cdots \rightsquigarrow v_n$ *is a path of* $\xi_r(G)$ *where* v_1 *is a resampler* $R(r)$ *and* v_n *is a sink, then there exists* $j \in \{2, \ldots, n-1\}$ *such that* $v_j = R(\frac{1}{r})$.

Type safety: $\xi_r(G)$ *is well-typed.*

We say that $\xi_r(G)$ *is a degraded version of* G.

Incoming path resampling is introduced to make sure that all inputs are resampled in the same way. It indicates that in case a node on a resampled path has several input ports and that one of them receives a resampled signal, we also have to resample the signal going into the other input ports by the same resampling factor, as shown on Figure 7.2. This is performed by inserting a resampling node connected to this input port. *Outcoming path resampling* ensures that a resampling by r is always followed by a reverse resampling, by $\frac{1}{r}$.

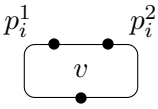


Figure 7.2.: We assume that node v is on path $v_1 \rightarrow \cdots \rightarrow v_n$. The resampled signal flows on this path through input port p_i^1 with resampling factor q. Node v has another input port, p_i^2. The signal coming into this port must also be resampled with resampling factor r.

Specific graphs. The *non-degraded graph* has no additional resamplers. It has the best quality and among the longest execution times (not necessarily the longest). If we should a resampling factor $r < 1$, the *fully-degraded graph* is obtained by having all the possible nodes resampled by r. It has the worst quality but among the shortest execution times per cycle.[2]

[2]Not necessarily the shortest indeed, as it requires to insert many resamplers, which adds to the overhead.

Rewriting the graph

The structure preservation property may lead to the insertion of many resamplers, which adds to the overhead of the degradations and entails a larger execution time. In some cases, we can merge the resamplers. For that, we perform a rewriting of the graph using rewriting rules that we describe here.

Merging outcoming resamplers from the same output port. If an output port p of node v is connected to several input ports $p_1, \ldots, p_n$, it is more efficient, with respect to the execution time, to insert the resampler and then a node with n outputs that distributes the signals, instead of inserting a resampler on each edge $p \to p_k$, as shown on Figure 7.3.

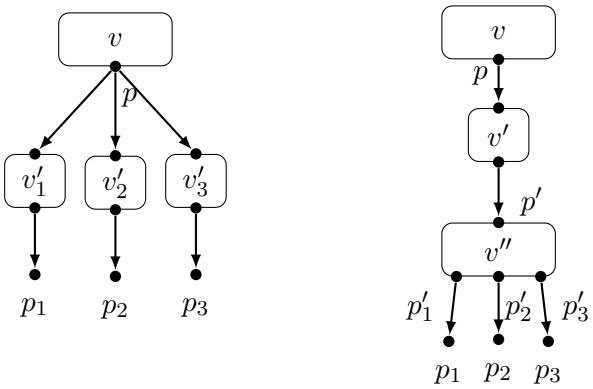


Figure 7.3.: Node v has one output port, p, which is connected to three input ports, p_1, p_2, p_2. On the left, we insert a resampler on each edge $p \to p_1, p \to p_2, p \to p_3$ with same resampling ratio, whereas on the right, we insert a node v'' with one input port p' and 3 outputs p'_1, p'_2, p'_3, and we insert the resampler v' on edge $p \to p'$.

Incoming resamplers into a mixer. When a mixer has only resamplers with the same resampling ratio as incoming nodes, we can remove those resamplers and rather insert one resampler after the graph, as shown on Figure. 7.4.

Downsampler followed by an upsampler. If a downsampler with resampling ratio $\rho < 1$ is immediatly followed by an upsampler with resampling ratio $\frac{1}{\rho} > 1$, both can be removed, as shown on Figure 7.5.

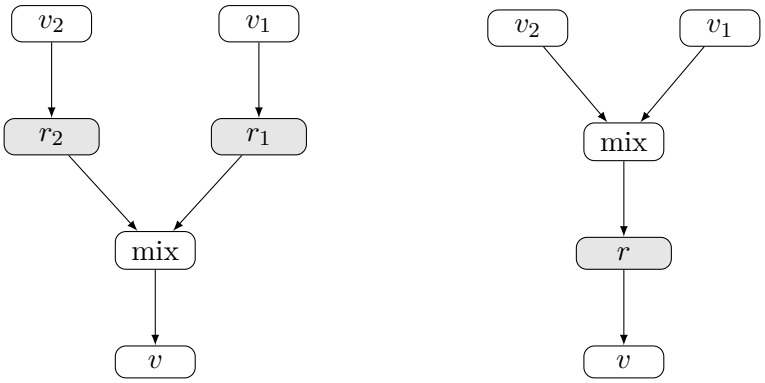


Figure 7.4.: Rewriting the graph in the presence of a mixer *mix*. Resamplers r_1 and r_2 with the same resampling ratio ρ are removed and a resampler with resampling ratio ρ is inserted after *mix*.

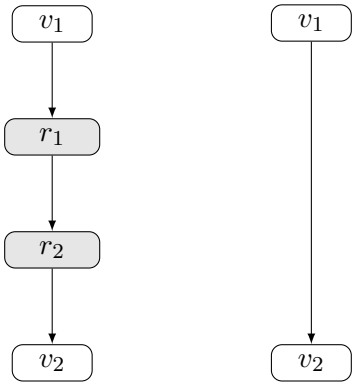


Figure 7.5.: Downsampler r_1 followed by an upsampler r_2, where both have the same resampling ratio.

Optimization problems

We can define the quality degradation as an optimization problem under constraints. Given a graph G, we want to:

- given a deadline, find the best possible quality;
- given a quality target, find the shortest possible execution time;
- find the Pareto front.

Best quality under time constraint. The optimization problem is, given a deadline D:

$$\text{maximize quality } q_G \text{ under the constraint:}$$

$$A_G < D$$

$$\text{where } A_G \text{ is the execution time of } G.$$

Best execution time under quality constraint. The optimization problem is, given a target quality q:

$$\text{minimize } A_G \text{ under the constraint:}$$

$$q_G > q$$

Pareto front. Our optimization problem is seen as a bi-objective optimization problem, where q_G has to be maximized and A_G has to be minimized, conjointly. There are no solutions that optimize both objectives: the non-degraded graph has the best quality but among the longest execution times, whereas the fully-degraded graph has the worst quality but among the shortest execution times. A solution is *Pareto-optimal* if it cannot be improved in any of the objectives without degrading another objective, *i.e.*, if there is no other solution that dominates it.

Definition 25 (Pareto domination)**.** *Given a graph G, ξ_1 and ξ_2 two resamplings, we say that ξ_1 dominates ξ_2 if:*

1. $q_{\xi_1(G)} \geq q_{\xi_2(G)}$ *and* $A_{\xi_1(G)} \leq A_{\xi_2(G)}$*;*
2. $q_{\xi_1(G)} > q_{\xi_2(G)}$ *or* $A_{\xi_1(G)} < A_{\xi_2(G)}$

The set of Pareto optimal solutions is called the *Pareto front*. In our case, with two criteria, we can visualize it on a graph with one criterion in abscissa and the other one in ordinate.

7.2.3. Exhaustive enumeration

We suppose a graph G has n nodes. To enumerate all the possible degraded versions, we can select all the possible subsets of nodes that can be degraded. After selecting a subset, we generate an intermediate rate graph where all edges are annotated with their sample rate. The nodes that can be resampled are the nodes for which one can insert a downsampler on a path leading to it and an upsampler on a path going from it, which excludes sources and sinks.The number of such possible nodes is $N = n - n_{\text{sources}} - n_{\text{sinks}}$. The maximum number of possible subsets is $2^N = 2^{n-n_{\text{sources}}-n_{\text{sinks}}}$.

After a subset of degraded nodes is chosen, adjacent nodes can be gathered in a degraded subtree and then resamplers are inserted at the start and end of the path, as shown in Algorithm 3 . For each path, we can choose a sampling rate inferior to the nominal sampling rate of the path in the list of admissible sampling rates.

Algorithm 3 Degrades a graph by inserting resamplers, given a graph where all edges where a degraded signal flows are marked with a boolean to_degrade. This works where there only two sample rates: one normal rate, and one degraded rate. For more rates, we use a integer storing the rate in Hz instead of the boolean.

```
function DEGRADE(graph)
    for edge in EDGES(graph) do
        source ←SOURCE(edge)
        pred_edges ←PREDECESSORS(source)
        pred_degraded ←all previous edges are set to to_degrade
        if (IS_EMPTY(pred_edges) or not pred_degraded) and edge.to_degrade then
            INSERTDOWNSAMPLER(edge) ▷ Insert a downsampler on the edge
        else if not IS_EMPTY(pred_edges) and not pred_degraded and not edge.to_degrade then
            INSERTUPSAMPLER(edge)       ▷ Insert an upsampler on the edge
        end if
    end for
end function
```

Random sampling. Enumerating all the degraded versions is exponential and becomes impractical for big graphs, so we can randomly sample a part of the degraded versions, by uniformly selecting subsets of nodes to be degraded. We

always make sure that the non-degraded graph and the fully degraded graphs are among the versions. For the experiments of Section 7.5, we will sample 64 graphs[3] in addition to the non-degraded graph.

7.2.4. Heuristics

The exhaustive enumeration is costly and can generate potentially hundreds of degraded versions. We also want more control on the degraded versions generated than with a random sampling. We present three heuristics: one resamples the longest shortest subpaths first, another one resamples from the outputs, and the last one resamples in the order of a topological sort of a graph. The last two ones are suitable for online degradation, when degrading a graph in real time.

First longest k-shortest path resampling

This heuristic selects first nodes and subpaths that are the slowest ones (according to the execution time model in Section 7.4.2) to execute among paths with a given number of nodes, in increasing size of number of nodes. The idea is to degrade first what takes a lot of time to be executed. Given a non-degraded graph G, we build a sequence of degraded graphs such that, for all $k \in \{1, \dots, n_{\max} - 1\}$, where $n_{\max}$ is the maximum number of degraded nodes,[4]

$$A_{v_1^{\xi_k(G)} \rightsquigarrow \dots \rightsquigarrow v_k^{\xi_k(G)}} < A_{v_1^{\xi_{k+1}(G)} \rightsquigarrow \dots \rightsquigarrow v_{k+1}^{\xi_{k+1}(G)}} \tag{7.7}$$

where $v_j^{\xi_k(G)}$ is a node of degraded graph $\xi_k(G)$.

We can find such a sequence by computing the ascending-ordered sequence $(l_k)_{k \in \{1,\dots,n_{\max}\}}$ of ascending-ordered k-longest paths and picking, if it exists, in each l_k, G_k such that $A_{G_k} > A_{G_{k-1}}$. Computing all the shortest paths in a directed acyclic is tractable, with algorithms such as the Jonhson algorithm or the Floyd-Marshal algorithm [Cor+09].

Resampling from the sinks

This heuristic downsamples increasingly larger subpaths that end at the sinks, in a depth-first backwards traversal way. Intuitively, downsampling nodes at the beginning of the audio graph is worse at degrading quality than downsampling at the end, as we lose some information on which we will compute later, and we cannot rebuild this information afterwards.

[3] Using a power of two makes it more efficient and easy to compute.

[4] $n_{\max} = n_G - n_{\text{sinks}} - n_{\text{sources}}$

For a graph G, the sequence of degraded graphs $(\xi_i(G))$ is defined as shown on Algorithm 4.

Algorithm 4 Computing the sequence of degraded graphs with a standard depth-first backward traversal. A node has two attributes, visited and to_degrade. to_degrade indicates that a node is included into the set of nodes to be degraded. For this heuristics, visited and to_degrade will actually have the same values. $\xi_i(G)$ is a sequence of graphs obtained from the non-degraded graph G, with $\xi_0(G) = G$. Note that modifying the attributes of currentNode modifies the graph.

```
function DEGRADEDGRAPHS(graph)
    i ←0
    ξ0(G) ←graph
    nodesToVisit ←SINKS(graph)
    while ISNOTEMPTY(nodesToVisit) do
        currentNode ←POP(nodesToVisit)
        currentNode.to_degrade ←true
        currentNode.visited ← true
        parents ←PARENTS(currentNode)
        nextNodes ←NOTVISITED(parents)
        PUSH(nodesToVisit, nextNodes)
        ξi(G) ←DEGRADE(graph)
        i ←i + 1
    end while
end function
```

Topologically-ordered resampling

Given a graph G with n nodes, we sort it topologically, leading to a sequence (v_i) of nodes, from which we deduce the sequence of degraded graphs (ξ_i). For $\xi_i(G)$, we enforce:

$$\begin{aligned} \forall k \geq i, \forall p \in \{1, \ldots, n\}, & v_k \rightsquigarrow \cdots \rightsquigarrow v_p \text{ is a path } \wedge\ o(v_p) = 0 \\ & \implies v_k \rightsquigarrow \cdots \rightsquigarrow v_p \text{ is a degraded path} \end{aligned} \tag{7.8}$$

7.3. A quality model for audio graphs

Here, we present models to evaluate the quality of an audio graph. The models should be easily and quickly computable on a graph given the nodes and its

structure. In Subsection 7.3.1, we present a general compositional quality model that can be adapted through choosing adequate qualities for nodes and adequate composition rules. In Subsection 7.3.2, we present how to determine empirical qualities and composition rules based on finite impulse responses of the individual nodes.

7.3.1. A general model of quality

The quality of an audio graph, of an audio signal in general, is a subjective matter and relates to psychoacoustics. It heavily depends on the semantics of the nodes and needs a reference graph that represents the best quality. We aim at presenting an *a priori* model of quality, that does not require to execute the full audio graph, which is necessary to be able to compute a quality in real time, and also to optimize an audio graph *offline* in a reasonable time.

The quality measure should be *compositional*, *i.e.* the quality of the graph $q_G \in [0, 1]$ must depend on the structure of the graph. The quality of a subgraph must be a function of the quality of its nodes and edges and of the incoming qualities on its inputs, not where the subgraph is placed in the graph. The worst quality is 0 and the best quality is 1. For each node v, we also note $q_v \in [0, 1]$ its quality (see below).

We define the quality $q_{v_1 \to \cdots \to v_n}$ on a path $v_1 \to \cdots \to v_n$ as $q_{v_1} \otimes \cdots \otimes q_{v_n}$ for an operator $\otimes$ with the following properties:

Associativity: $q_{v_1} \otimes (q_{v_2} \otimes q_{v_3}) = (q_{v_1} \otimes q_{v_2}) \otimes q_{v_3}$

Decreasing: $q_v \otimes q_{v'} \leq q_{v'}$ It means that quality never increases on the path, as the information lost by degrading cannot be rebuilt.

Identity element: There is an identity element $1_\otimes$ such that $1_\otimes \otimes v = v \otimes 1_\otimes = v$. Such an element is the quality which preserves for the output the quality of its input.

An obvious choice for an operator fulfilling these desired properties is multiplication on real numbers. For $v \to v'$:

$$q_{v \to v'} = q_v \otimes q_{v'} = q_v \times q_{v'}$$

On the path $v_1 \to \cdots \to v_n$, thanks to associativity:

$$q_{v_1 \to \cdots \to v_n} = \prod_{i=1}^{n} q_{v_i} \tag{7.9}$$

We also define a *join* operator $\oplus$ that models the quality resulting from joining two paths, such as $v_1 \to v_3$ and $v_1 \to v_2 \to v_3$ on Figure 2.11.

$$\begin{aligned} q_G &= q_{v_1 \to v_3} \oplus q_{v_1 \to v_2 \to v_3} \\ &= (q_{v_1} \otimes q_{v_3}) \oplus (q_{v_1} \otimes q_{v_2} \otimes q_{v_3}) \end{aligned} \tag{7.10}$$

In practice, we choose $\bigoplus_{k=1}^{n} q_k = \frac{1}{n} \sum_{k=1}^{n} q_k$ for n joining paths for mixer-like nodes and $\oplus = \min$ for the other nodes. For mixer-like nodes, we want to take into account that low-quality input streams can have very low volume and can be mixed with good-quality input streams, and hence have a good quality.

Assigning qualities to individual nodes

The *a priori* quality measure in the case of resampling is a function of the sample rate: the lower the sample rate, the lower the quality. In the case of a resampler with resampling ratio r on a stream with sample-period p, the output sample rate is $\frac{r}{p}$. If audio is sent too late to the output buffer, a click can be heard. We consider that it is worse to hear a click because of missing a deadline than to hear a resampled signal. A node that would entail always missing deadlines is given the worst possible quality, *i.e.*, 0. The quality q_v of a downsampled node is such that $q_v < 1_\otimes$. The quality of a non-downsampled node is 1.

7.3.2. Estimating quality of nodes using finite impulse responses

In 7.3.1, we choose the quality of an individual node to be in $[0, 1]$ using a heuristic. We can also be more precise and approximate all the digital effects by linear time-invariant digital filters.

Linear time-invariant digital filters (LTI filters)

Definition 26 (Linear filter)**.** *A filter v is* linear *if it has the following properties:*

scaling $\quad v(\lambda s) = \lambda v(s)$ *for* $\lambda \in \mathbb{R}$ *and s a signal*

superposition $v(s_1 + s_2) = v(s_1) + v(s_2)$ *for s_1, s_2 two signals*

LTI filters do not introduce new spectral components.

Definition 27 (Time-invariant filter)**.** *A filter v is* time-invariant *if, for a signal s, and for $N \in \mathbb{N}$:*

$$\forall n > N, v(s)[n - N] = v(\lambda i.s[i - N])[n]$$

It means that if the input signal is shifted by N samples, the output signal is also shifted by N samples.

