

2014

# Tracing the Compositional Process. Sound art that rewrites its own past: formation, praxis and a computer framework

Rutz, Hanns Holger

<http://hdl.handle.net/10026.1/3116>

---

<http://dx.doi.org/10.24382/3447>

Plymouth University

---

*All content in PEARL is protected by copyright law. Author manuscripts are made available in accordance with publisher policies. Please cite only the published version using the details provided on the item record or document. In the absence of an open licence (e.g. Creative Commons), permissions for further reuse of content should be sought from the publisher or author.*

This copy of the thesis has been supplied on condition that anyone who consults it is understood to recognise that its copyright rests with its author and that no quotation from the thesis and no information derived from it may be published without the author's prior consent.



# **Tracing the Compositional Process**

**Sound art that rewrites its own past:  
formation, praxis and a computer framework**

by

**Hanns Holger Rutz**

A thesis submitted to the University of Plymouth  
in partial fulfillment for the degree of

**Doctor of Philosophy**

Interdisciplinary Centre for Computer Music Research (ICCMR)

School of Humanities and Performing Arts

Faculty of Arts

August 2014



# Abstract

## Tracing the Compositional Process

Hanns Holger Rutz

The domain of this thesis is electroacoustic computer-based music and sound art. It investigates a facet of composition which is often neglected or ill-defined: the process of composing itself and its embedding in time. Previous research mostly focused on instrumental composition or, when electronic music was included, the computer was treated as a tool which would eventually be subtracted from the equation. The aim was either to explain a resultant piece of music by reconstructing the intention of the composer, or to explain human creativity by building a model of the mind.

Our aim instead is to understand composition as an irreducible unfolding of material traces which takes place in its own temporality. This understanding is formalised as a software framework that traces creation time as a version graph of transactions. The instantiation and manipulation of any musical structure implemented within this framework is thereby automatically stored in a database. Not only can it be queried *ex post* by an external researcher—providing a new quality for the empirical analysis of the activity of composing—but it is an integral part of the composition environment. Therefore it can recursively become a source for the ongoing composition and introduce new ways of aesthetic expression. The framework aims to unify creation and performance time, fixed and generative composition, human and algorithmic “writing”, a writing that includes indeterminate elements which condense as concurrent vertices in the version graph.

The second major contribution is a critical epistemological discourse on the question of observability and the function of observation. Our goal is to explore a new direction of artistic research which is characterised by a mixed methodology of theoretical writing, technological development and artistic practice. The form of the thesis is an exercise in becoming *process-like* itself, wherein the epistemic thing is generated by translating the gaps between these three levels. This is my idea of the new aesthetics: That through the operation of a re-entry one may establish a sort of process “form”, yielding works which go beyond a categorical either “sound-in-itself” or “conceptualism”.

Exemplary processes are revealed by deconstructing a series of existing pieces, as well as through the successful application of the new framework in the creation of new pieces.



# Contents

<b>Abstract</b>	<b>v</b>
<b>Acknowledgements</b>	<b>xix</b>
<b>Author's Declaration</b>	<b>xxi</b>
<b>Typographic Conventions</b>	<b>xxiii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Background Story . . . . .	1
1.2 Motivation . . . . .	6
1.3 Objective . . . . .	8
1.4 Thesis Overview . . . . .	10
<b>2 Two Layers of Time</b>	<b>15</b>
2.1 The Double Nature of the Compositional Process . . . . .	15
2.2 Outside of Time? . . . . .	19
2.3 The Separating Diaphragm . . . . .	24
2.4 Creation Time . . . . .	36
2.5 Accessing Time . . . . .	39
2.6 Access Methods . . . . .	47
2.7 Branching and Multiplicities . . . . .	52
2.8 A Comprehensive Model of the Compositional Process . . . . .	57
2.9 Conclusion . . . . .	60
<b>3 Beyond Control and Communication</b>	<b>63</b>
3.1 Composers in Control . . . . .	65
3.2 Models . . . . .	78

3.3	Notes from the Metalevel . . . . .	82
3.4	Injection . . . . .	90
3.5	Resolution . . . . .	102
<b>4</b>	<b>Traces</b>	<b>107</b>
4.1	Introduction . . . . .	107
4.2	Dissemination . . . . .	109
4.3	That Which Does Not Become Systemic . . . . .	128
4.4	Sound Similarity as Transversal Reading/Writing across Pieces . . . . .	130
4.5	Exploiting Graphemes . . . . .	165
4.6	Indeterminus . . . . .	171
4.7	Summary . . . . .	189
<b>5</b>	<b>Design and Implementation of a Tracing System</b>	<b>191</b>
5.1	The Programming Language . . . . .	191
5.2	Framework Overview . . . . .	193
5.3	System Façade and Transactional Encapsulation . . . . .	196
5.4	Durability . . . . .	207
5.5	Confluent Semantics . . . . .	214
5.6	Building a Confluent System . . . . .	229
5.7	Extensions and Alternatives to Persistence . . . . .	252
5.8	Event Processing . . . . .	258
5.9	Composable Expressions . . . . .	269
5.10	Performance Time . . . . .	272
5.11	Creating Sound Processes . . . . .	277
5.12	Editing Sound Processes . . . . .	285
5.13	Summary . . . . .	286
<b>6</b>	<b>Conclusions</b>	<b>289</b>
6.1	Discussion . . . . .	290

6.2	Process of the Thesis . . . . .	293
6.3	Contributions . . . . .	303
6.4	Limitations . . . . .	307
6.5	Recommendations for Future Work . . . . .	308
<b>7</b>	<b>Afterword</b>	<b>313</b>
<b>A</b>	<b>Contents of the DVD</b>	<b>321</b>
<b>B</b>	<b>Survey of the Scala Programming Language</b>	<b>323</b>
B.1	Basic Syntax and Types . . . . .	324
B.2	Scoping and Nesting . . . . .	327
B.3	Functional Aspects . . . . .	328
B.4	Type System . . . . .	338
B.5	Concurrency Abstractions . . . . .	347
B.6	Summary . . . . .	349
<b>C</b>	<b>Record of Activities</b>	<b>351</b>
	<b>List of References</b>	<b>357</b>



# List of Figures

2.1	Diaphragm model of composition . . . . .	25
2.2	A patch in <i>Pure Data</i> producing a sequence of tones . . . . .	26
2.3	Audiovisual installation <i>Command Control Communications</i> . . . . .	32
2.4	Screenshot from the timeline editor of <i>CCC</i> . . . . .	33
2.5	Possible evolution of a <i>PD</i> patch . . . . .	38
2.6	Transaction time database . . . . .	41
2.7	Schema of a multi-track editor . . . . .	43
2.8	Schema of a bitemporal data structure . . . . .	45
2.9	Decoupled creation and virtual performance time . . . . .	46
2.10	A binary search tree . . . . .	48
2.11	Modelling bitemporal data as an R-tree . . . . .	51
2.12	A branched and temporal structure . . . . .	54
2.13	Simple model of the compositional process . . . . .	56
2.14	Relationships between the two time layers . . . . .	58
3.1	Test-cycle based models . . . . .	66
3.2	Regulation and control (Ashby) . . . . .	69
3.3	Control system (Heylighen and Joslyn) . . . . .	72
3.4	Synthesis process model (Collins) . . . . .	76
3.5	Observer roles in the thesis project . . . . .	83
3.6	Observer-defined levels of thinking (Checkland) . . . . .	94
3.7	Computer based tracing of the compositional process, and the limits of control . . . . .	99
3.8	Bifurcation opening a space for the dance of material traces . . . . .	104
4.1	Phase model of relations between writing and reading . . . . .	108
4.2	<i>Natural Palimpsest</i> and <i>Kalligraphie</i> . . . . .	110

4.3	Sound installation <i>Amplifikation</i> . . . . .	111
4.4	Model of the installation <i>Dissemination</i> . . . . .	112
4.5	Exhibition photos from <i>Dissemination</i> . . . . .	113
4.6	Sketching out variants for a new glass plate based installation . . . . .	114
4.7	Installation <i>Zelle 148</i> . . . . .	116
4.8	Schematic of <i>Kalligraphie</i> . . . . .	118
4.9	Schema of the constituent sound processes in <i>Dissemination</i> . . . . .	119
4.10	Score rendering of <i>Dissemination</i> . . . . .	120
4.11	External references for <i>Dissemination</i> and their establishment over time . . . . .	121
4.12	Code commits to the <i>git</i> repositories of the composition of <i>Dissemination</i> . . . . .	124
4.13	Sketchbook note about Derrida's 'Signature Event Context' . . . . .	124
4.14	Momentary image of sound processes in <i>Dissemination</i> . . . . .	126
4.15	Wolkenpumpe live patching environment . . . . .	127
4.16	«Who is producing the sound?» . . . . .	131
4.17	Construction diagram of the initial canvas of <i>Residual</i> . . . . .	133
4.18	Translating a whole sound file to the Fourier domain in <i>FScene</i> . . . . .	134
4.19	A quotation from Feldman I put down in a diary . . . . .	135
4.20	Plan for the construction of <i>Zeichnung</i> . . . . .	138
4.21	Sketch illustrating the gradual change of form due to imperfect imitations . . . . .	139
4.22	Timeline arrangement of imitation no. 6 . . . . .	139
4.23	Form plan showing the timeline of the four materials . . . . .	141
4.24	Typology of transitions in <i>Zeichnung</i> . . . . .	141
4.25	Different ways of preparing the strings of <i>Inter-Play/Re-Sound</i> . . . . .	143
4.26	The piano equipped with transducers during the development in the ICCMR lab . . . . .	144
4.27	Tendency mask . . . . .	144
4.28	Screenshot from <i>Inter-Play/Re-Sound</i> . . . . .	145
4.29	Example selection in process TOUCH . . . . .	147
4.30	Photo of the <i>Writing Machine</i> sound installation . . . . .	148