Impulse response

Any LTI filter can be characterized by an *impulse response*. An *impulse response* is the output signal obtained from inputting a short signal with all frequencies, usually modelled as a Dirac delta or Kronecker delat function. A *frequency response* sampled on the frequency axis, with N samples, can be obtained from the impulse response:

$$\forall k \in \{0, \ldots, N-1\}, \mathcal{H}(k) = \frac{\mathrm{DFT}_k(o)}{\mathrm{DFT}_k(i)} \tag{7.11}$$

where i is the input signal and o is the output signal, and DFT_k is the discrete Fourier transform defined in Equation 7.12.

$$\mathrm{DFT}_k(x) = \sum_{n=0}^{N-1} x(n) e^{-j\omega_k nT} \text{ with } \omega_k = 2\pi f_s \frac{k}{N} \text{ where } f_s = \frac{1}{T} \tag{7.12}$$

We can measure the impulse response of the nodes of the audio graphs and how it increases or decreases the frequency content above the Nyquist frequency of the desired sampling rate. The quality q of the node in that case will be the average of the ratio of the magnitude of the original high-frequency content versus the degraded high-frequency one:

$$q = \frac{1}{N} \sum_{k=M}^{N} \left| \frac{\mathrm{DFT}_k(o)}{\mathrm{DFT}_k(i)} \right| \tag{7.13}$$

where M is the frequency band where the Nyquist frequency is located.

7.4. Ranking nodes by average execution time

In order to pick the quickest graphs, we need to be able to rank them by average execution time. To estimate the average case execution (ACET) time of an audio graph, possibly degraded, we need to have an estimation of the average execution of each kind of node in the graph and how to combine these estimations for the whole graph. We suppose that the perturbations on the execution time are independent for each node.

The ordering in execution times between two nodes, and thus on the whole graph, does not depend on the input buffer size. Hence, we choose a specific buffer size to perform all the measurements of execution time.

7.4.1. Average execution time of individual nodes

We measure the average execution time A_v of all the possible nodes that can be part of the audio graph. We do not care about the exact execution time, but rather of an ordering on the execution times between various versions of an audio graph. In addition, the execution time increases monotonically with the buffer input sizes, as shown on Figure 7.6. Thus, we only measure the average execution of a given buffer size, as shown on Table 7.1. However, some nodes can have a variable number of inputs and outputs, such has a `mixer` node. We do not want to measure all possible combinations of inputs and outputs. Experimentally, we find that:

$$\begin{aligned} A_{\text{mixer}}(n_{\text{inputs}}, n_{\text{outputs}}) = n_{\text{inputs}} \times \underbrace{(A_{\text{mixer}}(2,1) - A_{\text{mixer}}(1,1))}_{\text{cost of adding}} \\ + n_{\text{outputs}} \times \underbrace{(A_{\text{mixer}}(1,2) - A_{\text{mixer}}(1,1))}_{\text{cost of copying one buffer}} \end{aligned} \tag{7.14}$$

where $A_{\text{mixer}}(1,1)$ is a mixer with one input and one output, $A_{\text{mixer}}(2,1)$ transforms a stereo channel into a mono one by simply adding them, and $A_{\text{mixer}}(1,2)$ does not correspond really to a mixer but creates a stereo channel by copying twice its input.

Oscillator	Modulator	Linear resampler	Mixer 1-1	Mixer 2-1	Mixer 1-2
3.50	3.50	2.00	0.24	0.28	0.42

Table 7.1.: ACET for basic nodes for a buffer size of 256 samples on a MacBook Pro with 16 GiB RAM and 3.10 GHz processor with macOS Sierra. Execution times are in µs.

For the measurements, we use the rust benchmark library *criterion* [Hei19], which performs outlier classification, bootstrapping and linear regression between sample sizes to get robust statistics on the runs, as shown on Figure 7.7.

7.4.2. Average execution time on a path with degraded nodes and of the whole graph

Let G a graph and $\pi = v_1 \rightsquigarrow \cdots \rightsquigarrow v_n$ a path of G. We assume that the nodes in the path execute at the same initial rate.[5] We assume that the computation

[5] If a node v is executed twice as often as the other nodes, we can set $A_v^2 = 2 \times A_v$ and use subsequently A_v^2 instead of a_v.

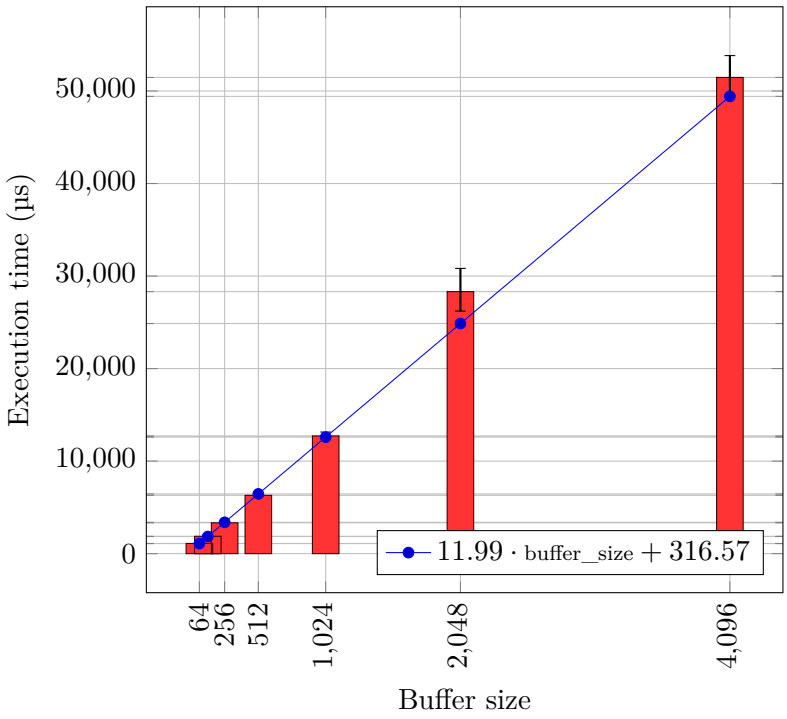


Figure 7.6.: Average execution time of an oscillator in function of the buffer size on a MacBook Pro with 16 GiB RAM and 3.10 GHz processor with macOS Sierra: powers of 2 from 64 to 4096 samples. We show the 95% confidence intervals and a linear regression of the average execution time.

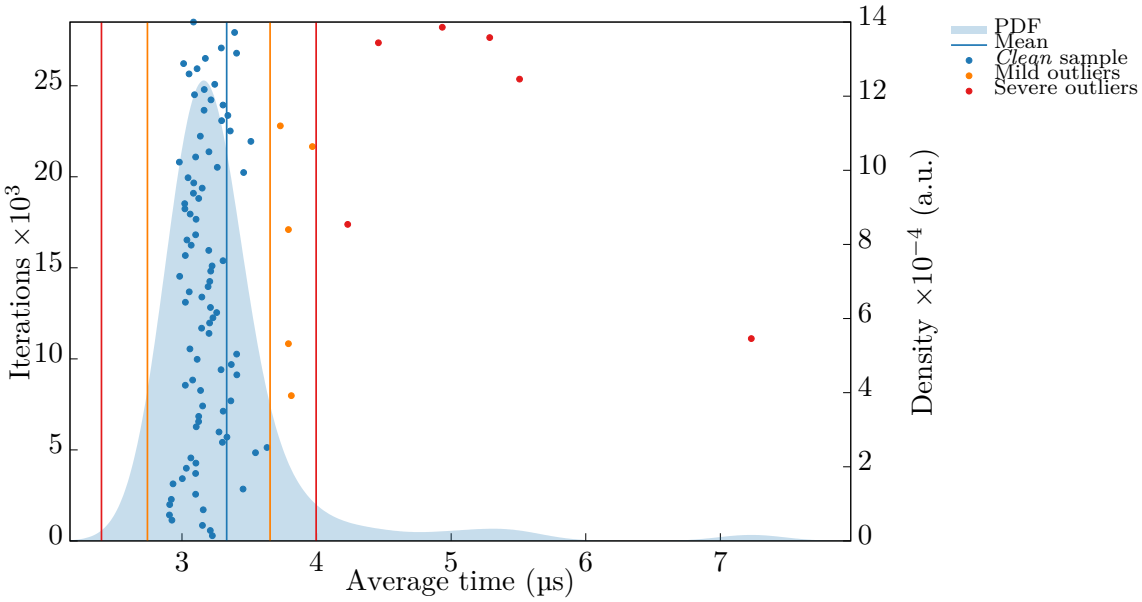


Figure 7.7.: Probability density function of the execution time of an oscillator with input buffer size of 256 samples. Outliers are probably due to the non real-time guarantees of the mainstream operating system on which the benchmark runs.

nodes process all their incoming samples and hence, that the complexity of their computations is at least linear. Therefore, we can bound execution times of v_k for $k \in \{2, \ldots, n-1\}$, the average execution time A_{v_k} and the worst execution time W_{v_K} for a downsampler with resampling factor $\frac{1}{r}$:

$$A'_{v_k} \leq \frac{1}{r} \times A_{v_k} \tag{7.15}$$

$$W'_{v_k} \leq \frac{1}{r} \times W_{v_k} \tag{7.16}$$

We can deduce a bound on the whole execution times of the degraded path between v_1 and v_n, $\pi' = v_1 \rightsquigarrow \ldots v'_1 \rightsquigarrow \cdots \rightsquigarrow v_k \rightsquigarrow \ldots v'N \rightsquigarrow \cdots \rightsquigarrow v_n$ where v'_1 and v'_n are respectively the downsampler and upsampler by r. We add the overhead of the resamplers and so we have:

$$A'_\pi \leq A_{v_1} + A_{v'_1} + \frac{1}{r} \times \sum_{k=2}^{n-1} A_{v_k} + A_{v'_n} + A_{v_n} \tag{7.17}$$

that's to say for the subpath $v_2 \rightsquigarrow \cdots \rightsquigarrow v_n$:

$$A'_\pi \leq A_{v_1} + A_{v'_1} + \frac{1}{r} A_{v_2 \rightsquigarrow \cdots \rightsquigarrow v_n} + A_{v'_n} + A_{v_n} \tag{7.18}$$

The execution time of graph G, on an uniprocessor, as the processing nodes are executed sequentially, is:

$$A_G = \sum_{v \in V_G} A_v \tag{7.19}$$

The execution time of the degraded graph G' is:

$$\begin{aligned} A_{G'} &= \sum_{v \in V_{G'}} A_v \\ &= \sum_{v \in V_{\text{non degraded}}} A_v + \sum_{v \in \pi', \pi' \in \Pi'} A_v - \sum_{v \in \pi'_1 \cap \pi'_2, \pi'_1, \pi'_2 \in \Pi'} A_v \end{aligned} \tag{7.20}$$

Π' is the set of degraded paths in G' and $V_{\text{non degraded}}$ is the set of non degraded nodes in the graph. The last term of the right member expresses that some paths can share nodes and that we do not want to count them several times.

Note that we take into account the execution times of the inserted nodes, so that the optimization is *overhead-aware*. For small graphs with audio processing nodes with an execution time of the same order of magnitude as the resamplers, the execution time of a degraded graph can be actually larger than the non-degraded one.

The case of graphs with nodes with the same average execution times Let G a graph with nodes $v_1, \ldots, v_n$. We assume that all nodes have the same average execution time A: for all i, $A_{v_i} = A$. We also assume that resamplers have the same average execution time as the nodes in graph G.

If the graph is a line, with only one path $v_1 \rightsquigarrow \cdots \rightsquigarrow v_2$ and that we resample only one subpath of that path, between node v_p and graph v_q with $p \leq q$, Equation 7.17 yields, for degraded graph G':

$$A_{G'} = \sum_{i \in \{1,\ldots,p-1\} \cup \{q+1,\ldots,n\}} A_{v_i} + A_{v_d} + A_{v_u} + \frac{1}{r} \sum_{i \in \{p,\ldots,q\}} A_{v_i} \tag{7.21}$$

where v_d and v_u are respectively the downsampler and the upsampler, A_{v_d} and A_{v_u} their average execution times, with resampling rate $\frac{1}{r} < 1$ and $r \in \mathbb{N} \setminus \{0\}$.

We want:

$$\begin{aligned}
A_{G'} < A_G &\iff A_{v_d} + A_{v_u} + \frac{1}{r} \sum_{i \in \{p,\ldots,q\}} A_{v_i} < \sum_{i \in \{p,\ldots,q\}} A_{v_i} \\
&\iff r(A_{v_d} + A_{v_u}) < (r-1) \sum_{i \in \{p,\ldots,q\}} A_{v_i} \text{ as } r \in \mathbb{N}^* \\
&\iff 2rA < (r-1)(q-p+1)A \\
&\iff \frac{2r}{r-1} < q-p+1
\end{aligned} \tag{7.22}$$

It means that we need to degrade at least $n_{\min} = \frac{2r}{r-1}$ nodes in one subpath to have a degraded graph faster than the non-degraded graph, when all nodes have the same average execution time. For $r = 2$ (*i.e.*, downsampling by 2), $n_{\min} = 4$. For graphs with such execution times and fewer than 6 nodes,[6] it is not worth it to degrade. As $n_{\min}$ decreases and $\lim_{r \to +\infty} \frac{2r}{r-1} = 2$, increasing the downsampling rate still requires at least 2 nodes minimum on the degraded subpath.

7.5. Experimental evaluation

In order to evaluate our theoretical models, we instantiate our models on a large number of graphs and compare the models. However, there are no reference benchmarks of audio graphs for IMSs. We decided to generate a huge number

[6] The source and the sink cannot be resampled, hence, a graph with 6 nodes has 4 degradable nodes.

of graphs, compute the theoretical execution time and quality of each graph, and then measure actual the execution time and quality by executing each audio graph. We then compare the theoretical values and the measured ones.

Audio graphs are executed faster-than-real-time and output a wav audio file. For the potential sources of graphs that are not synthetisers, we either use an audio file or generate white noise as inputs.

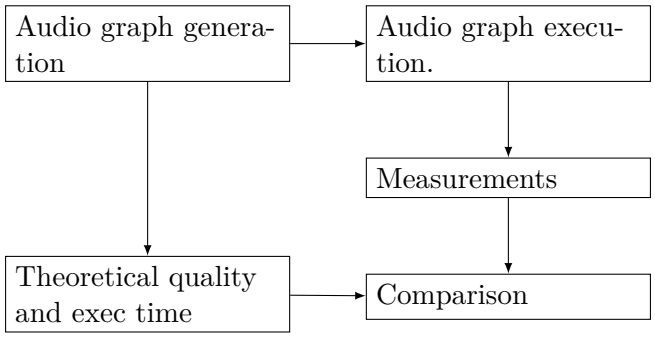


Figure 7.8.: The experimental setup to evaluate the models of quality and execution time.

7.5.1. Measuring execution time and quality

Execution time

To measure average execution times, we execute audio graphs on a large number of cycles using a simple prototype IMS we developed (See Appendix 3) and average the execution times of the cycles, after discarding the first cycles to take into account cache warming. The prototype IMS handles basic audio effects such as oscillators, modulators, and some effects imported from Faust, using the experimental Rust exporter of Faust, such as a transposer, a reverb (Zita_Reverb) or a guitar emulation.

Quality

To measure the quality of the degraded graphs, we compare the audio signal of the non-degraded graph G and the one of a degraded graph G'. Hence, we need a psychoacoustic distance between two signals. To generate new audio effects given audio examples in [San+18] , the authors compare the output x of potential audio effects with the example signal y, by computing a Fourier transform of ts time slices of the input signal and then taking the Euclidian

distance d of the first largest p peak frequency bins at time slice t:

$$\sum_{t=0}^{ts}\sum_{i=0}^{p} d\left(\text{bin}(i, \text{FFT}(x)[t]), \text{bin}(i, \text{FFT}(y)[t])\right) \tag{7.23}$$

The compared signals must be temporally-aligned.

Our distance aims at better quantifying how some frequencies are less perceived than other ones. Given the spectrum of each signal, we compute their constant-Q transform [Bro91], as we are interested in music signals. We then use psychoacoustics curves such as A-weighting [Moo12] or ITU-468 [Ass86] to account for the limited hearing range of human beings (up to 20 kHz) and that the perceived loudness of sound depends on the frequency. For the actual experiments, we have used A-weighting, which is both used for noise and pure sounds and so is more suitable for musical contents. Finally, we compute the L_2 norm between the two resulting spectra x_G and $x_{G'}$ and normalize it so that a distance of 0 leads to a quality of 1 and, of $+\infty$, a quality of 0.

$$q_{G'}^{\text{mes}} = \exp(-\frac{\|x_G - x_{G'}\|}{\text{nb\_bins}}) \tag{7.24}$$

where nb_bins is the number of spectral bins used in the constant-Q transform.

Equal-loudness contour ISO 226. Two sine waves of different frequencies have the same loudness if they are perceived as equally loud by a young non-hearing-impaired listener. Equal-loudness contours were first measured by Fletcher and Munson [FM33]. The international standard ISO 226:2003 [Suz+03] defines more precise loudness-equal contours, as shown on Figure 7.9. An equal-loudness curve defines a level of *phon*, the measure of loudness.

A-weighting. *A-weighting* is based on the set of *equal-loudness contours* measured by Fletcher and Munson and is in widespread use to measure noise levels. It is supposed to be used for low-level sounds (at 40 phon). The weighting function $A(f)$, in dB units, which must be added to the dB spectrum, is defined by, for frequency f:

$$R_A(f) = \frac{12194^2 \times f^4}{(f^2 + 20.6^2)\sqrt{(f^2 + 107.7^2)(f^2 + 737.9^2)}(f^2 + 12194^2)} \tag{7.25}$$

$$A(f) = 20\log_{10}(R_A(f)) + 2 \tag{7.26}$$

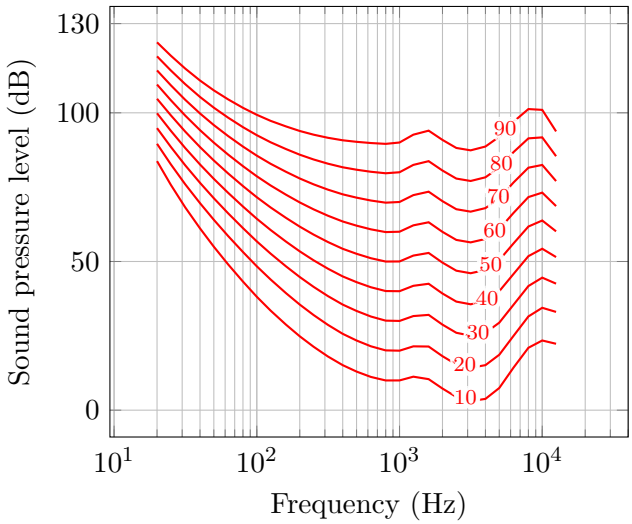


Figure 7.9.: ISO 226-2003 equal-loudness contours, in *phon*, with frequency, in Hz. For low frequencies, the sound pressure level must be higher to be heard as loud as sounds with a mid-range frequency.

ITU 468. It is a standard [Ass86] from the International Telecommunication Union that aims at measuring random audio noise. Contrary to A-weighting, which was first developed for pure tones, and then used for noise, ITU 468 has been specifically dedicated to noise measurement. It also includes a correction for tone bursts, which are perceived differently than long noises. The amplitude response $ITU(f)$ for a given frequency f is:

$$R_{ITU}(f) = \frac{1.246332637532143 \cdot 10^{-4}\, f}{\sqrt{(h_1(f))^2 + (h_2(f))^2}} \tag{7.27}$$

$$ITU(f) = 18.2 + 20 \log_{10}\left(R_{\mathrm{ITU}}(f)\right) \tag{7.28}$$

where

$$\begin{aligned} h_1(f) &= -4.737338981378384 \cdot 10^{-24}\, f^6 + 2.043828333606125 \cdot 10^{-15}\, f^4 \\ &\quad - 1.363894795463638 \cdot 10^{-7}\, f^2 + 1 \\ h_2(f) &= 1.306612257412824 \cdot 10^{-19}\, f^5 - 2.118150887518656 \cdot 10^{-11}\, f^3 \\ &\quad + 5.559488023498642 \cdot 10^{-4}\, f \end{aligned}$$

7.5.2. Comparing models and measurements

Graph generation

The graph generation is undertaken in two phases: first, generating the *structure* of the graph, and then picking an *actual audio processor* for each vertex in a node dictionary.

Exhaustive generation for a given number of nodes. We enumerate all the non-labelled weakly-connected directed acyclic graphs (WCDAGs) with n vertices. Non-labelled entails that $a \to b$ and $b \to a$ are isomorphic, *i.e.* are the *same* graphs. Given the set of vertices $V = \{0, \dots, n-1\}$, we undertake the following steps.

1. Compute the set of all the possible directed edges $\mathcal{E}$ between distinct vertices in one direction. Edges are all the possible pairs of distinct elements of the set of vertices V. The function pair that computes the set of all such possible pairs from a list of elements can be defined inductively, as $\text{pairs}(a_k, \dots, a_n) = \bigcup_{i \in \{k+1,\dots,n\}} \{(a_k, a_i)\} \cup \text{pairs}(a_{k+1}, \dots, a_n)$ It will entail acyclicity, as we cannot create new edges that would go a an already used vertex.
2. Compute $\mathcal{P}(\mathcal{E})$.
3. In a connected graph with n vertices, there are at least $n-1$ edges (*i.e.* chain graph). So we keep only subsets with $n-1$ edges of $\mathcal{P}(\mathcal{E})$, or more, in our admissible set of set of edges, E.
4. Build the set D of DAGs from E, one graph per subset.
5. Filter D to remove non weakly-connected graphs, by picking a node and then traverse the undirected version of the graph and counting the vertices. If there are the same numbers as the total number of vertices in the graph, it is weakly connected.

The set of all possible edges from n nodes has size:

$$(n-1) + n - 2 + \dots 1 = \sum_{k=1}^{n-1} k = \mathcal{O}(n^2) \tag{7.29}$$

Thus the powerset has size $\mathcal{O}(2^{n^2})$. The operations that follow the power generation reduce the number of graphs, so that upper bound remains correct.

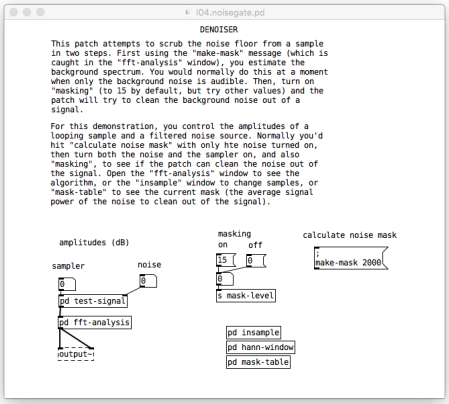
Random generation. As the increase in the number of graphs is over-exponential, it becomes untractable when $n > 6$ in practice. For $n = 7$ for instance, there are 3 781 503 possible DAGs. Hence, for larger number of nodes, we randomly generate graphs. There are various random directed-acyclic-graph generation models [CSH19]: layer-by-layer methods, random generation of triangular matrices, uniform random generation using a recursive/counting approach or Markov-chain Monte Carlo, or derivation from randomly generated orders. We have chosen to use an adaptation to directed acyclic graphs of a simple and flexible model, the Erdős–Rényil [ER60] random graph model, in the family of methods undertaking random generation through generation of triangular matrices. In this model, a graph can be chosen uniformly at random from the graphs with n nodes and M edges, or with n nodes and a given probability p of having an edge between two nodes. In that case, all graphs with n nodes and M edges have probability $p^M(1-p)^{\binom{n}{2}-M}$. The larger p, the more edges. The average number of edges is $\binom{n}{2}p$.

We want to get weakly connected DAGs. According to Erdős and Rény, if $np > 1$, the generated graphs have one large component, and other components with no more than $O(\ln(n))$ vertices. We proceed to a weakly-connected component decomposition of the graph and keep the largest component. In addition, we can select $p > \frac{(1+\epsilon)\ln(n)}{n}$ for $\epsilon \to 0$, and the graphs will almost surely (in the probabilistic meaning) be connected.

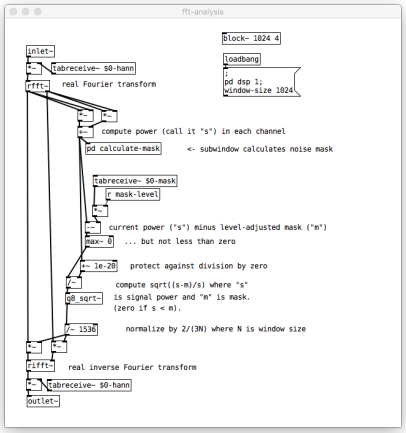
From Puredata patches. Audio graphs tend to exhibit a particular structure: few incoming and outgoing edges per node, which leads to long chains in the audio graph, with a few nodes with more inputs that typically mix signals, as shown on Table 7.2. To take into account this structure, we parsed all the Puredata patches of its tutorial and examples,[7] *i.e.* 133 graphs. Puredata patches can be nested: there can be subpatches inside a Puredata patch, as shown on Figure 7.10. Inputs and outputs of the subpatches inside the subpatch are denoted with special boxes: `inlet`, `outlet` for control, `inlet~`, `outlet~` for audio signals, `tabsend~` and `tabreceive~` for arrays. We flatten the nested structure into one big graph and keep the biggest connected component.

Node database. We maintain a database of possible audio processing nodes, with their estimated execution time, their numbers of input ports and output ports, and their possible control parameters. The database is a file with nodes described with our custom audiograph format (see Appendix 1).

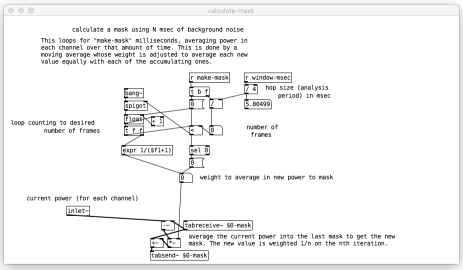
[7] They are included in the standard distribution of Puredata.



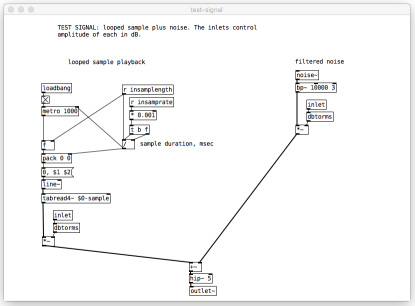
(a) Main patch



(b) `fft-analysis` subpatch.



(c) `calculate-mask` subpatch.



(d) `test-signal` subpatch.

Figure 7.10.: A denoiser patch with multiple subpatches. `fft-analysis` (b) and `test-signal` (d) are subpatches of the main patch (a), whereas `calculate-mask` (c) is a subpatch of `fft-analysis` (b).

$\overline{i(v)_{v\in G,G\in\mathcal{G}}}$	$\overline{o(v)_{v\in G,G\in\mathcal{G}}}$	$\overline{\max_{v\in G} i(v)}$	$\overline{\max_{v\in G} o(v)}$	$\overline{n_{\text{edges}}}$
1.1356	1.1356	2.2857	2.5274	16.0989
$\overline{n_{\text{vertices}}}$	$\max\{n_{\text{edges}}\}$	$\max\{n_{\text{vertices}}\}$	$\overline{d_G}$	$\max\{d_G\}$
14.1758	61	43	5.8571	13

Table 7.2.: Statistics on in and out degrees, number of edges, of nodes, diameter of graphs extracted from Puredata patches. $\overline{i(v)_{v\in G,G\in\mathcal{G}}}$ is the average in-degree; $\overline{o(v)_{v\in G,G\in\mathcal{G}}}$ is the average out-degree; $\overline{\max_{v\in G} i(v)}$ is the average max in-degree; $\overline{\max_{v\in G} o(v)}$ is the average max out-degree. In a directed graph, and due to the handshaking lemma, $\overline{i(v)_{v\in G,G\in\mathcal{G}}} = \overline{o(v)_{v\in G,G\in\mathcal{G}}} = \overline{\frac{n_{\text{ i.e. edges}}}{n_{\text{vertices}}}}$.

All the parameters of a node, except the number of input and output ports and the kind of processing, can be either given a value, or annotated with a range, or with a finite set. The annotation also indicates if we want to randomly pick a value among the possible values or generate a graph per value (for a finite set) or with a sampling of the range. A range is notated `[n,m]` and a finite set $\{\mathtt{e}_1, \ldots, \mathtt{e}_n\}$. Picking a value is notated with `@pick`, and enumerating or sampling is notated with `@all`. In Code 7.1, we show the definition of a modulator node that can take several frequencies as parameters and a volume in the range $[0, 1]$.

```
n1 =
{
    kind : "mod",
    in: 1,
    out : 1,
    freq : "@pick{20,440,1000,2500,6000}",
    volume : "@pick[0,1]",
    wcet: 3.5,
};
```

Code 7.1: A modulator node in the database of nodes. The frequency and the volume can take several values. For frequency, the values are taken from a finite set and, for volume, from a range.

From the graph structure to the audio graph. Given the structure of the graph, for each vertex in it, we can pick (or generate all possible versions of)

nodes with the same number of input ports as incoming edges and a number of output ports between 1 and the number of outgoing edges. There may be less output ports than outgoing edges because several outgoing edges can share one output port, as as shown on Figure 7.11.

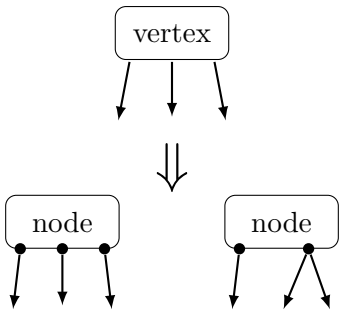


Figure 7.11.: Port sharing entails that a vertex can be replaced by a node with a smaller number of output ports. At the top, it is a vertex with 3 outgoing edges. At the bottom, we show two possible actual nodes generated from this vertex, one with 3 output ports, and one with 2 output ports and the second port shared by two outgoing edges.

Comparing rankings

After generating the audio graphs, the models of execution time in Section 7.4 and quality in Section 7.3 and the measurements in Section 7.5.1 make it possible to compute two rankings of the set of audio graphs. The theoretical one, σ_{th}, and the measured one, σ_{mes}, are two permutations of the same set and we aim at computing how far from each other those two permutations are.

Distances. A first way to compare permutations is to use a distance, such as Kendall's τ distance, Spearman's footrule distance, Cayley's distance [FV86], which counts the minimum number of transpositions that transforms one permutation into the other, or Ulam distance [AD99], which counts the number of *delete, shift, insert* operations.[8] We do not detail Kendall's τ distance and Spearman's footrule distance but rather the correlations they inspire.

Rank correlation. *Rank correlation* measures the relationship between rankings of the same size, and the *rank correlation coefficient* measures the sim-

[8]It can be described as an edit-distance on non-repetitive strings.

ilarity between two rankings. If the two rankings are in the same order, the correlation coefficient is 1, *i.e.*, the function that transforms the values of one of the rankings into the values of the other ranking is monotonic. If it is -1, the order is reversed. If the coefficient is 0, the rankings are completely independent.

Kendall's τ correlation coefficient [Ken48]. It is linked to the number of inversions[9] needed to transform one ordering into the other one. For pairs of observations $((x_i, y_i))_{i\in\{1,\dots,n\}}$, τ is defined as:

$$\tau = \frac{N_c - N_d}{n(n-1)/2} \tag{7.30}$$

where N_c is the number of concordant pairs, defined by:

$$N_c = |\{(i,j) \,|\, (x_i > x_j \wedge y_i > y_j) \vee (x_i < x_j \wedge y_i < y_i), i, j \in \{1,\dots,n\}\}|$$

and N_d is the number of discordant pairs, with:

$$N_d = |\{(i,j) \,|\, (x_i > x_j \wedge y_i < y_j) \vee (x_i < x_j \wedge y_i > y_i), i, j \in \{1,\dots,n\}\}|$$

Spearman's ρ correlation coefficient [Dan+78]. It is linked to the distance in positions of a same graph in the two orderings. For pairs of observations $((x_i, y_i))_{i\in\{1,\dots,n\}}$ such that all n ranks are distinct integers, ρ is defined by:

$$\rho = 1 - \frac{6\sum_{i=1}^{n} d_i^2}{n(n^2-1)} \tag{7.31}$$

where d_i is the difference between the two ranks x_i and y_i of each observation.

Other measures

We also collect the worst and best execution times, the worst qualities, and how many versions are faster than their non-degraded graph.

7.5.3. Results

The resamplers in use for these experiments are the linear resamplers, and we only resample by 2. The graphs are executed with an initial buffer size of 512 samples and a sample rate of 44 100 Hz, for 10000 cycles, which amounts

[9] If we have two elements at positions i and j, $i < j$ in an ordering, σ is an inversion if $\sigma(i) > \sigma(j)$ and only those elements are swapped.

to about 116 s of audio. We use two different databases of nodes: one where the execution time of nodes is the same order of magnitude as the one of a resampler, and one where the execution time of nodes is one order of magnitude smaller than the execution time of a resampler, in order to illustrate Equation 7.22. We use the heuristic quality measure of Section 7.3.1.

On one graph

We present the results for the 5-node graph of Figure 7.12a, which has 3 possible degraded versions, as shown on Figure 7.12. The maximum number of inserted resamplers is 3, and the minimum number is 2. The execution times and qualities are shown on Figure 7.13. The model, on Figure 7.13a, somewhat accurately mirrors the measured execution times on Figure 7.13b. The non-degraded graph has the best quality, and here, it also has the best execution time, because the nodes of the graph are basic nodes with short ACET, and so the decrease of execution time of the nodes on resampled paths is squandered by the overhead of adding resamplers.

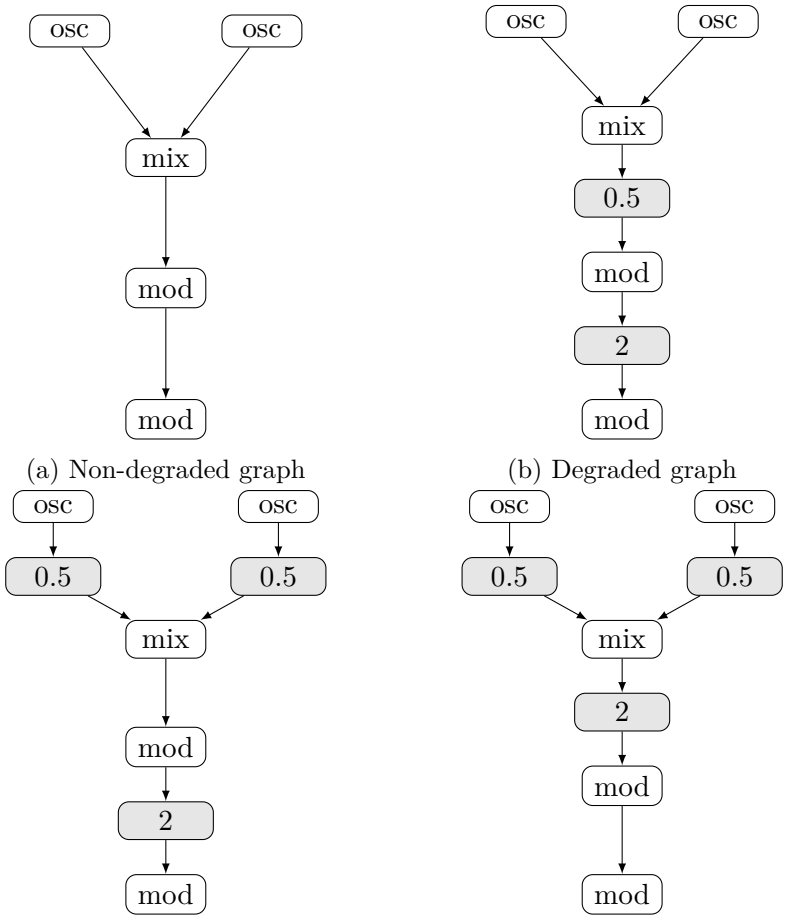
Exhaustive enumeration

We perform an exhaustive enumeration of all the 838 graphs with 5 nodes using the dictionary of basic nodes. The average number of versions of a non degraded graph (including the non-degraded graph) is 3.658 ± 2.067 and there are 57 graphs without degraded versions. The rank correlations between the model data and the measures for quality and cost are shown on Figure 7.14 with histograms. Most of the correlations are close to 1. When the non-degraded graph has no degraded version, the correlations are not defined and that is why the cumulative population on the histograms is less than 838. Only 33 degraded graphs (*i.e*, 2% of the graphs) are faster to execute than their non-degraded versions. Indeed, the overhead of the resamplers here is too much compared to what is gained by halving the buffer sizes on some branches.

On the contrary, when using a dictionary of slow nodes, 275 graphs, *i.e.*, 33%, have at least one degraded version faster than the non degraded version.