4.31	Diagram of the <i>Writing Machine</i> algorithm . . . . .	149
4.32	Iteration in constructing the first half of the first part in <i>Leere Null</i> . . . . .	154
4.33	Screenshot with the user interface for <i>Leere Null</i> . . . . .	155
4.34	Original concept of the two strategies in <i>Leere Null</i> . . . . .	157
4.35	Two possibilities of exploring a grapheme . . . . .	166
4.36	Wide shot and details of <i>Voice Trap</i> . . . . .	167
4.37	Excerpt from the version graph of <i>Voice Trap</i> . . . . .	168
4.38	Paper installation <i>Dots</i> . . . . .	169
4.39	Sound piece <i>Unvorhergesehen–Real–Farblos</i> . . . . .	171
4.40	Screenshot of the <i>(Inde)terminus</i> session in <i>Mellite</i> . . . . .	173
4.41	Iteration and recursion scheme of <i>(Inde)terminus</i> . . . . .	173
4.42	Detail screenshots of <i>Mellite</i> . . . . .	174
4.43	Planetarium Sternenturm Judenburg . . . . .	177
4.44	“Punch cards” of weekly times at which work on the compositions was done . .	178
4.45	Sequence in $\mathcal{T}_\kappa$ of sound files introduced to <i>Machinae Coelestis</i> . . . . .	179
4.46	Notes regarding specific parts of the field recordings used . . . . .	179
4.47	Frequencies of <i>Mellite</i> tool actions . . . . .	181
4.48	Distribution of the amount of contraction and expansion in resize actions . . . .	183
4.49	Distribution of the relative time shift in move actions . . . . .	184
4.50	Evolution of the distribution of audio region durations . . . . .	186
4.51	Motiongram for <i>(Inde)terminus</i> . . . . .	188
5.1	Architectural diagram of the framework . . . . .	195
5.2	Encapsulating variables, transaction context and executor in our own API . . . .	203
5.3	Different attempts to describe associated types . . . . .	208
5.4	Linked list data structure, and example of its traversal . . . . .	211
5.5	Subsequent key-value access in the data store for the traversal of list 3,5,8 . . .	211
5.6	Coupling in-memory STM and database transaction . . . . .	215
5.7	Directed acyclic version graph describing the evolution of a linked list . . . . .	220

5.8	Three instances of a node characterised by distinct paths from its seminal version	221
5.9	Illustration of path compression . . . . .	225
5.10	Measure type class for Finger Trees . . . . .	234
5.11	Finger Trees representing compressed paths . . . . .	236
5.12	A sub-tree with marked vertices . . . . .	238
5.13	Interpreting ancestor lookup as a two-dimensional nearest neighbour search . .	241
5.14	Querying a field $F$ by locating the query vertex $u$ in the mark tree . . . . .	246
5.15	Skip octree consisting of a full and one subsampled tree . . . . .	247
5.16	A skip list storing the marked vertices is used to control octree decimation . . .	250
5.17	Types of equidistant paths . . . . .	252
5.18	Pathological configuration violating the NN performance bounds . . . . .	253
5.19	Problem of operating retroactively on branched elements . . . . .	257
5.20	A version is retroactively inserted after a parent . . . . .	258
5.21	Push and pull phase in event dispatch . . . . .	268
5.22	Expression chains . . . . .	271
5.23	Interaction between scans, graphemes and graph functions . . . . .	281
6.1	Work on the software framework over time . . . . .	295
6.2	Excess of context . . . . .	303
B.1	Type hierarchy of <i>Scala</i> . . . . .	339

# List of Tables

3.1	Goal taxonomy of Pask . . . . .	75
4.1	Cross-correlations in the matching cost function of <i>Leere Null</i> . . . . .	157
4.2	Conceptual pairs of “condensation” . . . . .	161
5.1	System independent abstraction for object identifiers and access . . . . .	205
5.2	Fat field entries for the head and head fields . . . . .	224
5.3	Transaction local cache for ongoing write operations . . . . .	232
B.1	Common abstractions in the <i>SuperCollider</i> language and <i>Scala</i> . . . . .	325
B.2	Type system abstractions in <i>Scala</i> . . . . .	347



# List of Listings

5.1	Type members of the system abstraction . . . . .	206
5.2	Parametrising an object with a system and interaction of its type members . . . .	206
5.3	The transaction context provides <code>newID</code> and <code>newVar</code> to instantiate <code>S#ID</code> and <code>S#Var</code> . . . . .	207
5.4	Variable (mutable cell) . . . . .	207
5.5	Abstract interface of a key-value store for serialised data . . . . .	210
5.6	Implementation of a durable integer cell . . . . .	210
5.7	Factory methods for instantiating a list and its cells . . . . .	232
5.8	Creating a confluent persistent system initialised with a list of two cells . . . . .	232
5.9	Interface for querying and updating in a version tree . . . . .	244
5.10	Algorithms for querying and updating in a version tree . . . . .	245
5.11	Event definition and composition in <i>EScala</i> . . . . .	261
5.12	Reactive signals with <i>Scala.React</i> . . . . .	263
5.13	Events in an observed expression . . . . .	266
5.14	Engaging event chains and mapping updates . . . . .	267
5.15	Expressions, and their special forms variable and constant . . . . .	270
5.16	A bi-temporal expression associates a magnitude with a point in time. . . . .	272
5.17	<code>BiPin</code> , a bi-temporal breakpoint function . . . . .	273
5.18	<code>BiGroup</code> , a bi-temporal interval tree . . . . .	276
5.19	Interface of a sound process . . . . .	277
5.20	Example <code>SynthGraph</code> generating pink noise . . . . .	278
5.21	Configuring parameters of a sound process . . . . .	279
5.22	The <code>Transport</code> interface scans a group of processes in $\mathcal{T}_p$ . . . . .	282
B.1	Skeleton of a purely functional lazy stream . . . . .	332
B.2	Using an implicit argument list for the transaction context . . . . .	336
B.3	Path-dependent types . . . . .	346

B.4	Type projections . . . . .	347
B.5	Dataflow programming with <i>Akka</i> . . . . .	350

# Acknowledgements

I would like to thank everyone involved with or affected by this thesis for their patience and support. First and foremost, my director of studies Prof. Eduardo Miranda who kept giving me confidence throughout the process, and Prof. Gerhard Eckel who agreed to be my external supervisor. I am grateful to the Graduate School of the University of Plymouth for investing me with a scholarship that allowed me to concentrate full time on the thesis.

On my path towards the dissertation, I would like to thank my parents for always supporting me, no matter what decisions I took; Folkmar Hein for the great time I had at the Electronic Studio of the TU Berlin, and Robin Minard for sharing the trust in art; during the research period, my colleagues at the ICCMR in Plymouth, and also my colleagues at the computer music research group of the IEM Graz who eventually became my coworkers.

Regarding the realisation of sound pieces, I would like to thank Reni Hofmüller for the repeated invitations to gallery ESC Graz, Peninsula Contemporary Music Festival directors Simon Ible and Eduardo Miranda for their invitations to participate, and Martin Bricelj and the team from the Museum of Transitory Art for the invitation to festival SONICA in Ljubljana. I thank the State of Styria, the Institute for Music and Acoustics of the ZKM Karlsruhe, and the City of Judenburg for providing me with artist residencies to develop some of these works.

I am indebted to Kate Howlett-Jones for proof reading this text.

Above all, my appreciation goes to my wife and partner Nayarí for her unconditional love and immense patience, especially during the write-up phase which was marked by difficult periods of non-existent social life.

Finally, I would like to mention the moral support given by my cat Lucrecia. . .



# Author's Declaration

At no time during the registration for the degree of Doctor of Philosophy has the author been registered for any other University award without prior agreement of the Graduate Committee.

This study was financed with the aid of a studentship from the University of Plymouth Graduate School. Part of the practical work was developed through residencies funded by the State of Styria, the ZKM Karlsruhe, and the city of Judenburg, Austria.

Relevant scientific seminars and conferences were regularly attended at which work was often presented. A record of activities, including publications and presentations is found in Appendix C.