Large random graphs

We generate 100 random graphs with the dictionary of basic nodes with up to 10 nodes using the Erdős–Rényil model (see Section 7.5.2) with probability 0.3, which ensures that the graphs have one big component but not too many edges. If there are too many degraded versions, which would be too long to



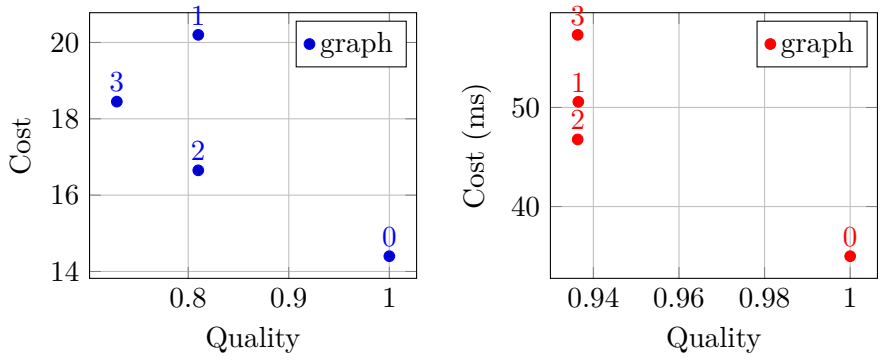
(a) Non-degraded graph

(b) Degraded graph

(c) Degraded graph

(d) Degraded graph

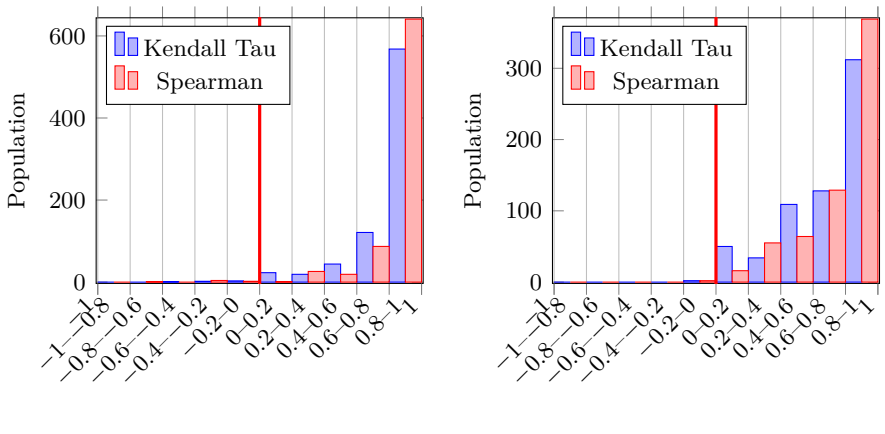
Figure 7.12.: The degraded graph 7.12a and all its degraded versions 7.12b, 7.12c and 7.12d. The resamplers are filled in light grey and annotated with their resampling ratio.



(a) Cost and quality model.

(b) Measured cost and quality.

Figure 7.13.: Execution time and quality for the graphs obtained from the graph 7.12a, according to the models and according to the measurements. Graph 0 is the non-degraded graph. Graph 1 is (7.12b); graph 2 is (7.12c); graph 3 is (7.12d).



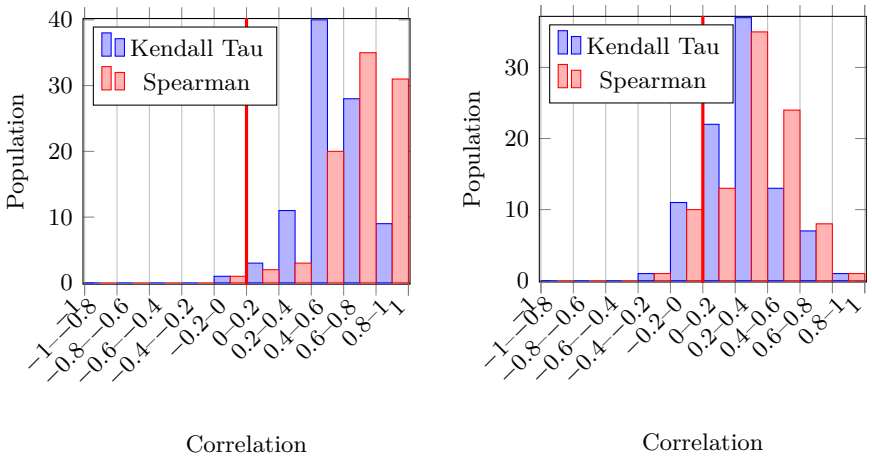
(a) Correlations between the cost model and the measured cost

(b) Correlations between the quality model and the measured quality

Figure 7.14.: Histogram of correlations for cost in 7.14a and quality in 7.14b for exhaustive enumeration of 5-node graphs, using Kendall Tau and Spearman correlations.

test, we only generate a subsample of size 65.[10] The histograms of correlations are shown of Figure 7.15. They show that the model execution time and the measured execution times are relatively well-correlated. For quality, the results are less good, as there are lots of correlations close to 0, but mainly strictly positive. The average number of versions is 23.26. As we also use basic nodes here, there are only 15 non-degraded graphs for which their degraded version is faster, whereas with slow nodes, there are 45 such graphs.

On Figure 7.16, we show a plot of the slowest version of a non-degraded graph against the fastest version. We use two different linear regression methods robust to outliers, Siegel [Sie82] and Theil-Sen [Sen68; The92]. The first one yields a 1.65 coefficient and the second one 2.34 , with a 90 % confidence interval between 1.54 and 3.07. Those values are close to 2, the value we expect when fully downsampling the audio graph by 2. The fastest version is in practice slower than the slowest divided by 2 as the sources and sinks are not resampled and the resamplers add some overhead.



(a) Correlations between the cost model and the measured cost

(b) Correlations between the quality model and the measured quality

Figure 7.15.: Histogram of correlations for cost in (7.15a) and quality in (7.15b) for 10-node random graphs, using Kendall Tau and Spearman correlations.

[10] The non-degraded graph plus a not too large power of 2, which comes from bounding the number of degraded graphs by the size of the powerset.

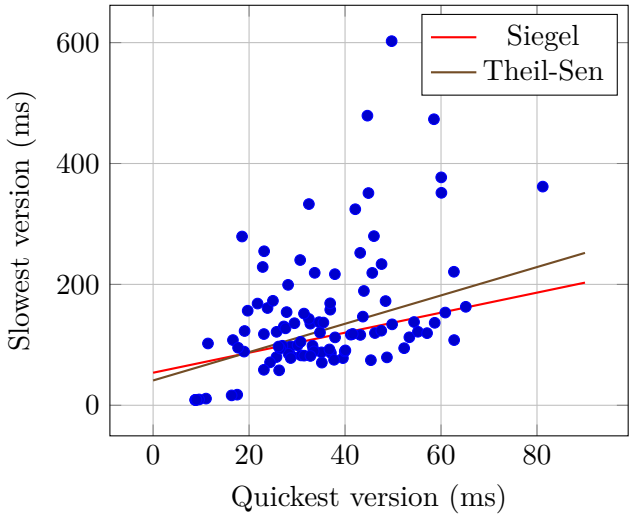


Figure 7.16.: Quickest and slowest versions for each non-degraded graph. We perform two linear regressions using methods robust to outliers, Siegel estimator [Sie82] and Theil-Sen estimator [Sen68; The92].

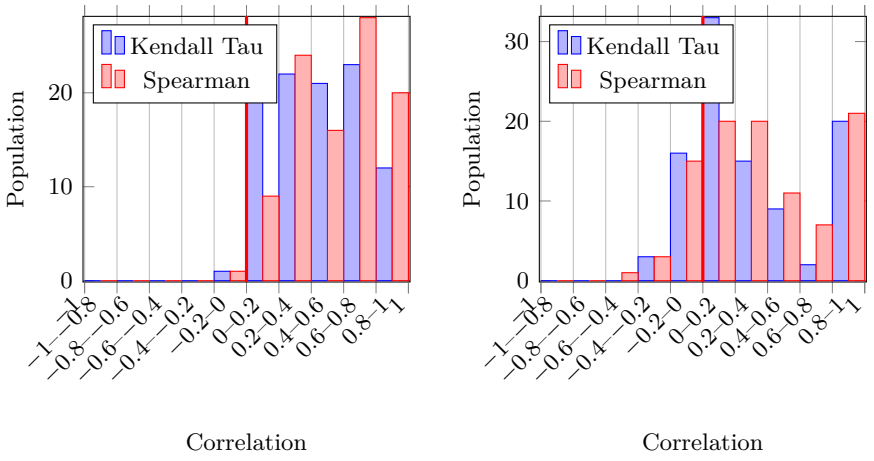
With graphs from Puredata

We use 133 Puredata patches from the Puredata tutorials as the structure for the generated audio graphs and nodes from the dictionary of basic nodes. We only keep graphs with 4 nodes or more, which amounts to 102 graphs. In average, there are 38.88 degraded versions[11] for a non-degraded graph. 48 graphs have degraded versions that are faster than their non-degraded graph. On average, 8 degraded versions, *i.e.*, 18.7% of the degraded versions, are faster than the non-degraded graph. The correlations between models and measures are shown on Figure 7.17.

Discussion

We evaluated the models and the algorithm on audio graphs that were exhaustively enumerated for graphs with few nodes, randomly generated for larger graphs, and generated from Puredata patches at last. The execution time model is quite accurate. The quality model is well correlated for small graphs, has to be improved on large random graphs and shows promising results for graphs generated from real Puredata patches.

[11] We remind that we limit the number of degraded versions to 65 maximum when we do random sampling because otherwise, it would take too long to test. See Section 7.4.2.



(a) Correlations between the cost model and the measured cost

(b) Correlations between the quality model and the measured quality

Figure 7.17.: Histogram of correlations for cost in 7.17a and quality in 7.17b for graphs generated from Puredata patches, with at least 4 nodes, using Kendall Tau and Spearman correlations.

We observe that large graphs without too many ramifications and with nodes with execution times at least an order of magnitude higher than the execution time of a resampler take advantage the most of the resampling optimization. Actual audiographs from IMSs, such as the ones from Puredata, have actually these characteristics. The heuristic quality measure is arbitrary. We could organize listening tests, but it would be impractical considering the huge size of the set of possible graphs and possible degraded graphs. We should rather use a more precise model based on measurements as proposed in Section 7.3.1.

8. Adaptive overhead-aware scheduling of audio graphs by resampling

The offline optimization techniques presented in Chapter 7 can be adapted to an *online* setup, where an audio graph executes in real time and the degradations are computed or applied on the fly, when the processor becomes overloaded and a deadline miss is predicted. The optimization computations need to be quick, as the short deadlines do not let enough time to do an exhaustive exploration for instance. We present techniques that are used in real-time systems for quality adaptation, and especially in multimedia systems in Section 8.1. In Section 8.2, we show how to adapt the degrading strategies can be used in the real-time case. In Section 8.3, we explain how an audio graph is executed during one cycle and how we estimate if a deadline is likely to be missed. The results are presented and discussed in Section 8.4. This work originates from our article [Don18].

8.1. Real-time systems and adaptation

Adaptive scheduling for hard or soft real-time systems has been dealt with by removing some tasks or by degrading them, like the *approximate programming paradigm* (see Section 7.1) or *mixed criticality*. Another way of dealing with tasks competing for resources is *resource reservation*.

In multimedia systems, a basic strategy [Liu+91] related to mixed criticality consists of dividing tasks between a mandatory and an optional part, which can be discarded in case of a processor overload. In [SN13], a task has two versions: P_1, with a good quality of service but that takes an unknown duration to complete, and P_2, which has a bad quality of service but has a known execution time. Two strategies are used to choose between the two versions: *first chance* and *last chance*. For *first chance*, as soon as a deadline miss is likely to happen for P_1, P_1 is aborted in favour of P_2. In *last chance*, P_2 is started first, and if

there is enough time before the deadline, P_1 is executed while the results of P_2 are kept in case of P_1 exceeding the deadline. Instead of switching to another version of the task, with a worse quality but a predictable execution time, in [SN13], some tasks can be selected to be totally aborted. Tasks are given an *importance* value. In case of overload, tasks with the least *importance* are killed until there is no more overload. To handle the termination of such tasks gracefully, tasks have two modes, a normal mode and a *survival* mode, with an abortion or an adjournment property, *e.g.*, freeing memory, saving partial computations to carry on later.

Real-time scheduling with this strategy does not entail too much overhead but does not handle dependencies between tasks, in particular for quality estimation.

Some mixed criticality approaches address graph-based tasks for mixed-criticality systems, such as in [EY16]. However, the dependencies between tasks are functional dependencies: all tasks in a graph have the same criticality, but it is possible to switch to other graphs with another criticality. Criticality levels are not like *qualities* that would depend on the topology of the graph.

In resource reservation [Årz+11], part of the processor computation resources is reserved to some tasks. For instance, in the case of multimedia, video tasks and audio tasks could have different reservations, as audio tasks are *more real-time* than video ones. Nevertheless, resource reservation requires a dedicated scheduler, which is not often present in mainstream operating systems. It does not deal with similar tasks linked by dependencies, such as in a graph, either.

The (m, k)-firm model [HR95] deals with qualit of service (QoS) by stating that at least m out of any k consecutive tasks must meet their deadlines. This model is used for streams and is well suited for multimedia, for instance for video, where some frames can be discarded without losing too much quality. However, it does not apply in our case, as we do not consider independent streams but a graph of audio streams, the quality of which depends on their position in the audio graph.

Dynamic voltage scaling (DVS) for scheduling modifies the frequency of the processor(s) and makes it possible to optimize the energy consumption. In [But02], in case of a change in the processor speed, the period of tasks in the system is reduced using the elastic approach [But+02], where the utilization of tasks is seen as a linear spring system where a force is applied. In [LZ09], it is used in the context of high-performance computing systems for applications with tasks with precedence constraints. However, the energy consumption of a task depends only on the frequency of the processor when the task is executed. It means that the quality is reduced in the same way for all the tasks and is

global. In an audio graph, the quality depends on the quality of the inputs and so is local.

8.2. Resampling strategies suitable for real-time scheduling

In a real-time context, we want to detect that a graph is likely to miss its delivery deadline, *i.e.*, the processor is overloaded, and to degrade it while keeping the quality as high as possible. We also have to take into account the *time overhead* to find a worthy degraded version, and hence we must aim at fast strategies. We present two strategies to adapt the graph:

- Use pre-computed degraded versions to swap the graph with;
- Degrade all or part of the remaining nodes to execute in a cycle with a heuristic.

Using pre-computed degraded versions

In Chapter 7, we presented strategies to enumerate and choose degraded versions of a graph based on the quality or the execution time of the graph. We want to decrease the number of graphs. As the execution times result from measurements, it is likely that all graphs have different execution times, so we first cluster the graphs by execution times. For that, we use a 1-dimensional version of k-means, called *ckmeans* [WS11], where the number of clusters k can be estimated within a provided range, with a complexity of $O(kn\log(n))$ for n versions to cluster (and $O(nk)$ if they are already sorted by execution times).

Then we can compute the Pareto front (see Section 7.2.2) of execution times and quality of the clusters, *i.e.* for each cluster, we keep the graph with the best quality, as pictured in Figure 8.1. We finally get a set $\mathcal{G}$ of representative degraded graphs.

Swapping graphs. As we swap whole graphs, when we detect a possible deadline miss, we can only use a degraded version starting from the next cycle. An issue is that the average execution times A we get are measured for a given buffer size m and a given system. We first do a rough approximation of the execution time for another buffer size m' as $A' = \frac{m'}{m} \times A$. We then pick the graph that has the maximum quality while respecting the deadline d *i.e.* $G_{\text{pick}} = \text{argmax}_{G\in\mathcal{G}} \{q_G \,|\, A_G < d\}$. If the set of such graphs is empty, we pick the graph with the minimum execution time, even though its execution time

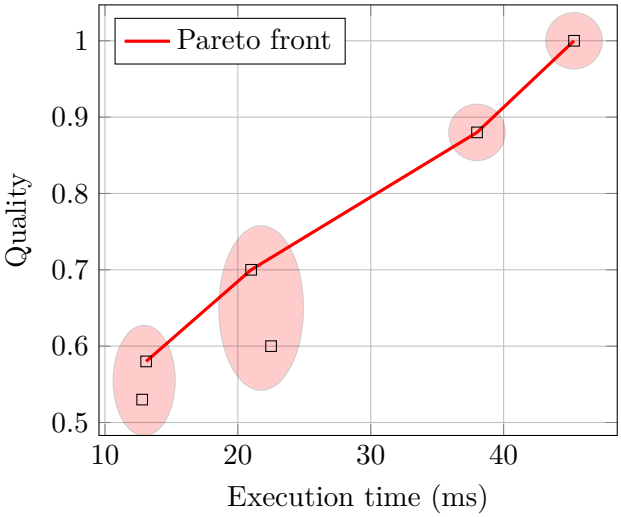


Figure 8.1.: Pairs of execution time and quality for degraded graphs and their non-degraded graph. A red ellipsis is a cluster of graphs where the clustering is done on the execution time axis. In each cluster, we pick the best quality graph, which gives us an approximate Pareto front with four graphs.

exceeds the deadline. If the estimation of the execution time of the degraded graph was not accurate, and we still miss the deadline, we also pick the graph with minimum execution time for the next cycles.

Using heuristics

We want to be able to degrade the graph in the middle of an audio cycle. Yet, it means that we do not want to and cannot degrade the nodes that have already executed. We design two heuristics:

- *progressive strategy*, with a backward traversal from the sinks, inspired by the offline heuristics in Section 7.2.4;
- *exhaustive strategy*, which degrades all remaining nodes.

In the *progressive strategy*, we start from one of the output nodes, and we traverse the graph backwards and see how inserting a downsampling node on a path going to this output node would change the estimated remaining execution time, until the estimation of remaining execution time is lower than the remaining time before the deadline, as shown in Figure 8.2. Other branches can be explored if it is not enough. Then the downsampling and upsampling nodes are inserted on the paths chosen to be resampled.

However, exploring the branches is $O(n)$ in the number of nodes of the graph hence is costly and can create too much overhead. The *exhaustive strategy* addresses that by not exploring the graph but rather degrading brutally all the remaining nodes, by downsampling all inputs to remaining nodes if they are not already downsampled. This strategy has a complexity of $O(1)$.

For each strategy, we estimate the execution time of the remaining node if using the degraded version. If that estimated degraded execution time entails a deadline miss, we consider it is not worth it degrading.

Transient and permanent overload

Informally, a *transient* overload of the processor happens when the processor is overloaded for less than k cycles, whereas a *permanent* overload occurs when it is overloaded for more than k cycles. Here, k is fixed in advance manually.

If a possible deadline miss is detected during the execution of the graph, it is a sign of a possible transient overload. In that case, we want to degrade the graph while executing it, at the middle of a cycle. Therefore, we use the heuristics. On the contrary, in case of permanent overload, we do not need to degrade during the execution of an audio cycle, hence we rather use a pre-computed degraded version.

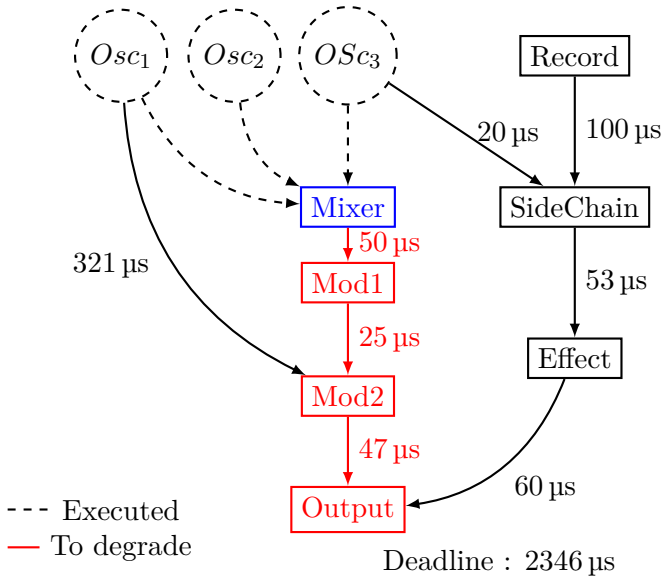


Figure 8.2.: The progressive strategy in action. The dashed nodes have already been executed. It is the turn of the Mixer node (in blue) to be executed, but the estimated remaining time exceeds the remaining allocated time budget and would entail a deadline miss. We traverse backwards from the Output and find out that degrading the red branch is enough not to miss the deadline.

8.3. Execution of the audio graph and prediction of deadline misses

A schedule of the nodes to execute has been computed using a topological sort. As shown in Algorithm 5, we execute the nodes in order of the schedule; we check before executing each node in the schedule if the elapsed time plus the estimated remaining time, computed using the model in Section 7.4.2, does not exceed the remaining time budget, risking a deadline miss. The environment provides the relative deadline. If the estimated remaining computation time exceeds it, we use one of the heuristics in chooseNodes (see Algorithm 5). This function does not generate a new graph but rather tags the nodes to be degraded and especially the first and last nodes of each path to be degraded. Again, to prevent adding too much overhead, we do not insert full resampler nodes but rather directly downsample and upsample. After each node, we update our estimation of the average execution time of the node. At the end of a cycle, we send some monitoring information to a recorder thread, which saves it on the disk for further analysis. We made sure to use a lock-free queue to send to the monitoring thread not to disturb the real-time audio computations.

Execution time estimation We measure the execution time of each node of the audio graph as well as the time to copy data in the audio channels and the time used by a resampler for each cycle. We update the average of those execution times at the end of each cycle, using a numerically stable expression [Knu97] of the mean:

$$A_v(n) = A_v(n-1) + \frac{T_v(n) - A_v(n-1)}{n}$$

where $A_v(n)$ is the average execution time of node v computed up to cycle n and $T_v(n)$ is the execution time of the node at cycle n. Using the mean for the whole execution assumes that the execution time of the nodes does not change too much, but that what changes is the relative deadline of the audio callback. We could also use a moving average or an exponential moving average to take more into account transient changes of node execution times.

8.4. Results and discussion

There are no commercial benchmarks of audio graphs of IMSs nor any reference benchmarks. Besides, audio graphs of pieces for IMSs are usually not shared

Algorithm 5 Execution of the graph for one cycle, possibly starting degradation with the heuristics at the middle of the cycle. node is the function that performs the sound processing of node. "buffers" is a set of buffers used as input and output buffers. Heuristics compute the set of nodes to degrade by updating flags associated to the node in chooseNodes. node.firstToDegrade indicates that the current buffer must be downsampled before performing the node processing, and node.lastToDegrade, that the buffer after must be upsampled after the node processing.

Require: S a schedule, G an audio graph with associated execution times, d deadline
  expectedRemainingTime ←0
  buffers ←CallfromSoundcard
  while S is not empty **do**
    node ←pop_first(S)
    UPDATE(expectedRemainingTime)
    if expectedRemainingTime ≥ 0 **then**
      CHOOSENODES(G, d, expectedRemainingTime) ▷ Heuristics to find the nodes to degrade
    end if
    if node.firstToDegrade **then**
      DOWNSAMPLE(buffers)
    end if
    buffers ←NODE.PROCESS(buffers)
    if node.lastToDegrade **then**
      UPSAMPLE(buffers)
    end if
    update performanceCounters
  end while
  TOSOUNDCARD(buffers)

as open source, as the audio graph is considered by the composers to be a part of a score, which is sold.

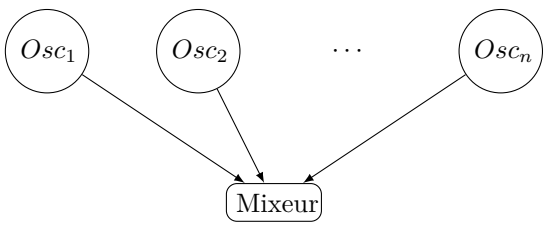
Hence, we use a real-time version of the basic IMS we have developed (see Appendix 3) to generate typical and random audio graphs to evaluate the heuristics, in a similar way of Chapter 7. We also generate graphs with pathological shapes, *i.e.*, chains and combs.

Setup. The online adaptive heuristics have been implemented in a prototype in Rust, which uses a Rust version of Portaudio [BB01a] for the audio callback. Resampling is performed with libsamplerate.[1] This is an open-source library that makes it possible to downsample down to a 256 ratio, and to upsample up to a 256 ratio. The sampling rate can be changed in real time. The library provides five resamplers, classified by decreasing order of quality: best, medium, faster sync resamplers (as in [SG84]), zero-order hold resampler, and a linear resampler. Experiments for random graphs have been run on a Mac Book Pro with an Intel Core i7 processor at 3.1 GHz with 16 GiB RAM. Experiments for the pathological graphs were run on a MacBook Pro with an Intel Core i7 processor at 2.6 GHz, with 8 GiB.

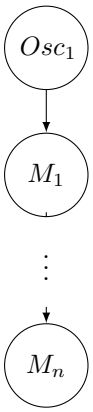
Experiments. We evaluated the two heuristics, the total and the progressive strategies. For the progressive strategy, we limit the backward traversal to only one branch here. For each adaptive strategy, total and progressive, we randomly generate audio graphs with a fixed number of nodes, n, using four types of simple audio processors: a sine oscillator, a sine modulator, a low-pass filter and a mixer. An oscillator is a source, and a modulator has one input and one output, as the low-pass filter. A mixer has an arbitrary number of inputs and one output. We generated graphs with 10, 400 and 1000 nodes. We chose only two sample rates for our finite set of sample rates: the sample rate of the soundcard, and half of it. The buffer size is 64 and the signal is mono. Each audio graph is executed for 500 cycles, *i.e.*, for 500 invocations of the audio callback. We chose a fixed number of cycles and not a fixed duration, as the time budget allocated to the audio callback can change.

We also evaluated the strategies on pathological graphs, with specific shapes: graphs with only one path, as in Figure 8.3b, and comb-shaped graphs, with only one-node paths from input to output, as in Figure 8.3a. In the first case, inserting a downsampler decreases all the following nodes. In the second case, we have to insert resamplers for all remaining paths, and so it exhibits the most possible overhead.

[1] `http://www.mega-nerd.com/SRC/`



(a) Comb-shaped audio graphs with n oscillators.



(b) Chain audio graph with one oscillator followed by n modulators.

Figure 8.3.: Two pathological kinds of graphs used for the online experiments.

Results. The results on the experiment with random graphs are shown in Table 8.1. From 400 nodes, the processor starts to be overloaded and the graph is degraded. In case degradation is decided, the total strategy performs worse on average than the progressive strategy: the time budget is lower and even negative for the total strategy. The total strategy brutally degrades all the remaining nodes to be executed. It inserts many more resamplers than the progressive strategy, about 5200 against 438, nodes for instance.

On the contrary, the progressive strategy is smarter but has much more variability in the time budget: it arbitrarily picks one parent to degrade when traversing the audio graph backwards. For 1000 nodes, the average deadline of the audio callback is 18 661 µs and the duration of choosing nodes, 6813 µs, so about 36% of the allocated callback duration.

For the pathological graphs, we show the results for the chain graphs and the exhaustive strategy on Figure 8.4. We tested a chain with 2000 modulators and another one with 3000. We also compared it with a comb graph with 2000 oscillators, in Figure 8.5, however, for 3000 modulators, the overhead of resampling the 2000 paths was too much to get meaningful results. For the graph with 2000 modulators, even though some cycles require degradation,

the time budget is enough not to miss a deadline.[2] For the graph with 3000 modulators, the degradation strategy prevents missing deadlines most of the time.

We started investigating the heuristics before working on the offline optimization that can pre-compute degraded versions of the graphs. Indeed, it appears that the overhead of the heuristics is usually too much for a real-time context: they insert too many resamplers, or they are smarter but take time to compute.

[2] It is surprising that we managed to execute more nodes on the less powerful MacBookPro, compared to the experiments with random graphs on a more powerful MacBookPro. We think it is due to the non-real time nature of the OS and the difficulty to reproduce the state of an OS, with all its background services, frequency scaling of the processor and so on.

Number of nodes	Mode	Degraded cycles	Number of edges	Time budget	Remaining time
10	EX	0.00	13.87	$18\,688.88 \pm 48.61$	3.26 ± 2.06
10	PROG	0.00	13.43	$18\,684.56 \pm 46.13$	3.16 ± 1.96
400	EX	2.24	23 961.39	9968.46 ± 1766.89	199.60 ± 1262.25
400	PROG	10.35	23 946.02	9241.36 ± 2944.51	602.82 ± 2816.39
1000	EX	500.00	149 883.01	$-45\,425.66 \pm 8753.99$	$19\,192.02 \pm 1976.93$
1000	PROG	500.00	149 771.73	$-52\,402.94 \pm 13\,086.58$	$26\,185.41 \pm 1827.86$

Number of nodes	Mode	Choosing duration	Number of resamplers	Degraded nodes
10	EX	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00
10	PROG	5.33 ± 4.04	0.00 ± 0.00	0.00 ± 0.00
400	EX	0.00 ± 0.00	5199.50 ± 3204.51	55.08 ± 41.44
400	PROG	921.04 ± 1016.85	438.24 ± 638.27	71.55 ± 46.84
1000	EX	0.00 ± 0.00	$53\,968.99 \pm 9416.71$	773.52 ± 72.17
1000	PROG	6813.13 ± 2489.17	$22\,448.03 \pm 6320.89$	748.36 ± 61.60

Table 8.1.: Results of the experiments. Each line corresponds to 100 random graphs with the same number of nodes. *EX* refers to the total heuristics and *PROG* to the progressive heuristics. All durations are in µs and when it is relevant, with their standard deviation. *Degraded cycles* is the average number of times a cycle has been degraded during the 500 cycles of a run. *Time budget* is the time budget that remains at the end of the execution of the callback. If it is negative, it means that the deadline has been missed by this duration. *Remaining time* is the time that is estimated to remain before finishing execution for one audio cycle when we first decide to degrade. *Choosing duration* is only relevant to the progressive strategy: it is the time to decide and choose the nodes to be degraded. The *number of resamplers* is the number of inserted resamplers during a degraded cycle. If it is strictly positive, it means that there were degraded cycles. *Degraded nodes* are the average of the number of degraded nodes per degraded cycles.

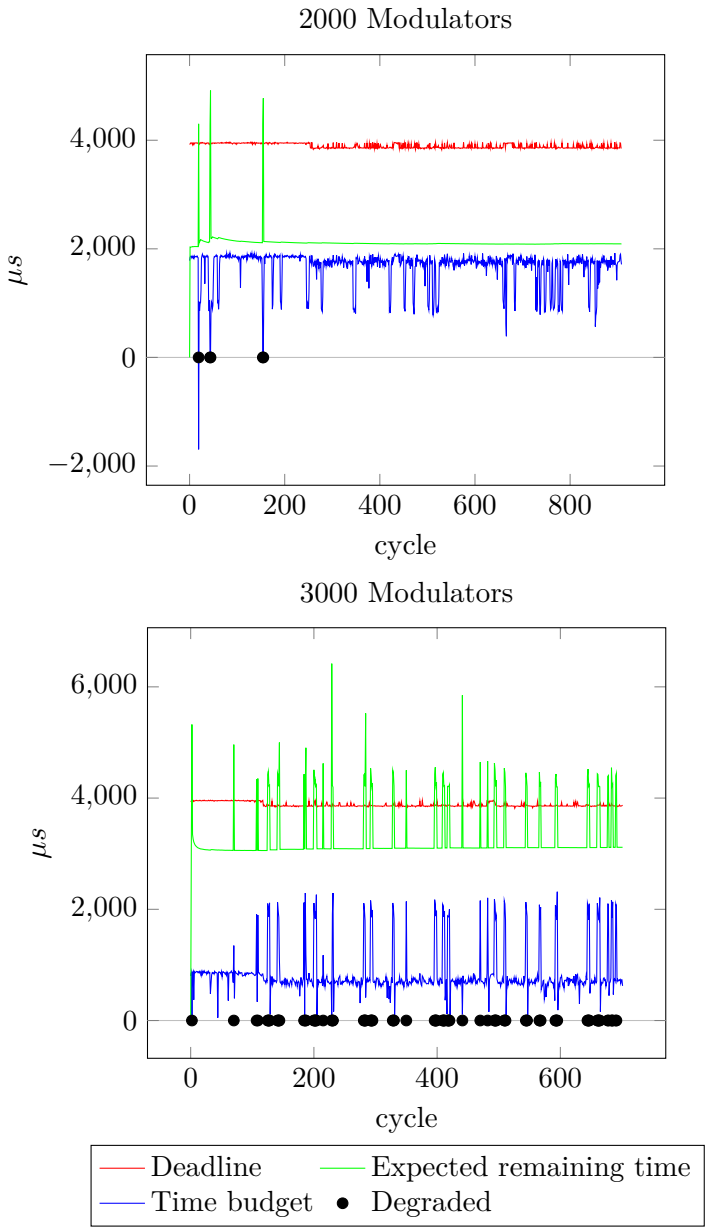


Figure 8.4.: Results for the chain graph of Figure 8.3b, with 2000 modulators and 3000 modulators respectively, using the exhaustive strategy. When the time budget is negative, it means there was a deadline miss. The expected remaining time is the estimated time for the cycle if a degradation decision is taken. Graphically, a degradation must occur if the estimated remaining time is above the deadline.

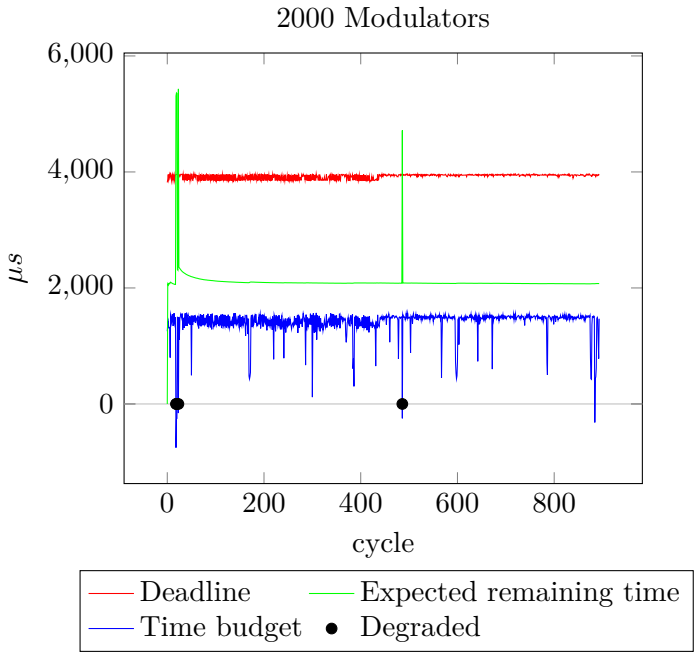


Figure 8.5.: Results for the comb graph of Figure 8.3b, with 2000 modulators using the exhaustive strategy. When the time budget is negative, it means there was a deadline miss. The expected remaining time is the estimated time for the cycle if a degradation decision is taken. Graphically, a degradation must occur if the estimated remaining time is above the deadline.

9. Conclusion and perspectives

9.1. Conclusion

That work was undertaken in the Repmus group at Ircam, in the team where Antescofo was born, and at Inria Paris. The idea of optimizing by resampling was born during a stay at the University of Salzburg in Austria with Prof. Christoph Kirsch.

Our work has drawn up to two directions related to the concept of *precision* or *approximation* in computing signals in audio graphs in IMSs and how it entails a trade-off between precision and performance, how to handle control and multiple rates, and how to force some rates by resampling to decrease the execution time of some processing.

At the end of this journey, I can draw some conclusions on the work and experiments carried out in this research.

Control and multiple rates. A first imprecision dwells in how multiple rates for signals are handled in IMSs: both periodic rates, usually corresponding to audio, whose timings are predictable, and aperiodic rates, for control events, for which the elements of the signal arrive at unpredictable instants. These controls events are not necessarily taken into account at the exact physical instant when they arrive. The audio signal is sampled and therefore, the control is possibly applied at the next sample. Moreover, often, audio samples are grouped and processed into blocks, and in that case, the control is applied to the next block. To better characterize how predictable and unpredictable multiple rates are, we presented a type system and a denotational semantics in Part I with timestamped buffers of samples as first-class citizens, which we compare with other formalizations of IMSs. The precision is represented by how much a buffer or a sample is delayed, *i.e.*, the latency, compared to when the sample arrived in the audio graph. There, we have tackled *buffering latency*. The type system distinguishes how a node can deal with a control

arriving at the middle of a buffer, at the next sample, or at the next buffer. We do not handle sub-sample accuracy. We do not describe how control can be smoothed either, as we reckon that it should be the responsibility of an explicit node `smooth`.