**Signed:** \_\_\_\_\_

**Date:** 14/08/2014

Word count for the main body of this thesis: 78923



# Typographic Conventions

- «quote» A direct quotation from a cited author. The reference usually follows directly after the closing marks, unless the citation source is obvious from a preceding reference. Ex. Derrida used the term precisely to denote «disengaging from the concept of polysemics».<sup>footnote</sup>
- “paraphrase” Creates distance; indicates either a paraphrased quotation, an unusual word usage or irony. Ex. This is what I have called “limits of control”. The sound recordings were picked up in “nature”.
- ‘concept’ Indicates either a term specifically introduced by an author or the fact that the sentence objectifies a concept. Ex. The processes are inspired by the metaphor of ‘dissemination’.
- emphasis* Stresses a specific word and makes it stand out visually. Ex. There is a *strangeness* in the relation between composing time and performing time.
- Title* Title of a piece of music, artwork, software library or application, a book or paper. Ex. *Terminus I, SuperCollider, Formalized Music*.
- NODE Reference to a node in a diagram or to an algorithm. Ex. To survive in an evolving world the variability of TEST demands an input to ACTION REPERTOIRE.
- code Inlined source code or name of a software routine. Ex. Cursors are regular variables containing the current paths, S#Var[S#Acc].



# Chapter 1

## Introduction

This chapter begins by introducing the background layer in front of which the thesis emerged. It names a number of dispositions that informed the work, such as the relationship between composer and machine, and a number of constants that pervade it, such as focusing on the negative structure of “gaps” which lie at the centre of processes. After a clarification of my motivation in the project, the overall research aim and objectives are stated. The last part presents the structure of the main chapters.

### 1.1 Background Story

When I am asked what my work is about, I tend to give a summary answer, usually enumerating that I compose electroacoustic music, that I make sound installations, that I do laptop improvisation, that I am a researcher in computer music, and that I develop music software. Depending on the context, I will put one item or the other at the beginning of the list or I will silently omit some from the list. It appears to be difficult to give them all a unifying frame or to please everyone. This heterogeneity is also a centre of this thesis, which originally set out to produce such a unifying frame, or one could say a stool so that I would not be caught between two stools any more.

My understanding of “making” music has always been very haptic. In my parental home, there was a piano and I think I tried to learn playing the piano at least three times in my childhood and adolescence, each attempt being ultimately frustrated by my impatience with practising the same material over and over again. I had an obstinate relationship with this instrument that resisted my will. However, I loved the sound that one could produce by striking the strings directly inside the corpus, perhaps using small objects to alter the timbre.

In my teenage years, I was involved with a group of friends running a show on the local community radio. Part of this activity was producing what could be called a naive form of “ars acustica”. With a microphone and cassette recorder from the broadcasting station I would go and collect different sounds from outside, then produce a sort of narrative—although not using spoken text—by assembling the different pieces. This was accomplished by using two tape recorders and manually pressing record and play on each of them to transfer bits of sounds from the source to the target tape. I would also try to physically cut and splice tape, something that was not very successful due to the fragility of the 4mm compact cassette format. I subjected the tape to heat, magnetism and other forces, in the hope that this would produce interesting alterations of the sound. That was also not very successful, usually resulting in the tape simply being destroyed or becoming silent.

There followed a period where I was saving money to buy a “music workstation”, one of the huge 1990s digital synthesisers which promised that you could run an entire music production with it. It had a tiny liquid crystal display and a dozen push buttons which led you to the depths of five or six levels of sub menus. The sound palette consisted of a few hundred fixed sampled waveforms which could be slightly manipulated, filtered and processed. For more money you could buy sound expansion boards, an option I never considered. The “notes” had to be recorded and overlaid with the claviature. This was also not very successful.

It was only with a digital sampler, a DAT recorder, a microphone and a desktop computer—I don’t recall the sequence in which these pieces came together—that some time between 1998 and 1999 things began to make sense. Concrete sound was a real material, it was something that could be captured and manipulated beyond recognition. I changed historiography for communication science and began to study at the Electronic Studio of the Technical University Berlin, becoming acquainted with the history and practice of electroacoustic music and learning about sound installation art. But the official design of the programme was foremost one of engineering: it included sound engineering and acoustics, digital signal processing, and a small amount of computer science as well. I had spent a substantial part of my childhood in front

of computer screens, although never in relation to music, and now the two came together. Motivated by my training in signal processing, I began to write my first music software *FScape*. But I also developed a separate thread by making friends in Berlin's electronic underground and noise music scene. My favourite project from that time was based on tiny electronic feedback circuits driven by a current generated with solar cells. We would spend days in soldering fumes, trying to find new sounds by using new combinations of components and integrated circuits. In a way, this was not unlike me cutting the compact cassette tape and submerging it in coke, but it was sonically much more convincing.

Why am I starting the text with this lengthy account of things which happened fifteen to twenty years ago? I find in it three dispositions, two general and one personal, which frame the thesis, as well as three aspects that will act as constants for the construction of the thesis.

### 1.1.1 Dispositions

First of all, the anecdote shows that there is only a very short history of general access to digital and computer-based music technology. While the paradigm of operating on stored sound material goes back to the beginnings of *musique concrète* and earlier, the ubiquity of personal computers is recent and the effects of having electroacoustic music and computers collide have scarcely figured as a subject of research. We have all sorts of tools which a contemporary electroacoustic composer or a sound artist working with electronic or electroacoustic sounds can use, but we do not have thorough concepts of these tools, other than those related to signal processing or user interfaces. We also have little knowledge of the «profound influence of computer science»<sup>1</sup> on this kind of music or the extent to which software interacts with the creative process.<sup>2</sup>

Secondly, we find ourselves in a permanent *struggle* with the technology. We may have a certain idea of what we are trying to achieve when using it, but it will always introduce a shift in what is being produced which is inherent in the machine and not under control—neither the control

---

<sup>1</sup>Gareth Loy and Curtis Abbott (1985), 'Programming Languages for Computer Music Synthesis, Performance, and Composition', *ACM Computing Surveys (CSUR)* 17(2), pp. 235–265.

<sup>2</sup>Barry Eaglestone et al. (2001), 'Composition Systems Requirements for Creativity: What Research Methodology', in: *Proceedings of Mosart Workshop on Current Research Directions in Computer Music*, Barcelona.

of the composer nor the control of the designer of the machine. It is often assumed that the machine embodies a “service”, with ideas on two ends of a spectrum. On one end, an engineered algorithm perfectly adhering to a design specification versus, on the other end, the idea of a perfect model of the human cognitive capabilities that can be implanted into a machine which thereby becomes transparent “support”. These ideas need not be dismissed, but they ignore the possibility that an experimental system arises from the friction between human composer and machine.

This is my next point: the anchorage of composition in an experimental situation primarily driven by its material embedding rather than a logical or symbolic foundation. This must not be mistaken for a preference of the bottom-up or inductive over the top-down or deductive perspective. One can have formal ideas which do not derive directly from particular givens, yet there will be a moment when they start to collide with the proper dynamics of the materials introduced into the piece and of the machines with which they are handled. One needs to gain experience in using these machines, repeat procedures over and over, to channel these dynamics, but the outcome will still not be something controlled by an external source.

### **1.1.2 Constants**

My preoccupation with composing, improvising or writing software has always favoured the process of the making over the regard of the piece or product that stands at its end. This seems hardly surprising, as the “work” of a composer, researcher or programmer of course *is* the construction. But it goes farther in that I often find myself defining a piece based on a set of procedures I want to try out, rather than a specific anticipated structure that stands at its end. Naturally, a *setting* is given; for example, when I plan to make a sound installation in a particular space, the space and the situation are given, and you have to write a proposal or outline of what you are going to do, which materials and resources you will need to allocate, etc. But they remain hypotheses for the sake of getting started. If there is a form plan, it is just a frame inside which the experimentation happens.

This preoccupation is reflected in my perception of (other artists’) pieces as well. As audience, I very much prefer to visit a sound installation than to attend a concert. I explain this by the

possibility of non-theatricality which allows me to define my own unrolling of a piece or situation in time. There must always be an element of non-narrativity. I observe a somewhat opposite motion: there is a tendency towards *more* theatricality, more staging, more narrativity in sound art which remains foreign to me. For instance, a keynote speaker at the 2012 International Computer Music Conference was artist and musician Seth Kim-Cohen, who with his book title *In the Blink of an Ear* appears to be picking up Marcel Duchamp's concept of 'non-retinal art'. We seem to be at a point where sound-in-itself is perceived as lagging behind the discourse in the fine arts. The remedy is seen in a conceptualisation of sound art. We have, for example, at that same conference the performance of Johannes Kreidler's work *Fremdarbeit*, based on the idea of the composer subcontracting a low-wage labour force in Asia to do the dirty work of composing.

While one might be stunned by the cleverness of these conceptions, the cheap sensation of shock cannot, in my opinion, compensate for a lack of self-navigation of the audience. It is therefore clear that by asking for the possibility of an autonomous traversal of a work, I am precisely not allotting this space to the quest of retracing the artist's intention or grand plan. Instead I am interested, both as audience and composer, to become *entangled* in the temporal, material and conceptual structure of a work. I am not opposed to narrativity as such, but I believe it is something that emerges from this entanglement and is something essentially constructed by the recipient and not the composer who can only interfere by offering a mesh of fragmented micro-narratives.

To give a simple example of this entanglement: My first encounter with sound art, at least according to my leaky memory, was with a work by Rolf Julius in the Weserburg museum in Bremen, near the place where I grew up. I cannot recall exactly which work it was, but it was a sound installation inside a narrow severed corner, perhaps with small sounds emanating from a pile of objects on the floor. I always enjoyed going to that museum because of its rigorous industrial atmosphere, a very quiet place except for your own footsteps, yet open and airy. Finding these tiny sounds among this quietness—maybe there was the sound of the kinetic machines of a Kienholz environment or from a Rebecca Horn swing—was a unique experience.

In hindsight it mixes with the effort of taking the bus from my hometown to travel there (I did not have a driver's licence yet). It also gets mixed up with things that happen later, for example seeing Terry Fox's *The Eye is Not the Only Glass that Burns the Mind* in Worpswede, probably merely due to its regional proximity, or maybe because this was in autumn 2011, not so long after Julius died. No matter what the reasons are, the important factor is that these coherences are outside the control of the artist. My hypothesis is that just as the essence of an artwork's reception lies in the process of its traversal, the same goes for the production of an artwork. There is a traversal undertaken by the composer, in conjunction with an apparatus, which to a great extent determines the "excess"<sup>3</sup> from which the work lives.

The second constant is the importance of gaps. There are gaps between the involved disciplines, their language and methodology, gaps between the way time is expressed in the different forms of sound art, a gap between the composer and the apparatus, a gap between the designer and the apparatus. Gaps between a piece and the next piece by the same composer, and so on. Another hypothesis is that what we are trying to observe, 'process', is animated mainly by these gaps.

The third constant is the fact that I am not subtracting myself from the equation, I will remain in it all the time. The original reason for this was that, situating this thesis more in a self-reflective notion of artistic research, my aim was to analyse and formalise my own work. During development, however, this point became less important compared to the possibilities of self-observation as a method not aimed at recreating myself as the artist subject, but enabling me to look very closely at the differential drift in writing processes, which will be outlined later in this chapter.

## 1.2 Motivation

My original motivation in starting the thesis project did not name the compositional process as its central theme yet. Instead I became more and more interested in using algorithmic ideas in my work which incorporated some sort of randomness or indeterminacy. I observed a gap between the sophisticated tools we have for crafting electroacoustic compositions, multitrack editors and sound transformation tools, and tools typically employed when there is interaction

---

<sup>3</sup>I finally come back to this term in Sect. 6.2.2.

or a non-linear time structure, such as *Max*, *Pure Data* or *SuperCollider*. The outcome of using the former are monolithic pieces with no possibilities of algorithmic elements which have not been fixed in advance (and which usually must be computed outside the application), while it is very hard to compose larger forms with the latter, often ending up in “live instruments” which still have to be played and enriched to produce pieces. The concept of a sound artefact is not a given in these sound *synthesis* systems, but must be emulated.

On the other hand, we find algorithmic composition, an area in which the acquisitions of computer science are exploited for the compositional practice. There is only one problem: These systems are made for the manipulation of symbols, something that can be readily applied to notated instrumental music but less so to electronic or electroacoustic music. Furthermore, the formalisation here stands at the beginning of the process rather than at its end. Putting these two worlds together is still a challenge. In addition, others have noted that «generative systems have ... tended to be limited to symbolic representations ... as opposed to audio».<sup>4</sup> The following statement from Eduardo R. Miranda is a good illustration of the gap:

«I do find beauty in algorithmic processes, but I often find their musical rendering somewhat frustrating. It is often the case that algorithmic music processes are more appealing than their actual outcome. At the end of the day my compositional methods often boil down to GOFEM (Good Old Fashioned Electroacoustic Music) practices.»<sup>5</sup>

So my initial question was: What is necessary to bridge electroacoustic music and generative sound art, how does one bring algorithmic thinking into the two, when the objects with which we are operating can hardly be conceived as symbols in a formal language? What is the specific characteristic of working with concrete sound? What happens to the convenient timeline canvas offered by tape composition software, if elements and decisions become indeterminate?

---

<sup>4</sup>Arne Eigenfeldt and Philippe Pasquier (2011), ‘Negotiated Content: Generative Soundscape Composition by Autonomous Musical Agents in *Coming Together: Freesound*’, in: *Proceedings of the 2nd International Conference on Computational Creativity*, Mexico City, pp. 27–32.

<sup>5</sup>Eduardo R. Miranda (2009), ‘Lovely Algorithms, Hot Weather and Uninspiring Solfeggio’, *Contemporary Music Review* 28(1), pp. 120–121.

For a while I was occupied with trying to think of a *representation* of musical time when linearity is taken away. It seemed to be something that was in the air: In 2009, I saw the exhibition on notation<sup>6</sup> at the ZKM Karlsruhe, a collaboration with Berlin's Academy of Arts. A year after, I got hold of a beautiful book called *Cartographies of Time*.<sup>7</sup> Thanks in particular to the former, my attention moved away from the endless possibilities of conceiving mappings of time and towards the idea that all these fantastic exhibits—ranging from dance to architecture to video art to literature—were connected by the idea of writing processes that produced them.

Thus, if we could find a way to unify the understanding of the non-linear unfolding of time in a generative piece and the way one operates on the timeline of tape composition, both would become possible renderings within an overall framework of computer music composition. At this point it became clearer that what I would be looking at was the process of composition, understood as a decision-making process involving both human and computer and occupying its own temporal field.

### 1.3 Objective

The central problem of this thesis is to find a perspective on computer-based sound art that advances its aesthetics. An approach is sought that, on the one hand, goes beyond the reduction of “sound-in-itself”, a discourse which has dominated electroacoustic music since the 1950s and which emphasises the transparency of the tools of production, in fact the transparency of anything that stands in the way of the purity of the sound surface and its phenomenological and/or narrative context. On the other hand, the solution is not to propose an intellectualised “non-cochlear” approach, a reheated version of 1960s and 1970s conceptual art.

The aim then is to stake out the contours of a new aesthetic grounded in a reflective engagement with the compositional process and its conditions, however understanding these from their material traces—intimately connected both to the apparatus used in the composition and the

---

<sup>6</sup>Hubertus Amelunxen, Dieter Appelt and Peter Weibel, eds. (2008), *Notation: Kalkül und Form in den Künsten (Catalog)*, Berlin: Akademie der Künste.

<sup>7</sup>Daniel Rosenberg and Anthony Grafton (2010), *Cartographies of Time: A History of the Timeline*, New York: Princeton Architectural Press.

actual synthetic or recorded sounds one works with—and not from a linguistic or cognitive point of view.

The objectives derived from this aim are:

- › to review and critique existing concepts and methods of tracing and understanding compositional process
- › to develop an understanding of the temporal unfolding of the compositional process and its relation to time as it appears to us in a performance
- › to find a suitable representation of this composition time and to establish the limits of this representation
- › to develop a software framework which implements this representation along with a set of abstractions that allow a broad palette of computer-based sound art to be realised, ranging from tape composition to generative sound installation
- › to analyse existing pieces and to develop new pieces subject to questions of traceability and reflectivity of the compositional process
- › to demonstrate the interweaving of software development, artistic practice and theoretical reflection as a novel methodology for the emerging field of *artistic research*

The last item needs some explanation. We argue that due to the difficult nature of observing process—which will be elaborated during the course of the thesis—we require a heterogeneous methodology oscillating between these three layers of the discourse. There are two motivations behind this. First, as Niklas Luhmann points out in a discussion of deconstructivism: «Given the narrowness of academic citation circles, there are many possibilities of cross-fertilization that remain unused.»<sup>8</sup> In this regard, the methodology is an attempt to bring artistic and scientific thought together in a new way. Second, we propose to work with the concept of ‘experimental systems’ coined by Hans-Jörg Rheinberger, because it is compatible with many aspects of

---

<sup>8</sup>Niklas Luhmann (1993), ‘Deconstruction as Second-Order Observing’, *New Literary History* 24(4), pp. 763–782.

artistic production, such as foundation in material traces, a vagueness of the ‘epistemic thing’ (that which we wish to *know* or *experience*) and the situation of ‘extimacy’ in which the researcher finds herself or himself. We are subjecting the thesis itself to differential iterations—producing both coherence and gaps between the three layers—since this way we create a true recursion which avoids the problem of vertical hierarchy and lack of connectivity, as discussed in Sect. 3.3; the re-entry has a distancing effect (disruption, self-observation of the speaker) which only just allows the traces to appear, as everything is *duplicated*—or, as Jacques Derrida says: Deconstruction has to happen from the inside.

## 1.4 Thesis Overview

The thesis is made up of three products: This text, a software framework, and a number of sound pieces. Software and sound pieces are documented on the accompanying DVD, the contents of which are described in Appendix A. Even though I consider all three media essential for the success of the thesis, for its appraisal this written text should be the main source. It includes the discussion and analysis of the sound pieces to the degree that is considered relevant with respect to the objectives, and the same goes for the presentation of the design and implementation of the software framework.

The written thesis is structured along four chapters which are complementary to each other. Although each chapter assumes familiarity with the preceding chapters, they can also be read independently. The reader will encounter snippets of programming code and patches. It is not always necessary to understand these codes in great detail in order to follow the ideas being discussed. Generally, code examples are only given when their detailed understanding is required to follow the thesis, most noticeably in Chap. 5.