Optimization by resampling. Another imprecision resides in how fine-grained the sampled representation of a signal is. The more high-frequency content there is in the signal, the more fine-grained it must be, or in more technical terms, the higher the sample rate must be, which hints at how we degrade the signal, by *resampling*. Given an audio graph, we generate degraded versions of it that may decrease its execution-time. In Chapters 7 and 8, we actually deal with the *processing latency*. We describe an execution time model based on the average execution time and a structural quality model for an audio graph, where we describe how to combine qualities from individual nodes into paths of the audio graph. We show how we can choose subgraphs to degrade in the graph to optimize for quality or execution time under a time or quality constraint. The models are evaluated in experiments by enumerating the degraded versions and measuring empirical execution time and quality.

The execution-time model, based on the average execution time, is quite accurate but our quality model, which is structural but does not dive deeply into assessing the quality of an individual node, is less accurate for large graphs.

We also designed heuristics to explore a relevant subset of the degraded graphs but they need more evaluation. We use the optimized graphs in a real-time context to switch to them in case of overload, as well as heuristics that can degrade a graph at the middle of an execution cycle. The optimized graphs are useful in case of *permanent overload*, whereas we thought the heuristics would be more relevant for *transient overload*. However, experimental results show that these heuristics are still costly in a real-time context and we need to pursue experimenting with the swapping to the degraded versions.

Another obstacle to optimizing by resampling is that composers and musicians might be afraid of *degrading* their music, even though it is not audible. Therefore, the optimization will be probably more embraced when degrading from high sample rates, from 96 kHz to 48 kHz for instance.

An audio extension for Antescofo. We developed an audio extension to Antescofo which beforehands had to delegate audio processing to its host, such as *PureData* or Max/MSP. The extension is high-level; it aims at connecting heterogeneous nodes coded in Faust or in C++, not at coding directly audio processing nodes in Antescofo. Connecting nodes becomes a first-class citizen

reaction in Antescofo and as such, the audio graph can be reconfigured during execution as a reaction to some events, *i.e.*, we can change the connections between the audio nodes. However, nodes and graphs are not yet totally integrated into the programming model of Antescofo. For instance, nodes and audio graphs are not Antescofo values, and so we cannot store them in an array or a map. It makes it more difficult to programmatically build a graph and add new nodes during execution.[1] The audio extension handles multiple rates, for instance, for FFT analysis or video processing, which we demonstrate with a speed detector that leverages the OpenCV C++ library. Antescofo, as a complex scenario description language gives information about when some control can happen. For some kinds of control curves, the control can be sample-accurate. However, the splitting semantics of `elastic` input stream nodes is yet to be implemented.

Tooling. In addition to the development of an audio extension to Antescofo, this work led to two tools to optimize an audio graph:

- `ims-analysis`, an OCaml tool (see Appendix 2) to analyze audio graphs in IMSs and generate optimized versions. Puredata and Max patches, in addition to a custom audio graph format (see Appendix 1) are supported;
- `audio-adaptive-scheduling`, a small IMS (see Appendix 3) coded in Rust that can execute audio graphs in real time in our audio-graph format and perform online degradation with the heuristics.

9.2. Perspectives

In this section, we discuss improvements, new developments and new research directions. We list the perspectives in the same order of the topics of our work.

9.2.1. Type system and semantics

A difficulty of our type system is how we represent overlapping periodic buffers. They have the `periodic` type, where we lose the sample-periodicity and buffer size information. In the case of an FTT on overlapping buffers, it seems it is not a huge issue as the overlapping buffers are immediately consumed by the FFT node which outputs one multidimensional sample per period, *i.e.*, an array of frequency bins which has a known buffer size of 1. However, for the

[1] Yet, Antescofo has a powerful macro system, and so we can still build complex graphs more easily at loading time.

sake of elegance and as we may want to perform different operations on the overlapping buffers, we could introduce another type which would correspond to an homogeneous periodic stream, *i.e.*, buffer with the same size:

$$\texttt{homogeneous}(p, n, m, e)$$

where $p \in \mathcal{P}$ is the sample-period, $n \in \mathbb{N} \setminus \{0\}$ is the buffer size, $e \in \mathcal{U}$ is the sample type, and $m \in \mathbb{Z}$ with $n + m > 0$ is the number of samples added or removed from a buffer. If $m = 0$, it corresponds to a `buffered` type. If $m > 0$, it models overlapping buffers. If $m < 0$, it models a stream with buffers with a gap in-between. We are not sure if that last case refers to any real situation in audio processing though.

We also want to add more theoretical results, such as a study of the completeness of the type system or the causality of the semantics. We took care of defining causal operators but did not prove causality formally. Another interesting result that we could prove is that we can deduce an upper-bound on the latency computed by the semantics by looking only at the type system.

When a node computes only on controls, *i.e.*, `aperiodic` streams, our semantics activates the node on the union of the timestamps of the inputs. We can make it more flexible: similarly to Max/MSP and Puredata, which distinguish *hot* and *cold* inlets, we could also choose that the node is activated only on timestamps of the union of the so-called *hot* inputs.

9.2.2. Audio extension of Antescofo

The Antescofo score language makes it possible to express complex scenarios. We think that we could better schedule the audio processing if we better make use of the timing information in the score. For instance, if a sensor that controls some parameters is only instantiated after some measure in the score, we can assume that this control parameter will not change until that measure. In that case, we could compute the audio statically, a bit in advance, when the processor is less loaded. Furthermore, Antescofo has been recently able to compile pure functions (functions defined with `@fun_def`)). We want to allow the Antescofo programmer to define signal processing functions also using these functions, when they have float arrays as arguments.

Another point of improvement is to fully implement the semantics of Part I for the audio extension. So far, when a control arrives at the middle of the processing of a buffer for a node that can process sample by sample, we do not split the buffer but delay the control to the next block.

9.2.3. Optimization of audio graphs by resampling

More experiments. Our first experiments in Chapter 7.5 and Chapter 8.4 gave us insights on the execution time and quality model, but we still need to assess how the heuristics in the offline case behave, *i.e.*, how much of the degraded graph space they explore. In the online case, we have realized that the heuristics that can degrade at the middle of the cycle do not behave well and take too much time, either because they insert too many resamplers, or because they search the graph for too long. We suppose that using pre-computed graphs is a more effective solution. We are working on evaluating to which extent.

Model of quality and operators. In Section 7.3.1, we propose a structural mode of quality and we choose a specific join operator $\oplus$ and path operator $\otimes$, respectively min and the average, and $\times$. Other choices of operators can lead to a semiring structure that enables efficient optimization algorithms [Moh02].

Driving the precision of control with a quality model. Currently, if a node supports sample accuracy, we make sure that the control is taken into account at the precision of a sample during the execution. However, to pursue the trade-off between precision and performance, we could only perform sample-accurate computations when the node can do it *and* if the quality model has not selected the node to be degraded.

Exact model of quality of a small audio-processing language. The current quality model is not precise: it assumes that downsampling impacts all nodes in the same way. However, some nodes only touch the low range of the spectrum and hence are oblivious to the downsampling. If we know exactly what each node does, we can get a better approximation of the range of frequencies output by a node, using abstract interpretation techniques. This is feasible for a small language with simple arithmetic nodes, such as $+$, $\times$ and so on.

In Faust [OJ16], bounds on the amplitude of the signal are computed. We would like to compute also bounds on the frequency range. It will actually be always encompassed between 0 and the maximum audible frequency for humain beings, *i.e.*, 20 kHz, as we do not care about non-audible frequencies.

Speeding up the the enumeration. In the experiments of Section 7.5, we realized that a non-negligible number of degraded graphs take more time to execute than the non-degraded graph. Therefore, we want to detect that problem early in the generation to prevent unnecessary computations. When enumerating, we first get the possible sets of degraded nodes within the graph. In practice,

we do not need to build all the associated graphs to save some memory and accelerate the enumeration. We can get a crude lower bound of the execution time just with the subset of degraded nodes. Given a graph G, the execution time of a degraded version G' is given by:

$$A_{G'} = \sum_{v \in V_{\text{non degraded}}} A_v + \sum_{v \in V_{\text{degraded}}} A_v + \sum_{v \in V_{\text{resamplers}}} A_v \tag{9.1}$$

To estimate the number of resamplers necessary to degrade a subset, we look for the nodes that do not have incoming connections from, or outcoming connections to another node of the subset. It gives us an estimation of the number of resamplers to insert and hence a lower bound on the execution time of the degraded graph. If the subset is implemented with a hash table, we can check if a connection leads to a node in the subset in $O(1)$ in average. Then we can check whether the lower bound is smaller or not than the execution time of the non-degraded graph. If it is much higher[2], the graph associated to the degraded subsets does not need to be generated.

Implementation in Faust. We want to implement the offline optimization algorithm into Faust. We think it would be a coherent addition to a language that aims at describing mathematically signal processing and let the compiler optimize. Adding the optimization algorithm in Faust requires Faust support of multirate, which is still a work in progress, but it may be a good motivation to complete its implementation.

This work has been undertaken with IMSs as target. IMSs are outstanding examples of *cyber-physical systems*, where embedded systems at work in our daily life – airplanes, cars, drones – interact with the physical world and with human beings, the *human-in-the-loop*. We surmise that this thesis may prove useful to model, analyze and execute more general *cyber-physical systems*.

[2]Much higher, not just higher, to take into account the imprecision of the estimation of the average execution time.

Bibliography

My publications

[Che+16] Nathanaël Cheriere, Pierre Donat-Bouillud, Shadi Ibrahim, and Matthieu Simonin. "On the Usability of Shortest Remaining Time First Policy in Shared Hadoop Clusters". In: *SAC 2016-The 31st ACM/SIGAPP Symposium on Applied Computing*. Pisa, Italy, Apr. 2016. URL: https://hal.inria.fr/hal-01239341.

[DG17] Pierre Donat-Bouillud and Jean-Louis Giavitto. "Typing heterogeneous dataflow graphs for static buffering and scheduling". In: *ICMC 2017 - 43rd International Computer Music Conference*. Shanghai, China, Oct. 2017. URL: https://hal.inria.fr/hal-01585489.

[DGJ19] Pierre Donat-Bouillud, Jean-Louis Giavitto, and Florent Jacquemard. "Optimization of audio graphs by resampling". In: *DAFx-19 - 22nd International Conference on Digital Audio Effects*. Proceedings of the 22nd International Conference on Digital Audio Effects. Birmingham, United Kingdom, Sept. 2019. URL: https://hal.inria.fr/hal-02284258.

[DJS15] Pierre Donat-Bouillud, Florent Jacquemard, and Masahiko Sakai. "Towards an Equational Theory of Rhythm Notation". In: *Music Encoding Conference 2015*. Florence, Italy, May 2015. URL: https://hal.inria.fr/hal-01105418.

[DK17] Pierre Donat-Bouillud and Christoph M Kirsch. "Work-in-Progress: Adaptive Scheduling with Approximate Computing for Audio Graphs". In: *2017 IEEE Real-Time Systems Symposium (RTSS)*. IEEE. 2017, pp. 372–374.

[Don+16] Pierre Donat-Bouillud, Jean-Louis Giavitto, Arshia Cont, Nicolas Schmidt, and Yann Orlarey. "Embedding native audio-processing in a score following system with quasi sample accuracy". In: *ICMC 2016-42th International Computer Music Conference*. ICMC. Sept. 2016.

[Don18] Pierre Donat-Bouillud. "Ordonnancement adaptatif d'un graphe audio avec dégradation de qualité". In: *JIM 2018 - Journées d'Informatique Musicale*. Amiens, France, May 2018, pp. 1–9. URL: https://hal.archives-ouvertes.fr/hal-01791407.

[FGD19] José Miguel Fernandez, Jean-Louis Giavitto, and Pierre Donat-Bouillud. "AntesCollider: Control and Signal Processing in the Same Score". In: *ICMC 2019 - International Computer Music Conference*. New York, United States, June 2019. URL: https://hal.inria.fr/hal-02159629.

[JDB15a] Florent Jacquemard, Pierre Donat-Bouillud, and Jean Bresson. "A Structural Theory of Rhythm Notation based on Tree Representations and Term Rewriting". In: *Mathematics and Computation in Music: 5th International Conference, MCM 2015*. Ed. by David Meredith Tom Collins and Anja Volk. Vol. 9110. Lecture Notes in Artificial Intelligence. Oscar Bandtlow and Elaine Chew. London, United Kingdom: Springer, June 2015, p. 12. URL: https://hal.inria.fr/hal-01138642.

[JDB15b] Florent Jacquemard, Pierre Donat-Bouillud, and Jean Bresson. *A Term Rewriting Based Structural Theory of Rhythm Notation*. Research Report. ANR-13-JS02-0004-01 - EFFICACe, Mar. 2015, p. 11. URL: https://hal.inria.fr/hal-01134096.

References

[7419] Cycling 74. *Max 8 poly documentation*. July 2019. URL: https://docs.cycling74.com/max8/refpages/poly~.

[AA09] Pau Arumı and Xavier Amatriain. "Time-triggered static schedulable dataflows for multimedia systems". In: *Multimedia Computing and Networking 2009*. Vol. 7253. International Society for Optics and Photonics. 2009, p. 72530D.

[AAG06] Xavier Amatriain, Pau Arumi, and David Garcia. "CLAM: A framework for efficient and rapid development of cross-platform audio applications". In: *Proceedings of the 14th ACM international conference on Multimedia*. ACM. 2006, pp. 951–954.

[ABD11] Saamer Akhshabi, Ali C Begen, and Constantine Dovrolis. "An experimental evaluation of rate-adaptation algorithms in adaptive streaming over HTTP". In: *Proceedings of the second annual ACM conference on Multimedia systems*. ACM. 2011, pp. 157–168.

[AD99] David Aldous and Persi Diaconis. “Longest increasing subsequences: from patience sorting to the Baik-Deift-Johansson theorem”. In: *Bulletin of the American Mathematical Society* 36.4 (1999), pp. 413–432.

[ADA08] Antoine Allombert, Myriam Desainte-Catherine, and Gérard Assayag. “Iscore: a system for writing interaction”. In: *Proceedings of the 3rd international conference on Digital Interactive Media in Entertainment and Arts*. ACM. 2008, pp. 360–367.

[AG12] Andrea Agostini and Daniele Ghisi. “Bach: An environment for computer-aided composition in max”. In: *ICMC*. 2012.

[AK08] Oana Andrei and Hélene Kirchner. “A rewriting calculus for multigraphs with ports”. In: *Electronic Notes in Theoretical Computer Science* 219 (2008), pp. 67–82.

[Årz+11] Karl-Erik Årzén, Vanessa Romero Segovia, Stefan Schorr, and Gerhard Fohler. “Adaptive resource management made real”. In: *3rd Workshop on Adaptive and Reconfigurable Embedded Systems*. 2011.

[Ass86] ITU Radiocommunication Assembly. *RECOMMENDATION ITU-R BS.468-4 - Measurement of audio-frequency noise voltage*. 1986.

[BAA11] Jean Bresson, Carlos Agon, and Gérard Assayag. “Openmusic–visual programming environment for music composition, analysis and research”. In: 2011.

[Bau+13] Guillaume Baudart, Florent Jacquemard, Louis Mandel, and Marc Pouzet. “A synchronous embedding of Antescofo, a domain-specific language for interactive mixed music”. In: *Proceedings of the Eleventh ACM International Conference on Embedded Software*. IEEE Press. 2013, p. 1.

[BB01a] Ross Bencina and Phil Burk. “PortAudio-an Open Source Cross Platform Audio API.” In: *ICMC*. 2001.

[BB01b] B. Bhattacharya and S. S. Bhattacharyya. “Parameterized dataflow modeling for DSP systems”. In: *IEEE Transactions on Signal Processing* 49.10 (Oct. 2001), pp. 2408–2421. DOI: 10.1109/78.950795.

[Bel+11] Jose A Belloch, Alberto Gonzalez, Francisco-Jose Martınez-Zaldıvar, and Antonio M Vidal. “Real-time massive convolution for audio applications on GPU”. In: *The Journal of Supercomputing* 58.3 (2011), pp. 449–457.

[Ben11] Ross Bencina. "The SuperCollider Book". In: ed. by S. Wilson, D. Cottle, and N. Collins. MIT Press, 2011. Chap. Inside Scynth. URL: http://supercolliderbook.net/.

[Bev+10] Frédéric Bevilacqua, Bruno Zamborlin, Anthony Sypniewski, Norbert Schnell, Fabrice Guédy, and Nicolas Rasamimanana. "Continuous realtime gesture following and recognition". In: *Gesture in Embodied Communication and Human-Computer Interaction: Lecture Notes in Computer Science (LNCS) volume 5934*. Springer Verlag, 2010, pp. 73–84. URL: http://articles.ircam.fr/textes/Bevilacqua09b/index.pdf.

[BFW10] Eric Battenberg, Adrian Freed, and David Wessel. "Advances In The Parallelization Of Music And Audio Applications." In: *ICMC*. 2010.

[BG14] Jean Bresson and Jean-Louis Giavitto. "A reactive extension of the openmusic visual programming language". In: *Journal of Visual Languages & Computing* 25.4 (2014), pp. 363–375.

[BG92] Gérard Berry and Georges Gonthier. "The Esterel synchronous programming language: Design, semantics, implementation". In: *Science of computer programming* 19.2 (1992), pp. 87–152.

[Bha+18] Shuvra S Bhattacharyya, Ed F Deprettere, Rainer Leupers, and Jarmo Takala. *Handbook of signal processing systems*. Springer, 2018.

[Bil+96] Greet Bilsen, Marc Engels, Rud Lauwereins, and Jean Peperstraete. "Cycle-static dataflow". In: *Signal Processing, IEEE Transactions on* 44.2 (1996), pp. 397–408.