The following paragraphs give an overview of the chapters. For each chapter, a “word cloud” was generated which convenes the most important terms and their relative frequencies, thereby giving an alternative key to the form of the chapters.









## Chapter 2

# Two Layers of Time

The purpose of this chapter is to understand what the compositional process is, and to reach out for computational models which could be exploited to represent this process. Noting that process marks both activity and result, this double signification is studied in the conceptual history of the term and its concrete employment in the work of Iannis Xenakis and Pierre Schaeffer. Xenakis' opposition of "outside-time" and "in-time" stimulates a discussion of representations for musical time, and it is shown how the writing of this time is itself embedded in a different layer of time, creation time. Temporal methodologies in database research are reviewed, and the bitemporal model is identified with creation and performance time. The recognition of branching and concurrent movements in the process of composition challenges the view that a piece of music terminates this process and gives birth to a comprehensive model which may unite seemingly disparate forms such as tape composition and generative sound installation.

### 2.1 The Double Nature of the Compositional Process

Composer and computer music pioneer Gottfried Michael Koenig wrote an essay in 1978 titled *Composition Processes*. It is an interesting document since it outlines Koenig's understanding of the role of a composer, an understanding certainly shared by other composers, but which will also be challenged in the course of this text. Moreover, it begins with the observation of a peculiarity of the term 'composition' which acted as a *detonator* for the whole thesis project:

«By musical composition we generally understand the production of an instrumental score or a tape of electronic music. However, we also understand composition as the result of composing: the scores of instrumental or electronic pieces, an electronic

tape, even a performance (we say for instance: “I have heard a composition by composer X”). The concept of composition is accordingly closed with regard to the result, but open with regard to the making of a composition; it tells us nothing about preparatory work, whether it is essential for the composition or not.»<sup>1</sup>

It is this confusion between an activity and an observable entity which gave rise to the title of this section—the double nature of composition. It is born out of the following assumption: «Composing terminates in pieces». A “piece” is thus like a diaphragm between the process of composing and the reception of a performance, it makes both separable. From the quotation it is clear that Koenig assumes an asymmetry between the two sides, production and consumption. His responsibility as composer is the former, and it ends when the artefact, the score or tape, has been created. Indeed, as the essay evolves, there is the core of composing—«the intellectual act of invention»—and there are subordinate, auxiliary activities, such as the «sonic realization», which in the case of instrumental music may constitute the duty of others (musicians).

Before questioning the impermeability of the diaphragm, we shall follow Koenig first by discussing the compositional process as the production of a piece of music. This production is carried out with the help of computer programs. A program requires certain musical parameters as input, and will output a table of musical data which will be transcribed into a score or used for the realisation of a tape. Koenig observes a range of composer personalities forming a continuum from the «constructive» to the «intuitive» type. He notes that the former prefers to provide little input to the system and relies more on its ability to autonomously generate results from which he<sup>2</sup> picks the ones that match his taste, whereas the latter type has little trust in the system’s autonomy and tries to provide as much input and as many constraints as possible.

But there is otherwise very little detail regarding the process *as processuality*. Process in Koenig’s text is rather flattened to “procedure”. As such, it encompasses the definition of rules, the subsequent application of compositional methods, which then produces results (the

<sup>1</sup>Gottfried Michael Koenig (1978/1993b), ‘Kompositionsprozesse’, in: *Ästhetische Praxis*, vol. 3, Texte zur Musik, Saarbrücken: PFAU Verlag, pp. 191–210. The English translation of the originally German text is the authorised version, as published by Koenig on his website <http://www.koenigproject.nl> (visited on 02/06/2012)

<sup>2</sup>Whenever I refer to a composer in general, naturally this is gender neutral. Only for simplicity I have chosen to use male pronouns.

final or preliminary composition). If the composing program is put in the place of method application, the chain rules  $\rightarrow$  program  $\rightarrow$  results is obtained, and if ‘program’ is taken as a black box, the composition process is literally timeless. The rules do not reflect the time in which they have been formulated—they originate from an «act of invention», rather than a process of invention<sup>3</sup>—, the application of methods or the program consumes time only due to an inconvenient requirement (labour), and the results are transcribed into a chronological score whose temporal extension is independent of the time in which the transcription takes place.

### 2.1.1 Conceptual History of Process

Intuitively, we relate the notion of ‘process’ to its latin root, *procedere* meaning “to move forward”, and imply that this motion has a temporal extent. K. Röttgers has analysed the conceptual history of the term,<sup>4</sup> going backwards from its appearance in the work of Georg Wilhelm Friedrich Hegel at the beginning of the 19th century. Of the various meanings it had in the middle ages, that of technical procedure makes its way into alchemy. For instance, Paracelsus describes process as a work which is carried out according to rules (quite similar to Koenig’s usage). In early chemistry, it is still the good intention of the chemist and the careful regard of the sequence of steps which guarantee the success of a process. However, at the end of the 18th century the view prevails that nature has its own forces which dominate process, man is becoming less and less the actor. He does not make the rules but merely discovers them in nature. Process may have natural or artificial origins. The application of the word is slowly extended to other areas, such as “process of life”, especially with its new ability to be self-sustaining (autonomous).

In the natural philosophy of Friedrich Wilhelm Joseph Schelling, process and *organisation* form an interplay. Natural processes are permanent (their conditions exist continuously), they are thus “becoming”, they are indeed productivity, but never product. Novalis extends the use of the term into the cognitive realm (Denkerzeugungsproceß, thought-generating process), and Schelling finally postulates “first order processes” which no longer require any form of visibility—even

---

<sup>3</sup>This is reinforced by Koenig’s proposed approach of using a description of models: «... given [!] the rules, find the music»

<sup>4</sup>Kurt Röttgers (1983), ‘Der Ursprung der Prozessidee aus dem Geiste der Chemie’, *Archiv für Begriffsgeschichte* 27, pp. 93–157.

though they owe their supposition purely to the need to avoid a tautological explanation of the causes of the visible processes. According to Röttgers' interpretation, Novalis amplifies the process concept beyond the simple sequence of steps. Processes now possess *their own adequate temporal order* which must be respected, as otherwise a process is impurified. Furthermore, this temporal order may be expressed as *spatialiation*, as can be seen in organic growth which is just the articulation of time.

### 2.1.2 A Definition of Process

Before attempting our own definition of process, we seek a contemporary definition to start with. N. Rescher introduces *process philosophy*, a branch drawing a line from Heraclitus via Gottfried Wilhelm Leibniz to Henri Bergson, the American pragmatists, and finally Alfred N. Whitehead. Although the interest here is not so much in process philosophy or Hegel's idealism, a useful definition of process as such is given:

«A process is a structural succession of states of affairs which accordingly form a unified overall complex of terms connected by “and then.”»<sup>5</sup>

Rescher furthermore identifies three distinctive features of process: It consists of phases that are connected to form a complex. This complex establishes a temporal coherence, and the temporal dimension is irreducible. A process possesses an underlying formal structure or shape.

In the light of this differentiated view, little has been actually said about the compositional process. As previously noted, despite its title Koenig's essay somehow navigates around the process's complex and temporally cohesive character, merely adumbrating how one arrives at rules and applies them. He is interested in the modelling of the compositional process in software, but acknowledges that actual human behaviour may be very irregular, posing «greatest difficulties of representation in program structures». Indeed he gives a more accurate description of this behaviour:

«A composer is more accustomed to being influenced by a spontaneous idea than by prepared plans; he decides and discards, writes down and corrects, remembers

---

<sup>5</sup>Nicholas Rescher (2006), *Process Philosophical Deliberations*, Heusenstamm: ontos verlag, p. 2.

and forgets, works towards a goal; replaces it during his work by another—guided by criteria which are more likely to be found in psychology than in music theory.»<sup>6</sup>

Like Koenig, this thesis also does not look into the psychology of the composer's mind. Instead, the interest in process stems from the supposition that a model of this process may be used as a creative device in the formulation of a piece itself. A tentative definition at this point thus focuses on the creative aspect of process—*becoming*—instead of its termination in a product. A process is...

- › ... a succession of steps or phases of change...
- › ... which are not necessarily hierarchical, but follow a coherent and temporal pattern.
- › It may have been initiated by a concrete intention...
- › ... but ultimately it is upheld by its inner dynamic.
- › As long as the process is upheld, it is productive. Otherwise it terminates in a product.
- › The interaction with the process puts the observer in the role of an experimenter.<sup>7</sup>

## 2.2 Outside of Time?

We first restrict ourselves to “fixed” compositions. By this it is meant that the composer's responsibility indeed ends with an exhaustive notation prescribing the succession of sounds in a performance. For this, it does not matter whether that prescription takes the form of a “tape” or an instrumental score devoid of aleatoric elements. It goes without saying that in both cases, electroacoustic diffusion or reproduction by instrumentalists, a responsibility remains with the performance—it depends on many factors such as the acoustics, the dynamics of the audience, the speaker system and setup, the acousmatic interpretation, and so forth. However, the studying of *the composition* may be based solely on what is notated or on a recording of an arbitrary performance. This definition thus covers all the music Koenig was speaking about.

---

<sup>6</sup>Koenig, ‘Kompositionsprozesse’, p. 197.

<sup>7</sup>This proposition stems from the original alchemistic meaning of process.

In general, two motions can be distinguished: one using deduction to go from concept to material, the other using induction to go from material to concept. While this thesis is not about their work, of the composers in computer and electroacoustic music who have written extensively about their work, Iannis Xenakis (with *Formalized Music*, 1992) and Pierre Schaeffer (with *Traité des objets musicaux*, 1977) are perhaps good candidates to represent tendencies towards either of these directions.

For example, in the chapter *Free Stochastic Music*, Xenakis outlines the phases through which a musical work is constructed. Assuming the existence of some «initial conceptions» (ideas, provisional data), he then suggests a decision about the kinds of sounds used—more a choice of medium, such as a set of instruments, electronic generators, or granular construction—then prescribing successive decisions about macro structure (overall form and development in time) and micro structure (relationships between individual elements). In the next phase, macro and micro structures are “programmed”; finally, the programs are executed, yielding symbolic results which must then be translated into a form such as a score. The last phase is the sonic realisation, which may be an orchestral performance, a computerised sound synthesis or the construction of an electroacoustic tape.

These phases thus constitute a procedure similar to (although more detailed than) the description that Koenig gave. What is interesting about Xenakis here is that he makes a distinction between musical structures *outside-time* and musical structures *in-time* which he associates with the micro composition phase. Even though he adds that the order of the phases is not strict—relativising the perspective of a pure top-down construction—he maintains a hierarchy between outside-time and in-time structures throughout the book. Especially in chapters *Towards a Metamusic* and *Towards a Philosophy of Music* he emphasises the value of outside-time structures, which he claims have been neglected in recent Western music and conflated with in-time structures in serial music.

But what is a musical structure conceived outside time? Let us start with in-time structures. These relate to what Xenakis calls primary time, a time that «appears as a wax or clay on which operations and relations can be inscribed and engraved». Entities can either appear

simultaneously or successively, and in the latter case a notion of anteriority can be established. Thus, Xenakis argues, the rules which can be applied to in-time structures form an algebra in-time which is characterised by an asymmetry. For instance, if the anteriority relation is  $\top$ , with two distinct sounds  $a$  and  $b$  we clearly have  $a \top b \neq b \top a$ . In contrast, when forgetting about their temporal embeddedness, one can just notice that the two sounds are distinct, irrespective of their relative positions. An algebra outside-time is then symmetric with operator  $\vee$  denoting «put side by side without regard to time.» The operator is commutative, since  $a \vee b = b \vee a$  (as if  $a$  and  $b$  were put in a set or a “bag”).

Xenakis’ trick is to propose a model in which in-time, the «lexicographic» ordering of sounds, appears sandwiched between this reduced perception of sounds (just determining whether they are distinct or not) and another layer which he calls «metric» time. Metric time is the layer of relative durations. With three non-simultaneous elements one can establish the proportion in which they «divide time into two sections within the events», e.g.  $\underbrace{a \quad b \quad c}$ . Since only relative durations are considered, metric time is indeed part of outside-time construction. The three parts of the sandwich correspond with successive stages of child development according to Jean Piaget, and hence Xenakis implicitly defends his emphasis on outside-time structures on psychological grounds—the first layer is primordial, thus closer to the origin of man, the third layer requires the greatest ability of abstraction, thus is the purest.

Ultimately this is an *aesthetic* premise which one may or may not share. It would be easy to juxtapose it for example with Morton Feldman, who was interested «... in how Time exists before we put our paws on it, our minds, our imaginations, into it.»<sup>8</sup> The opposition of in-time versus outside-time should instead be reduced to a *functional* angle. From that angle, a non-psychological and aesthetically less partial interpretation could stress that outside-time on the one hand is a tool of abstraction which allows the building of structures with generalised transformations such as taken from set theory or relying on mathematical properties such as commutativity or associativity. On the other hand, it is also a strategy for partitioning the compositional process: The *spatialisation* of the musical material—because this is essentially

---

<sup>8</sup>Morton Feldman (1988), ‘Between categories’, *Contemporary Music Review* 2(2), pp. 1–5.

what the mental co-presence stated as elements «put side by side without regard to time» is<sup>9</sup>— corresponds with the opening of a temporal interval in the creation process. It is the interval in which the composer reasons about the sonic entities, constructs transformations, elaborates rules to be applied on them, and so forth. In a “fixed” composition that interval is only closed with the transition to the in-time layer, and it is only due to the complete detachedness of the in-time algebra from any temporal markings within this “interval of reasoning” that one can speak of the latter as an outside-time activity.

It appears that the process in the case of electroacoustic music is very different. Indeed Schaeffer begins by discriminating it from what he calls ‘a priori music’. A priori can be translated as deductive, where statements can be formulated that are not grounded in experience. Schaeffer criticises that musical constructions can thus be made from the postulates of arbitrary rules. The dismissal of this kind of rules is based on the concept of music as a communication process— what is desired is an understandable work, and therefore it must be founded on rules which can be ultimately rediscovered by the audience who only have their ears to decode the music, and therefore if the rules are disconnected from the aural perception they will remain obscure.

Composition is only half of the work in electroacoustic music if it is taken literally as synthesis— putting things together. Most of Schaeffer’s writing concentrates on the new preceding stage of analysis, an abstraction process only at the end of which an abstracted system of rules appears.<sup>10</sup> Under the premise of a successful communication, the *translation* process of the couple analysis– synthesis is “dangerous”: «... the chemist is never assured to succeed with a synthesis, since it does not derive with certainty from the analyses.»<sup>11</sup>

The use of the metaphor of a chemist is of course exciting here,<sup>12</sup> as it reveals an understanding of process which in Röttgers’ essay can be located in the work of Alexander Nicolaus Scherer

<sup>9</sup>cf. Bergson who goes even further, pointing out in ‘Time and Free Will’ that the conception of a temporal order already implies a spatial mental image (Henri Bergson [1910], *Time and Free Will, An essay on the Immediate Data of Consciousness*, trans. by Frank Lubecki Pogson, London: George Allen & Unwin, chapter II).

<sup>10</sup>‘Abstract syntax’ versus ‘abstracted syntax’ has been used as one of the two axes along which S. Emmerson locates the language of electroacoustic music, cf. Simon Emmerson (1986), ‘The Relation of Language to Materials’, in: *The Language of Electroacoustic Music*, ed. by Simon Emmerson, London: Macmillan, pp. 17–39

<sup>11</sup>«Ainsi le chimiste n’est-il jamais assuré de réussir une synthèse, qui ne saurait se déduire avec sécurité des analyses.» (Pierre Schaeffer [1966/1977], *Traité des objets musicaux, essai interdisciplines*, Paris: Editions du Seuil, p. 381)

<sup>12</sup>See also p. 418, where Schaeffer speaks of an alchemy of sound of which the experimenter is dreaming.

in 1795 and which marks the transition of chemistry to an empirical science. Schaeffer's whole programme is that of a musical *research* which goes far beyond the mere activity of composing; it is a "science of hearing", and it is based on *experimentation* and *observation* which take place in the *laboratory*.

On the technological level, though, Schaeffer emphasises a correspondence with optics, and compares his most important apparatus, the tape recorder, both to photography and microscopy. All contribute to the separation of a sensation from its transitory nature. The recorder provides «the ability to store, repeat, and freely examine sounds which were hitherto ephemeral, depending on the play of instrumentalists and the immediate presence of an audience».<sup>13</sup> But the subjection of sound to a meticulous and completely new observation—since the sound can be looped or slowed down—is not the only contribution of the recorder. For Schaeffer, it is its contribution to abstraction—removing the original context in which the sound occurred—which is more important, as it forms an integral part of his concept of 'objet sonore' based on structuralist and phenomenologist ideas of an abstract structure within which sound objects are embedded.<sup>14</sup> But just as in the Xenakis case, we must separate functional from aesthetical regard. Under the aspect of the dynamic nature of process, it is interesting to note that the fixation on tape produces an equivalent of the "outside-time" situation. The captured sound is the physical artefact adumbrating the abstract 'objet sonore', and, not unlike Xenakis' set theoretical considerations, may be subject to a set of "timeless" organising operations: «... we form collections of objects where we distinguish a sound criterion, and we find out if these objects, despite being disparate in their other criteria, will reveal relations of the particular criterion which make sense, that is to say can be qualified, ordered or located in our field of musical perception».<sup>15</sup> Outside-time again appears as spatial co-presence (ordering in the field of perception).