[BJ13] Karim Barkati and Pierre Jouvelot. "Synchronous programming in audio processing: A lookup table oscillator case study". In: *ACM Computing Surveys (CSUR)* 46.2 (2013), p. 24.

[Ble11] T. Blechmann. "Supernova-A Multiprocessor Aware Real-Time Audio Synthesis Engine For SuperCollider". MA thesis. Vienna University of Technology, 2011. URL: http://tim.klingt.org/publications/tim_blechmann_supernova.pdf.

[BLJ91] Albert Benveniste, Paul Le Guernic, and Christian Jacquemot. "Synchronous programming with events and relations: the SIGNAL language and its semantics". In: *Science of computer programming* 16.2 (1991), pp. 103–149.

[BPT06] Neil Burroughs, Adam Parkin, and George Tzanetakis. "Flexible Scheduling for DataFlow Audio Processing." In: *ICMC*. 2006.

[Bra00] G. Bradski. "OpenCV Library". In: *Dr. Dobb's Journal of Software Tools* (2000).

[Bro91] Judith C Brown. "Calculation of a constant Q spectral transform". In: *The Journal of the Acoustical Society of America* 89.1 (1991), pp. 425–434.

[Buc+04] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. "Brook for GPUs: stream computing on graphics hardware". In: *ACM transactions on graphics (TOG)* 23.3 (2004), pp. 777–786.

[But+02] Giorgio C Buttazzo, Giuseppe Lipari, Marco Caccamo, and Luca Abeni. "Elastic scheduling for flexible workload management". In: *IEEE Transactions on Computers* 51.3 (2002), pp. 289–302.

[But02] Giorgio C Buttazzo. "Scalable applications for energy-aware processors". In: *International workshop on embedded software*. Springer. 2002, pp. 153–165.

[BWJ14] Karim Barkati, Haisheng Wang, and Pierre Jouvelot. "Faustine: a vector faust interpreter test bed for multimedia signal processing". In: *International Symposium on Functional and Logic Programming*. Springer. 2014, pp. 69–85.

[Cam15] Ede Cameron. "Parallelizing the ALSA modular audio synthesizer". PhD thesis. Concordia University, 2015.

[CDC16] Jean-Michaël Celerier, Myriam Desainte-Catherine, and Jean-Michel Couturier. "Rethinking the audio workstation: tree-based sequencing with i-score and the LibAudioStream". In: *Sound and Music Computing Conference*. Hamburg, Germany, Aug. 2016. URL: `https://hal.archives-ouvertes.fr/hal-01360797`.

[Cel+15] Jean-Michaël Celerier, Pascal Baltazar, Clément Bossut, Nicolas Vuaille, Jean-Michel Couturier, et al. "OSSIA: Towards a unified interface for scoring time and interaction". In: 2015.

[CG14] Arshia Cont and Jean-Louis Giavitto. "Antescofo workshop at ICMC: Composing and Performing with Antescofo". In: *Joint ICMC - SMC Conference*. The remake of *Anthèmes 2* is part of the tutorial and it can be downloaded at `http://forumnet.ircam.fr/products/antescofo/`. Athens, Greece, Sept. 2014.

[Cha+04] Francois Charot, Madeleine Nyamsi, Patrice Quinton, and Charles Wagner. "Modeling and scheduling parallel data flow systems using structured systems of recurrence equations". In: *Proceedings. 15th IEEE International Conference on Application-Specific Systems, Architectures and Processors, 2004.* IEEE. 2004, pp. 6–16.

[Col+03] Nick Collins, Alex McLean, Julian Rohrhuber, and Adrian Ward. "Live coding in laptop performance". In: *Organised sound* 8.3 (2003), pp. 321–330.

[Con+12] Arshia Cont, José Echeveste, Jean-Louis Giavitto, and Florent Jacquemard. "Correct Automatic Accompaniment Despite Machine Listening or Human Errors in Antescofo". In: *Proceedings of International Computer Music Conference (ICMC).* IRZU - the Institute for Sonic Arts Research. Ljubljana, Slovenia, Sept. 2012. URL: `http://hal.inria.fr/hal-00718854`.

[Con10] Arshia Cont. "A coupled duration-focused architecture for real-time music-to-score alignment". In: *Pattern Analysis and Machine Intelligence, IEEE Transactions on* 32.6 (2010), pp. 974–987.

[Cor+09] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms.* MIT press, 2009.

[CSH19] Louis-Claude Canon, Mohamad El Sayah, and Pierre-Cyrille Héam. "A Comparison of Random Task Graph Generation Methods for Scheduling Problems". In: *arXiv preprint arXiv:1902.05808* (2019).

[Dan+78] Wayne W Daniel et al. *Applied nonparametric statistics.* Houghton Mifflin, 1978.

[Dan+93] Roger B Dannenberg et al. "The implementation of nyquist, a sound synthesis language". In: *PROCEEDINGS OF THE INTERNATIONAL COMPUTER MUSIC CONFERENCE.* INTERNATIONAL COMPUTER MUSIC ASSOCIATION. 1993, pp. 168–168.

[Dan02] Roger B Dannenberg. "A language for interactive audio applications". In: *PROCEEDINGS OF THE INTERNATIONAL COMPUTER MUSIC CONFERENCE.* INTERNATIONAL COMPUTER MUSIC ASSOCIATION. 2002.

[Dan08] Roger B Dannenberg. "Is Music Audio Processing Embarrassingly Parallel?" In: *ICMC.* Citeseer. 2008.

[Dan97] Mark Danks. "Real-time Image and Video Processing in GEM." In: *ICMC.* 1997.

[DB96] Roger B Dannenberg and Eli Brandt. "A flexible real-time software synthesis system". In: *Proceedings of the International Computer Music Conference*. INTERNATIONAL COMPUTER MUSIC ASSOCIATION. 1996, pp. 270–273.

[DC16] Roger B Dannenberg and Zhang Chi. "O2: Rethinking Open Sound Control". In: *Proceedings of the International Computer Music Conference*. 2016, p. 494.

[Déc+99] François Déchelle, Riccardo Borghesi, Maurizio De Cecco, Enzo Maggi, Butch Rovan, and Norbert Schnell. "jMax: an environment for real-time musical applications". In: *Computer Music Journal* 23.3 (1999), pp. 50–58.

[Del+87] C Delgado, P Loos, K Fritzson, and N Andersson. "Semantics of digital circuits". In: *Lecture Notes in Computer Science* 285 (1987).

[DL06] Gilles Dowek and Jean-Jacques Lévy. *Introduction à la théorie des langages de programmation*. Editions Ecole Polytechnique, 2006.

[DM98] Leonardo Dagum and Ramesh Menon. "OpenMP: An industry-standard API for shared-memory programming". In: *Computing in Science & Engineering* 1 (1998), pp. 46–55.

[Dry15] Alexandros Drymonitis. "Embedded Computers and Going Wireless". In: *Digital Electronics for Musicians*. Berkeley, CA: Apress, 2015, pp. 97–139. ISBN: 978-1-4842-1583-8. DOI: 10.1007/978-1-4842-1583-8_3. URL: https://doi.org/10.1007/978-1-4842-1583-8_3.

[Ech+11] José Echeveste, Arshia Cont, Jean-Louis Giavitto, and Florent Jacquemard. "Formalisation des relations temporelles dans un contexte d'accompagnement musical automatique". In: *8e Colloque sur la Modélisation des Systèmes Réactifs (MSR'11)*. Vol. 45. 2011, pp. 109–124.

[Ech+13] José Echeveste, Arshia Cont, Jean-Louis Giavitto, and Florent Jacquemard. "Operational semantics of a domain specific language for real time musician–computer interaction". In: *Discrete Event Dynamic Systems* 23.4 (2013), pp. 343–383.

[Ech15] José Echeveste. "Un langage de programmation pour composer l'interaction musicale". PhD thesis. Paris VI, 2015.

[EGC13] José Echeveste, Jean-Louis Giavitto, and Arshia Cont. *A Dynamic Timed-Language for Computer-Human Musical Interaction*. Research Report RR-8422. Dec. 2013. URL: `https://hal.inria.fr/hal-00917469`.

[ER60] Paul Erdős and Alfréd Rényi. "On the evolution of random graphs". In: *Publ. Math. Inst. Hung. Acad. Sci* 5.1 (1960), pp. 17–60.

[EY16] Pontus Ekberg and Wang Yi. "Schedulability analysis of a graph-based task model for mixed-criticality systems". In: *Real-time systems* 52.1 (2016), pp. 1–37.

[Fag+09] Dario Faggioli, Fabio Checconi, Michael Trimarchi, and Claudio Scordino. "An EDF scheduling class for the Linux kernel". In: *Proceedings of the Eleventh Real-Time Linux Workshop*. Citeseer. 2009.

[Fea91] Paul Feautrier. "Dataflow analysis of array and scalar references". In: *International Journal of Parallel Programming* 20.1 (1991), pp. 23–53.

[Fer+17] José Miguel Fernandez, Thomas Köppel, Nina Verstraete, Grégoire Lorieux, Alexander Vert, and Philippe Spiesser. "Gekipe, a gesture-based interface for audiovisual performance." In: *NIME*. 2017, pp. 450–455.

[ffi09] John ffitch. "Parallel execution of csound". In: *Proc. of the International Computer Music Conference (ICMC*. 2009.

[FM33] Harvey Fletcher and Wilden A Munson. "Loudness, its definition, measurement and calculation". In: *Bell System Technical Journal* 12.4 (1933), pp. 377–430.

[FOL11] Dominique Fober, Yann Orlarey, and Stéphane Letz. "FAUST Architectures Design and OSC Support." In: *International Conference on Digital Audio Effects*. Ed. by IRCAM. Paris, France, 2011, pp. 231–216. URL: `https://hal.archives-ouvertes.fr/hal-02158816`.

[FV86] Michael A Fligner and Joseph S Verducci. "Distance based ranking models". In: *Journal of the Royal Statistical Society: Series B (Methodological)* 48.3 (1986), pp. 359–369.

[Gho+04] Arkadeb Ghosal, Thomas A Henzinger, Christoph M Kirsch, and Marco AA Sanvido. "Event-driven programming with logical execution times". In: *International Workshop on Hybrid Systems: Computation and Control*. Springer. 2004, pp. 357–371.

[GLL99] Alain Girault, Bilung Lee, and Edward A Lee. "Hierarchical finite state machines with multiple concurrency models". In: *IEEE Transactions on computer-aided design of integrated circuits and systems* 18.6 (1999), pp. 742–760.

[GM03] Vincent Goudard and Remy Muller. "Real-time audio plugin architectures". In: *Comparative study. IRCAM-Centre Pompidou. France* (2003).

[GN00] Emden R. Gansner and Stephen C. North. "An open graph visualization system and its applications to software engineering". In: *SOFTWARE - PRACTICE AND EXPERIENCE* 30.11 (2000), pp. 1203–1233.

[GS10] Marc Geilen and Sander Stuijk. "Worst-case performance analysis of synchronous dataflow scenarios". In: *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*. ACM. 2010, pp. 125–134.

[GS90] Carl A Gunter and Dana S Scott. "Semantic domains". In: *Formal Models and Semantics*. Elsevier, 1990, pp. 633–674.

[Gua16] Adrien Guatto. "A synchronous functional language with integer clocks". PhD thesis. PSL Research University, 2016.

[Haa+17] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. "Bringing the web up to speed with WebAssembly". In: *ACM SIGPLAN Notices*. Vol. 52. 6. ACM. 2017, pp. 185–200.

[Hei19] Brook Heisler. *Citerion, a statistics-driven micro-benchmarking tool*. Mar. 2019. URL: `https://bheisler.github.io/criterion.rs/book/criterion_rs.html`.

[HHK01] Thomas A Henzinger, Benjamin Horowitz, and Christoph Meyer Kirsch. "Giotto: A time-triggered language for embedded programming". In: *International Workshop on Embedded Software*. Springer. 2001, pp. 166–184.

[HLR92] Nicolas Halbwachs, Fabienne Lagnier, and Christophe Ratel. "Programming and verifying real-time systems by means of the synchronous data-flow language LUSTRE". In: *IEEE Transactions on Software Engineering* 9 (1992), pp. 785–793.

[HR95] Moncef Hamdaoui and Parameswaran Ramanathan. "A dynamic priority assignment technique for streams with (m, k)-firm deadlines". In: *IEEE transactions on Computers* 44.12 (1995), pp. 1443–1451.

[Izh17] Roey Izhaki. *Mixing audio: concepts, practices, and tools*. Routledge, 2017.

[JO11] Pierre Jouvelot and Yann Orlarey. "Dependent vector types for data structuring in multirate Faust". In: *Computer Languages, Systems & Structures* 37.3 (2011), pp. 113–131.

[JUC19] JUCE team at ROLI. *Soul language*. July 2019. URL: `https://github.com/soul-lang/SOUL`.

[Kah87] Gilles Kahn. "Natural semantics". In: *Annual symposium on theoretical aspects of computer science*. Springer. 1987, pp. 22–39.

[Ken48] Maurice George Kendall. "Rank correlation methods." In: (1948).

[Kie+15] Marc Aurel Kiefer, Korbinian Molitorisz, Jochen Bieler, and Walter F Tichy. "Parallelizing a Real-Time Audio Application–A Case Study in Multithreaded Software Engineering". In: *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*. IEEE. 2015, pp. 405–414.

[Kir02] Christoph M Kirsch. "Principles of real-time programming". In: *International Workshop on Embedded Software*. Springer. 2002, pp. 61–75.

[Knu97] Donald Ervin Knuth. *The art of computer programming*. Vol. 3. Pearson Education, 1997.

[KS12] Christoph M Kirsch and Ana Sokolova. "The logical execution time paradigm". In: *Advances in Real-Time Systems*. Springer, 2012, pp. 103–120.

[Kum02] Vipin Kumar. *Introduction to parallel computing*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[LA04] Chris Lattner and Vikram Adve. "LLVM: A compilation framework for lifelong program analysis & transformation". In: *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*. IEEE Computer Society. 2004, p. 75.

[Leb16] Jakob Leben. "Arrp: A Functional Language with Multi-dimensional Signals and Recurrence Equations". In: *Proceedings of the 4th International Workshop on Functional Art, Music, Modelling, and Design*. FARM 2016. Nara, Japan: ACM, 2016, pp. 17–28. ISBN: 978-1-4503-4432-6. DOI: 10.1145/2975980.2975983. URL: http://doi.acm.org/10.1145/2975980.2975983.

[Let+15] Stéphane Letz, Sarah Denoux, Yann Orlarey, and Dominique Fober. "Faust audio DSP language in the Web". In: *Linux Audio Conference*. Mainz, Germany, 2015, pp. 29–36. URL: https://hal.archives-ouvertes.fr/hal-02159002.

[Let+17] Stéphane Letz, Yann Orlarey, Dominique Fober, and Romain Michon. "Polyphony, sample-accurate control and MIDI support for FAUST DSP using combinable architecture files". In: *Linux Audio Conference*. Ed. by Vincent Ciciliato, Yann Orlarey, and Laurent Pottier. Saint-Etienne, France: CIEREC, 2017, pp. 69–75. URL: https://hal.archives-ouvertes.fr/hal-02159003.

[LFO13] Stéphane Letz, Dominique Fober, and Yann Orlarey. "COMMENT EMBARQUER LE COMPILATEUR FAUST DANS VOS APPLICATIONS ?" In: *Journées d'Informatique Musicale*. Paris, France, May 2013, pp. 137–140. URL: https://hal.archives-ouvertes.fr/hal-00832224.

[Liu+91] Jane WS Liu, Kwei-Jay Lin, Wei Kuan Shih, Albert Chuang-shi Yu, Jen-Yao Chung, and Wei Zhao. *Algorithms for scheduling imprecise computations*. Springer, 1991.

[LM87] Edward A Lee and David G Messerschmitt. "Synchronous data flow". In: *Proceedings of the IEEE* 75.9 (1987), pp. 1235–1245.

[LMQ91] Hervé Le Verge, Christophe Mauras, and Patrice Quinton. "The ALPHA language and its use for the design of systolic arrays". In: *Journal of VLSI signal processing systems for signal, image and video technology* 3.3 (1991), pp. 173–182.

[LNK05] Mikael Laurson, Vesa Norilo, and Mika Kuuskankare. "PWGLSynth: A visual synthesis language for virtual instrument design and control". In: *Computer Music Journal* 29.3 (2005), pp. 29–41.

[LOF05] Stéphane Letz, Yann Orlarey, and Dominique Fober. "Jack audio server for multi-processor machines". In: *International Computer Music Conference*. Ed. by ICMA. Barcelona, Spain, 2005, pp. 1–4. URL: https://hal.archives-ouvertes.fr/hal-02158920.

[LOF10] Stephane Letz, Yann Orlarey, and Dominique Fober. "Work stealing scheduler for automatic parallelization in faust". In: *Linux Audio Conference*. 2010.

[LOF13] S Letz, Y Orlarey, and D Fober. "Dynamic compilation of parallel audio applications". In: *Compilers for Parallels Computing* (2013).

[LOF18a] Stéphane Letz, Yann Orlarey, and Dominique Fober. "An Overview of the FAUST Developer Ecosystem". In: 2018.

[LOF18b] Stéphane Letz, Yann Orlarey, and Dominique Fober. "FAUST Domain Specific Audio DSP Language Compiled to WebAssembly". In: *Companion Proceedings of the The Web Conference 2018*. International World Wide Web Conferences Steering Committee. 2018, pp. 701–709.

[Lom11] Chris Lomont. "Introduction to intel advanced vector extensions". In: *Intel White Paper* (2011), pp. 1–21.

[Loy85] Gareth Loy. "Musicians make a standard: the MIDI phenomenon". In: *Computer Music Journal* 9.4 (1985), pp. 8–26.

[LP95] Edward A Lee and Thomas M Parks. "Dataflow process networks". In: *Proceedings of the IEEE* 83.5 (1995), pp. 773–801.

[LS16] Edward Ashford Lee and Sanjit A Seshia. *Introduction to embedded systems: A cyber-physical systems approach*. Mit Press, 2016.

[LT14] Jakob Leben and George Tzanetakis. "Declarative Composition and Reactive Control in Marsyas". In: *ICMC*. 2014.