---

<sup>13</sup> «... de pouvoir conserver, répéter, examiner à loisir des sons jusqu'ici éphémères, liés au jeu des instrumentistes, à la présence immédiate des auditeurs», (ibid., p. 32)

<sup>14</sup> cf. Brian Kane (2007), 'L'Objet Sonore Maintenant: Pierre Schaeffer, sound objects and the phenomenological reduction', *Organised Sound* 12(1), pp. 15–24

<sup>15</sup> «... nous formons des collections d'objets où nous distinguons tel critère sonore, et nous cherchons si ces objets, malgré le disparate de leurs autres critères, feront apparaître des relations du critère considéré, qui aient un sens, c'est-à-dire qui soient qualifiables, ordonnables ou repérables dans notre champ perceptif musical», (Schaeffer, *Traité des objets musicaux*, p. 381)

### 2.3 The Separating Diaphragm

The inside-outside brings us back to the double nature of composition as formulated by Koenig. On the one hand, composition is “a piece”, the process’s result, an engraving of musical data associated with chronological (Koenig) or lexicographic (Xenakis) time. On the other hand, it is the process of production itself. The models of the latter seen so far all enumerate phases of this process, or at least present possible methods. Koenig names three methods: interpolation (from macro- to micro-structure), extrapolation (from micro- to macro-structure), and chronological-associative method (an ‘in-time’ anticipation of the temporal unfolding of the piece and linear construction of musical statements in this order). Xenakis presents various approaches, including the free stochastic already discussed (choice of medium and sound sources, macro structure, micro structure, programming, translation to score). Schaeffer distinguishes a conceptual and a technical process. The conceptual process—musical research—comprises the study of sound objects through a typological and morphological analysis of existing sounds, the study of musical objects through the development of musical criteria which organise sound objects, and finally the synthesis of musical objects, guided by these criteria and through the application of methods such as ‘variation’ and ‘texture’. The technical process—the electroacoustic chain—identifies all the phases from the generation of sounds (*factures sonores*), through their recording, to cutting and montage, to filtering and ‘modulation’.<sup>16</sup>

It has also been seen that there is a notion of musical data abstracted from the flow of time, whether defined as an outside-time algebra or a metric time in the case of Xenakis, or a recording of a sound in the case of Schaeffer, which can be voluntarily decomposed and re-assembled. This data is eventually positioned in lexicographic time (Xenakis) or copied to tape through a synchronised reading (Schaeffer calls *lecture synchrone* the mixing down of the materials as a set of synchronised tape tracks). For the sake of unifying terminology, we shall call this temporal succession the performance time, denoted  $\mathcal{T}_p$ . More precisely, this shall be the physical time<sup>17</sup> when the piece is actually sounding, whereas we speak of virtual or prospective performance

<sup>16</sup>The performance phases, sonic realisation in the case of Xenakis, and acousmatic projection in the case of Schaeffer, have been left out.

<sup>17</sup>As this will be later codified in software, we are deliberately excluding any references to psychologically perceived time here.

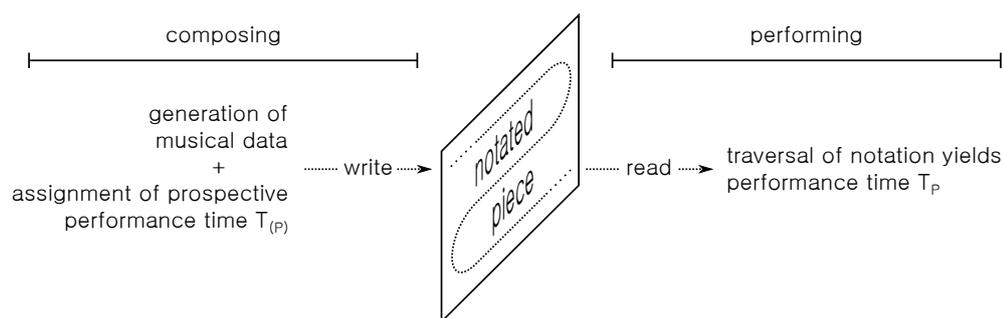


Figure 2.1: Diaphragm model of composition

time, denoted  $\mathcal{T}_{(P)}$ , when referring to a notated or written musical datum. This yields a model of the compositional process compatible with the three composers discussed so far, shown in Fig. 2.1.

It shows the piece as diaphragm separating the composition procedure from its performance. It is permeable only in one direction, and it functions as *causa finalis* for the composition. But it is also a lens refracting (actualising)  $\mathcal{T}_{(P)}$ . And finally, it is a memory cell written in the composition procedure and read in a performance, allowing them to be separated in time and space, while also allowing multiple performances of the same “composition”. Like any model, this is a simplification, especially since it shows only an aperture: it neglects that generation of musical data—if we admit that no musical invention stems from pure “originality”—is a reading process (development of rule systems, gathering of sound material); and that the performance itself is a writing process, transforming the piece through an interpretation which goes beyond the refracting lens (instrumental interpretation, acousmatic interpretation) and inscribing it in the minds of the audience. While the latter is not a subject of this thesis, the former aspect, the reading relation of composing, will be addressed at a later point.

### 2.3.1 Models of Virtual Performance Time

Staying first on the left-hand side of the diaphragm model, existing approaches to modelling  $\mathcal{T}_{(P)}$  in computer composition systems are examined. Since this is the *only time layer* in current systems, it suffices to speak simply of representation of ‘time’ in this section. A basic typology

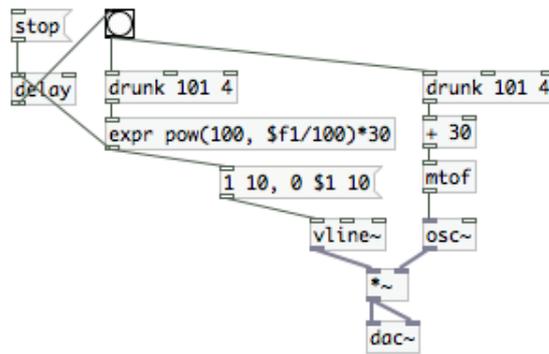


Figure 2.2: A patch in *Pure Data* producing a sequence of tones

was given by H. Honing.<sup>18</sup> He distinguishes tacit, implicit and explicit representations of time. The first in this enumeration, ‘tacit’, is a bit misplaced, since it merely means that «time is not represented at all». It leaves us with two types of representations, ‘implicit’ denoting an unstructured attribution of temporal positions, i.e. without specific *relations* between temporal values, whereas ‘explicit’ representation does allow for the specification of relations.

The trichotomy seems flawed for several reasons. First, the example given of a tacit “representation” (or no representation) is Miller Puckette’s *Max* system<sup>19</sup>. *Max* and its variant *Pure Data* (*PD*) use the data-flow paradigm to describe sound processes, so necessarily there are ways to express time. Fig. 2.2 shows a very simple patch in *PD* which produces a monophonic melodic contour that moves up and down in pitch and accelerates and decelerates randomly. When asking someone unfamiliar with *PD* to spot the objects responsible for temporal unfolding here, they might well guess that it is the `delay` object; and anyone familiar with *PD* will have no problem in stating that the temporal succession is produced by random numbers from 0 to 100 which are translated into an exponential scale of 30 to 3000 milliseconds used to space each tone apart.

It would therefore in this case be better to speak of an implicit or second-order temporal representation in this case, meaning that instead of giving a list of temporal values, a process is

<sup>18</sup>Henkjan Honing (1993), ‘Issues on the representation of time and structure in music’, *Contemporary Music Review* 9(1), pp. 221–238.

<sup>19</sup>At the time *Max* was still a two part system consisting of the actual signal processing part (*MAX*) and graphical front-end *Patcher* (Miller Puckette [1998], ‘The Patcher’, in: *Proceedings of the 24th International Computer Music Conference (ICMC)*, Cologne, pp. 420–429)

described that produces them. Whenever such values have already been produced and are given as numeric or symbolic lists or attributes of an object, we shall use the term explicit or first-order temporal representation, and this may have varying degrees of expressiveness (degrees of specifying relations between those values or attributes). In contrast, ‘tacit’ and non-tacit in Honing’s proposal are not useful because they are just metonyms for his distinction between declarative and procedural structures. The temporal effects of the former are «open to introspection and reflection», while they may be difficult to define in the case of procedural structures, precisely because they allow the definition of interactions and may yield indeterminate temporal effects. The delay object appearing “crossed out” by the cables indicates the feedback nature of this example.

Secondly, A. Marsden notes that the distinction between implicit and explicit as defined by Honing does not matter from a logical perspective.<sup>20</sup> Instead, Marsden begins with an ontology of musical time, and finds four dimensions:

- (1) **Shape.** Whether the flow of time is linear, branching, or circular. A branching perspective might see the future as a set of trajectories branching off from the present. In the opposite direction, an archaeologist may see the different interpretation of artefacts as time branching into the past. In musical representation, branching could be useful for parallel motion between musical layers for which no synchronisation is given, or the possibility of choosing among alternative versions of a section.
- (2) **Extent.** Finite, having a defined beginning and ending, infinite, where either a beginning or ending does not exist, or unbounded which is a situation where the bounds are not (yet) known.
- (3) **Individuals.** The “atoms” of time. Either points (durationless moments) or periods (also called intervals) or events. Events are entities associated with points or periods, and therefore a better distinction would be abstract time atoms versus indicated or embodied time atoms. The latter become clear when considering an event-based system and un-

---

<sup>20</sup>Alan Marsden (2000), *Representing musical time: a temporal-logic approach*, Lisse: Swets & Zeitlinger Publishers.

bounded time; the system may still be operational in time, because time is embodied by a finite number of events (despite “time as such” having no bound). Events somehow oscillate between ontological and representational nature. Purely ontologically speaking, the question is whether time exists independent of something changing (the event) or not.