[LZ09] Young Choon Lee and Albert Y Zomaya. "Minimizing energy consumption for precedence-constrained applications using dynamic voltage scaling". In: *Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*. IEEE Computer Society. 2009, pp. 92–99.

[McC02] James McCartney. "Rethinking the computer music language: SuperCollider". In: *Computer Music Journal* 26.4 (2002), pp. 61–68.

[McC96] James McCartney. "SuperCollider: a new real time synthesis language". In: *Proceedings of the International Computer Music Conference*. 1996. URL: `http://www.audiosynth.com/icmc96paper.html`.

[Mic13] Romain Michon. "Faust2android: a Faust architecture for Android". In: *Proceedings of the 16th International Conference on Digital Audio Effects (DAFx-13), Maynooth, Ireland*. 2013, pp. 2–6.

[ML02] Praveen K Murthy and Edward A Lee. "Multidimensional synchronous dataflow". In: *IEEE Transactions on Signal Processing* 50.8 (2002), pp. 2064–2079.

[Moh02] Mehryar Mohri. "Semiring frameworks and algorithms for shortest-distance problems". In: *Journal of Automata, Languages and Combinatorics* 7.3 (2002), pp. 321–350.

[Moo12] Brian CJ Moore. *An introduction to the psychology of hearing*. Brill, 2012.

[Mor+17] Fabio Morreale, Giulio Moro, Alan Chamberlain, Steve Benford, and Andrew P McPherson. "Building a maker community around an open hardware platform". In: *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*. ACM. 2017, pp. 6948–6959.

[Mor02] Lionel Morel. "Efficient compilation of array iterators for lustre". In: *Electronic Notes in Theoretical Computer Science* 65.5 (2002), pp. 19–26.

[Mos90] Peter D Mosses. *Handbook of Theoretical Computer Science, volume 2, chapter Denotational Semantics*. 1990.

[MP05] Louis Mandel and Marc Pouzet. "ReactiveML: a reactive extension to ML". In: *Proceedings of the 7th ACM SIGPLAN international conference on Principles and practice of declarative programming*. ACM. 2005, pp. 82–93.

[MPP10] Louis Mandel, Florence Plateau, and Marc Pouzet. "Lucy-n: a n-synchronous extension of Lustre". In: *International Conference on Mathematics of Program Construction*. Springer. 2010, pp. 288–309.

[NIS14] HIROKI NISHINO. "LC: A Mostly-strongly-timed Prototype-based Computer Music Programming Language that Integrates Objects and Manipulations for Microsound Synthesis". PhD thesis. 2014.

[Nis18a] Hiroki Nishino. "ON-STACK COMPUTATION OF AUDIO VECTORS FOR UNIT-GENERATOR-BASED SOUND SYNTHESIS". In: *International Workshop on Computer Music and Audio Technology*. 2018.

[Nis18b] Hiroki Nishino. "Unit-generator Graph as a Generator of Lazily Evaluated Audio-vector Trees". In: *Proc. of Sound and Music Computing*. 2018.

[NL09] Vesa Norilo and Mikael Laurson. "Kronos-a vectorizing compiler for music dsp". In: *Proc. of the 12th Int. Conference on Digital Audio Effects (DAFx-09)*. Vol. 317. 2009.

[Nor15] Vesa Norilo. "Kronos: A declarative metaprogramming language for digital signal processing". In: *Computer Music Journal* 39.4 (2015), pp. 30–48.

[Nor16] Vesa Norilo. "Kronos Meta-Sequencer–From Ugens to Orchestra, Score and Beyond". In: *42nd International Computer Music Conference 2016*. 2016.

[OFL04] Yann Orlarey, Dominique Fober, and Stephane Letz. "Syntactical and semantical aspects of Faust". In: *Soft Computing* 8.9 (2004), pp. 623–632.

[OFL09] Y. Orlarey, D. Fober, and S. Letz. "Faust: an efficient functional approach to DSP programming". In: *New Computational Paradigms for Computer Music* (2009).

[OJ16] Yann Orlarey and Pierre Jouvelot. "Signal rate inference for multidimensional faust". In: *Proceedings of the 28th Symposium on the Implementation and Application of Functional Programming Languages*. ACM. 2016, p. 1.

[OLF09] Yann Orlarey, Stephane Letz, and Dominique Fober. *Adding automatic parallelization to Faust*. na, 2009.

[OS14] Alan V Oppenheim and Ronald W Schafer. *Discrete-time signal processing*. Pearson Education, 2014.

[PB02] Benjamin C Pierce and C Benjamin. *Types and programming languages*. MIT press, 2002.

[PH13] David A Patterson and John L Hennessy. *Computer Organization and Design MIPS Edition: The Hardware/Software Interface*. Newnes, 2013.

[Pou06] Marc Pouzet. "Lucid synchrone, version 3". In: *Tutorial and reference manual. Université Paris-Sud, LRI* 1 (2006), p. 25.

[Pto14] Claudius Ptolemaeus. *System design, modeling, and simulation: using Ptolemy II*. Vol. 1. Ptolemy. org Berkeley, 2014.

[Puc+96] Miller Puckette et al. "Pure Data: another integrated computer music environment". In: *Proceedings of the second intercollege computer music concerts* (1996), pp. 37–41.

[Puc02a] M. Puckette. "Using Pd as a score language". In: *Proc. Int. Computer Music Conf.* Sept. 2002, pp. 184–187. URL: http://www.crca.ucsd.edu/~msp.

[Puc02b] Miller Puckette. "Max at Seventeen". In: *Comput. Music J.* 26.4 (2002), pp. 31–43. ISSN: 0148-9267. DOI: http://dx.doi.org/10.1162/014892602320991356.

[Puc09] Miller Puckette. "Multiprocessing for pd". In: *Proc. of the 3rd Int'l Pure Data Convention (PDCON09)*. 2009.

[Puc88] Miller Puckette. "The Patcher". In: *Proceedings of International Computer Music Conference (ICMC)*. 1988, pp. 420–429.

[Puc97] M. Puckette. "Pure data". In: *Proc. Int. Computer Music Conf.* Thessaloniki, Greece, Sept. 1997, pp. 224–227. URL: http://www.crca.ucsd.edu/~msp.

[PWW97] Alex Peleg, Sam Wilkie, and Uri Weiser. "Intel MMX for multimedia PCs". In: *Communications of the ACM* 40.1 (1997), pp. 24–38.

[RH11] Stuart Rosen and Peter Howell. *Signals and systems for speech and hearing*. Vol. 29. Brill, 2011.

[RH91] Frédéric Rocheteau and Nicolas Halbwachs. "Implementing reactive programs on circuits a hardware implementation of LUSTRE". In: *Workshop/School/Symposium of the REX Project (Research and Education in Concurrent Systems)*. Springer. 1991, pp. 195–208.

[Rit+93] Sebastian Ritz, Matthias Pankert, V Zivojinovic, and Heinrich Meyr. "Optimum vectorization of scalable synchronous dataflow graphs". In: *Application-Specific Array Processors, 1993. Proceedings., International Conference on*. IEEE. 1993, pp. 285–296.

[Row93] Robert Rowe. *Interactive Music Systems: Machine Listening and Composing*. AAAI Press, 1993.

[RT08] Rasmus V. Rasmussen and Michael A. Trick. "Round robin scheduling – a survey". In: *European Journal of Operational Research* 188.3 (2008), pp. 617–636. ISSN: 0377-2217. DOI: http://dx.doi.org/10.1016/j.ejor.2007.05.046. URL: http://www.sciencedirect.com/science/article/pii/S0377221707005309.

[San+18] Mark Santolucito, Kate Rogers, Aedan Lombardo, and Ruzica Piskac. "Programming-by-example for audio: synthesizing digital signal processing programs". In: *Proceedings of the 6th ACM SIGPLAN International Workshop on Functional Art, Music, Modeling, and Design*. ACM. 2018, pp. 18–25.

[Sch70] Manfred R Schroeder. "Digital simulation of sound transmission in reverberant spaces". In: *The Journal of the Acoustical Society of America* 47.2A (1970), pp. 424–431.

[Sch87] Martine Schlag. "The planar topology of functional programs". In: *Conference on Functional Programming Languages and Computer Architecture*. Springer. 1987, pp. 174–193.

[Sen68] Pranab Kumar Sen. "Estimates of the regression coefficient based on Kendall's tau". In: *Journal of the American statistical association* 63.324 (1968), pp. 1379–1389.

[SG84] Julius O Smith and Phil Gossett. "A flexible sampling-rate conversion method". In: *Acoustics, Speech, and Signal Processing, IEEE International Conference on ICASSP'84*. Vol. 9. IEEE. 1984, pp. 112–115.

[Sha49] Claude Elwood Shannon. "Communication in the presence of noise". In: *Proceedings of the IRE* 37.1 (1949), pp. 10–21.

[Sie82] Andrew F Siegel. "Robust regression using repeated medians". In: *Biometrika* 69.1 (1982), pp. 242–244.

[Smi19a] Julius O Smith III. *Interpolated delay lines, ideal bandlimited interpolation, and fractional delay filter design*. Apr. 2019. URL: `http://ccrma.%20stanford.%20edu/%5C~%7B%7D%20jos/Interpolation`.

[Smi19b] Julius O Smith. *Digital audio resampling home page*. Apr. 2019. URL: `http://www-ccrma.%20stanford.%20edu/%5C~%7B%7D%20jos/resample`.

[SN13] Ralf Steinmetz and Klara Nahrstedt. *Multimedia systems*. Springer Science & Business Media, 2013.

[Sor18] Andrew Carl Sorensen. "Extempore: The design, implementation and application of a cyber-physical programming language". PhD thesis. 2018.

[Sta+93] International Organization for Standardization/International Electrotechnical Commission et al. "Coding of moving pictures and associated audio for digital storage media at up to about 1.5 Mbit/s". In: *ISO/IEC 11172* (1993).

[Ste96] Ralf Steinmetz. “Human perception of jitter and media synchronization”. In: *IEEE Journal on selected Areas in Communications* 14.1 (1996), pp. 61–72.

[Suz+03] Yôiti Suzuki, Volker Mellert, Utz Richter, Henrik Møller, Leif Nielsen, Rhona Hellman, Kaoru Ashihara, Kenji Ozawa, and Hisashi Takeshima. “Precise and full-range determination of two-dimensional equal loudness contours”. In: *Tohoku University, Japan* (2003).

[The+06] Bart D Theelen, Marc CW Geilen, Twan Basten, Jeroen PM Voeten, Stefan Valentin Gheorghita, and Sander Stuijk. “A scenario-aware data flow model for combined long-run average and worst-case performance analysis”. In: *Formal Methods and Models for Co-Design, 2006. MEMOCODE'06. Proceedings. Fourth ACM and IEEE International Conference on*. IEEE. 2006, pp. 185–194.

[The92] Henri Theil. “A rank-invariant method of linear and polynomial regression analysis”. In: *Henri Theil's contributions to economics and econometrics*. Springer, 1992, pp. 345–381.

[TL15] Baltasar Trancón y Widemann and Markus Lepper. “The shepard tone and higher-order multi-rate synchronous data-flow programming in Sig”. In: *Proceedings of the 3rd ACM SIGPLAN International Workshop on Functional Art, Music, Modelling and Design*. ACM. 2015, pp. 6–14.

[VE90] Barry Vercoe and Dan Ellis. “Real-time CSound: Software Synthesis with Sensing and Control.” In: *ICMC*. Vol. 90. 1990, pp. 209–211.

[Ven+15] Swagath Venkataramani, Srimat T Chakradhar, Kaushik Roy, and Anand Raghunathan. “Computing approximately, and efficiently”. In: *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*. EDA Consortium. 2015, pp. 748–751.

[Vir07] Tuomas Virtanen. “Monaural sound source separation by nonnegative matrix factorization with temporal continuity and sparseness criteria”. In: *IEEE transactions on audio, speech, and language processing* 15.3 (2007), pp. 1066–1074.

[Wan09] G. Wang. “The ChucK audio programming language." A strongly-timed and on-the-fly environ/mentality"”. PhD thesis. Princeton University, 2009.

[WCS15] Ge Wang, Perry R. Cook, and Spencer Salazar. "ChucK: A Strongly Timed Computer Music Language". In: *Computer Music Journal* 39.4 (2015), pp. 10–29. DOI: 10.1162/COMJ_a_00324.

[WCS16] Ge Wang, Perry R Cook, and Spencer Salazar. "Chuck: A strongly timed computer music language". In: *Computer Music Journal* (2016).

[WFC07] Ge Wang, Rebecca Fiebrink, and Perry R Cook. "Combining Analysis and synthesis in the Chuck Programming Language." In: *ICMC*. 2007.

[Wil09] Christopher Wilson. *Csound Parallelism*. Tech. rep. Technical Report CSBU-2009-07, Department of Computer Science, University of Bath, 2009.

[Wri05] Matthew Wright. "Open Sound Control: an enabling technology for musical networking". In: *Organised Sound* 10.3 (2005), pp. 193–200.

[WS11] Haizhou Wang and Mingzhou Song. "Ckmeans. 1d. dp: optimal k-means clustering in one dimension by dynamic programming". In: *The R journal* 3.2 (2011), p. 29.

[WSR10] Graham Wakefield, Wesley Smith, and Charles Roberts. "LuaAV: Extensibility and heterogeneity for audiovisual computing". In: *Proceedings of Linux Audio Conference*. 2010.

[Xip19] Xiph.org Foundation. *Free Lossless Audio Code (Flac)*. July 2019. URL: https://xiph.org/flac.

[Zhu+12] Zeyuan Allen Zhu, Sasa Misailovic, Jonathan A Kelner, and Martin Rinard. "Randomized accuracy-aware program transformations for efficient approximate computations". In: *ACM SIGPLAN Notices*. Vol. 47. 1. ACM. 2012, pp. 441–454.

[Zic02] David Zicarelli. "How I Learned to Love a Program That Does Nothing". In: *Comput. Music J.* 26.4 (2002), pp. 44–51. ISSN: 0148-9267. DOI: http://dx.doi.org/10.1162/014892602320991365.

[Zmö04] Johannes Zmölnig. "Gem for pd-recent progress." In: *ICMC*. 2004.

[ZV06] Zoran Zivkovic and Ferdinand Van Der Heijden. "Efficient adaptive density estimation per image pixel for the task of background subtraction". In: *Pattern recognition letters* 27.7 (2006), pp. 773–780.

Tools

1. The audio graph format

We developed a custom format to describe audio graphs. The format must be able to handle graphs with ports and several edges between two nodes, which is not directly possible in the well-known graph visualisation Graphviz format [GN00]. It must also be able to represent graphs from various IMS such as Max/MSP, Puredata or Chuck, selecting a meaningful and relevant set of common attributes.

The audiograph format is used to exchange graphs between our analysis tool `ims-analysis` and a small IMS coded in Rust. `ims-analysis` can also convert Max/MSP patches and Puredata patches to the audiograph format.

Declaring a node A node has three main attributes: `kind`, similar to the *classname* in Max/MSP, `in`, the number of input ports and `out`, the number of output ports, as in Code .1. Attribute `kind` is compulsory but the two other ones can be omitted and will have a value of 0 in that case. More attributes can be added to control other parameters of a specific node. For instance, for a synthesizer, we add a `freq` attribute, which controls the synthesized frequency.

```
n1 =
{
    kind : "osc",
    freq : "440",
    in: 0,
    out : 1,
};
```

Code .1: An simple oscillator node with no input port and one output port.

Connecting nodes Ports of nodes are connected together using the `->` keyword, as in Code .2. Connections can also be chained.

```
n1.1 -> n3.2;
n3.1 -> n4.1 -> n5.1 ;
```

Code .2: Output port 1 of node n1 is connected to input port 2 of node n3. On the second line, we write the connections between three nodes.

2. The `ims-analysis` program

The tool is open source and available on `https://github.com/programLyrique/ims-analysis`.

We give here the command line options of the tool:

```
usage: ims_analysis [options] [input_file]

options:

    Make analysis and optimizations of IMS programs

    --version             show program's version and exit
    -h, --help            show this help message and exit
    --debug               Debug messages

    Input:

        Input options

        --connect-subpatches
            Connects subpatches to get only one connected graph,
            not several components per subpatch

    Display:

        Display options

        -oSTR, --output-name=STR
            Name of the output file (without extension)
        -d, --dot           Outputs a dot file of the signal processing graph
        -e, --audiograph    Outputs an audiograph file
            of the signal processing  graph
```

```
    -s, --stats         Stats about the processing and the graphs
    -r, --report        A report about the optimization process

Optimizations:

    Various optimizations

    -w, --downsample    Optimization by downsampling

    Downsampling tweaking:

        -aFLOAT, --deadline=FLOAT
            Deadline of the audio callback in ms
        -mFLOAT, --resampler-dur=FLOAT
            Duration of a resampler in ms
        -x, --exhaustive  Exhaustive exploration
        -l, --random      Random exploration
        --merge-resamplers
            Merging resampler optimization
        --nb-samples=INT  Number of samples
        -z, ----use-graphs
            From existing audio graphs
        -nINT, --nb-nodes=INT
            Number of nodes in case of enumerating/random
         generation all connected directed graphs with n nodes
        -pFLOAT, --edge-prob=FLOAT
            Edge probability in case of random generation of
            connected directed graphs with n nodes
        --node-file=STR   Definitions of possible nodes for use for full
            enumeration.
```

3. The Rust prototype IMS

The tool is open source and available on `https://github.com/programLyrique/audio-adaptive-scheduling`. It also features scripts to reproduce the experiments.

We give here the command line options of the tool:

```
USAGE:
audiograph [FLAGS] [OPTIONS] <INPUT> <--real-time|--bounce>
```

```
FLAGS:
    -a, --audio-input    Audio input used as source when bouncing
    -b, --bounce         Execute the graph offline (bounce),
                                as fast as possible.
    -h, --help           Prints help information
    -m, --monitor        Monitor execution and save it as a csv file.
    -r, --real-time      Execute in real-time
    --silent               No output at all on the terminal.
    -V, --version        Prints version information

OPTIONS:
    -c, --cycles <NbCycles>    Number of cycles to execute the audio graph

ARGS:
    <INPUT>    Sets the audiograph to use.
```

List of Figures

List of Tables

List of Algorithms

List of Codes