- (4) **Texture.** Whether time passes in discrete steps or flows continuously. For discrete time, whether a minimum leap size (granularity) exists or not.

In terms of representation, Marsden chooses the approach of temporal logic. As in other forms of logic, it is a symbolic system of rules where propositions can be made that are true or false (which hold or do not hold). It is extended by a temporal qualification of propositions, so that they become conditional under some temporal constraint—e.g. ‘always’, ‘never’, ‘at some time’, ‘before’, ‘after’ or ‘during’ some time, and so forth. The logical, rule-based approach naturally corresponds to the declarative representation in Honing’s paper, although logical statements can be handled procedurally. This will be discussed later in terms of reactive event systems.

For application in artificial intelligence, and consequently in computer music systems, the work of J. F. Allen is seminal.<sup>21</sup> He defines a temporal logic based on time intervals and a set of six relations that qualify pairs of intervals, such as  $STARTS(t_1, t_2)$ , meaning that interval  $t_1$  begins synchronously with  $t_2$  but ends before  $t_2$  ends, or  $OVERLAP(t_1, t_2)$ , meaning that  $t_1$  starts before  $t_2$  and ends after  $t_2$  begins. These relations are asymmetric, so there are six variants with reversed arguments, and finally there is  $EQUAL(t_1, t_2)$  to denote two identical intervals. These relations can be combined with normal logical operators such as conjunction (‘and’) or disjunction (‘or’) to yield implications or equivalences between expressions. For example,  $BEFORE(t_1, t_2) \wedge BEFORE(t_2, t_3) \Rightarrow BEFORE(t_1, t_3)$ : interval  $t_1$  appearing before  $t_2$  and  $t_2$  appearing before  $t_3$  implies that  $t_1$  must appear before  $t_3$ . What makes Allen’s approach interesting is that he provides an online algorithm for maintaining a constraints graph. In this graph, each interval is represented by a vertex, and the constraints (possible relations) form the

<sup>21</sup>James F. Allen (1983), ‘Maintaining Knowledge about Temporal Intervals’, *Communications of the ACM* **26**(11), pp. 832–843; James F. Allen (1984), ‘Towards a General Theory of Action and Time’, *Artificial Intelligence* **23**(2), pp. 123–154.

arcs between vertices. A  $12 \times 12$  ‘transitivity table’ is given which defines how the dynamic insertion of a new constraint propagates to neighbouring nodes.

Temporal relations and temporal constraints have been used interchangeably in the last paragraph. In constraints programming, a variable such as an initially unrestricted temporal position is incrementally specified by adding relations/constraints, and the constraint solving algorithm may then find a solution or several solutions, giving particular values for those variables. Here it means that possible points or intervals are found for the temporally embedded objects, such that each of the temporal relations stated for them holds. An overview of constraints programming in the context of music composition is given by T. Anders.<sup>22</sup>

Unlike the stepwise narrowing of the possible solutions for the variables through the addition of constraints, dynamic removal of constraints is more difficult to implement, so a system which allows constraints to be *themselves subject to time* requires additional constructs. As an example, the work of A. Allombert et al.<sup>23</sup> picks up both the Allen relations and a temporal constraints approach called NTCC (non-deterministic temporal concurrent constraint calculus<sup>24</sup>). They design a system for the “authoring” of interactive scores, where objects can be placed on a timeline and connected through temporal constraints. One type of objects are ‘control points’ which function as interactive triggers during the performance. While “interaction” is restricted to a musician activating those triggers at predefined times, these times allow for variation within a given interval, and the score playhead may be issued to jump, producing loops or skipping certain parts. Their research seems to have led to using petri-nets instead of NTCC to model the temporal succession in a performance, however they are investigating how to add constraints *objects* which only for a defined interval provide a particular restriction on the performance, such as regulating the number of sound layers playing at a time.

---

<sup>22</sup>Torsten Anders (2007), ‘Composing Music by Composing Rules: Design and Usage of a Generic Music Constraint System’, PhD thesis, Belfast: School of Music & Sonic Arts, Queen’s University, chap. 3.

<sup>23</sup>Antoine Allombert et al. (2006), ‘Concurrent constraints models for interactive scores’, in: *Proceedings of the 3rd Sound and Music Computing Conference (SMC)*, Marseille, 14:1–14:8; Antoine Allombert et al. (2008), ‘A System of interactive scores based on qualitative and quantitative temporal constraints’, in: *Proceedings of the 4th International Conference on Digital Arts (ARTECH)*, Porto, pp. 1–8.

<sup>24</sup>Catuscia Palamidessi and Frank D. Valencia (2001), *A Temporal Concurrent Constraint Programming Calculus*, tech. rep. RS-01-20, BRICS Basic Research in Computer Science.

### 2.3.2 Visual Representation of Virtual Performance Time

A major problem arises when trying to present  $\mathcal{T}_{(P)}$  graphically, for example within a user interface. Again, with pieces where composition process and performance are clearly distinguished, this results in different strategies for graphical representation in either case. As this study focuses on the compositional side (although we will eventually show how both become the same), we will restrict this review to questions regarding the score's accessibility to *writing*. A recent special issue on *Virtual Scores and Real-Time Playing* of the *Contemporary Music Review* (Vol. 29, No. 1, 2010) broadly covers the performance aspects of the matter.

As Marsden notes, «Two particular characteristics of musical time, which at least pose difficulties for representation in a one-dimensional linear order, are of particular importance. These are indeterminacy and recurrence.»<sup>25</sup> That is to say, whenever there is a discrepancy, a required non-trivial transformation, between  $\mathcal{T}_{(P)}$  and  $\mathcal{T}_P$ . The need to visualise this transformation stems from the teleological thinking present to some degree in most composers: it comes from a wish to anticipate how a piece sounds or unfolds in the performance. For some, it may be sufficient to imagine this before their mind's eye, particularly when the conceptual aspect of a piece dominates or when its theme is unpredictability or rejection of control (Earle Brown's *December 1952*, for instance). Here lies an undeniable advantage of notation for instrumentalists, because on the one hand the composer is only limited by his imagination when writing the score—which may combine traditional notation with newly invented symbols or graphical and typographical tricks, descriptive text elements, etc.—and on the other hand he may rely on the intelligence and skill of the players.

The situation changes for indeterminate or variable compositions involving electroacoustic material or sound synthesis. A notation for musical data must be found which allows the sound-producing part of the system to read it in the intended way. A visualisation must be provided by the system that corresponds with the declarative or procedural statement of the non-trivial transformation of prospective performance time. Since such transformations are often not a predefined set within the system but written as statements in a general language,

<sup>25</sup>Marsden, *Representing musical time: a temporal-logic approach*, p. 3.

often the only solution is to dispense with any graphical representation of  $\mathcal{T}_{(P)}$ . This is the case with most *Max* patches or pieces of *SuperCollider* code. Another solution is to develop editors which are tailored specifically for a piece (because they enable a notation of the idiosyncratic transformations of that piece, but are not general enough).

### ***Command Control Communications***

An example of the idiosyncratic case is the sound installation *Command Control Communications* (2007; also *CCC*), a collaboration between Cem Akkan (who composed the visual part) and me (sound composition).<sup>26</sup> The playful idea is to recycle cultural waste so that an interesting aesthetic object arises which is both autonomous but also an ironic comment on its material. The visual material is drawn from commercial Hollywood movie advertising (“movie trailers”) dissected into small categorised loops. The visitor to the installation is given a huge remote control with which the four simultaneously projected but otherwise independent “channels” can be switched. Fig. 2.3 shows a photograph. The irony of the images is counterbalanced by stripping off the original soundtrack, replacing it with an electroacoustic composition, loosely synchronised with the image of each channel, creating a polyrhythmic texture with surreal qualities.

A custom editor was written for the sound composition of the installation. It realises one large timeline where all the loops are subsequently located, with a total duration of eight hours.<sup>27</sup> A screenshot of the timeline in the interval 2h21m13s to 2h21m43s is shown in Fig. 2.4. During the playback, each of the four video channels controls an individual synchronised playhead on this timeline. Variability in the sound loops is achieved by two mechanisms:

- (1) Each sound object is represented by a container, called ‘molecule’, which can contain any number of alternative sound regions, called ‘atoms’. The bounding box of a molecule is formed by the union of the contained atoms. In Fig. 2.4 molecules are shown as dashed boxes, having labels starting with the letter *M*, and atoms having labels starting with the letter *A*. For example, molecule  $M_{1987}$  contains four atoms which refer to four

---

<sup>26</sup>Sound and video examples of this piece can be found on the accompanying DVD.

<sup>27</sup>The number of video loops was reduced later, however, so sound composition was carried out for only about half of the material on the timeline.



Figure 2.3: Audiovisual installation *Command Control Communications*

different sound file regions. The atoms have slightly different onset times and durations on the timeline. Each time a playhead cursor reaches the left margin of the molecule, the algorithm decides which of the four atoms will be *realised* (inscribed in  $\mathcal{T}_P$ ).

- (2) Each atom has three variable parameters: time offset, volume, and pitch/speed. They are specified as minimum and maximum values along with an algorithm that controls their realisation within these boundaries. For example, the only atom in molecule  $M_{1988}$  has a duration of 1.7 seconds and a time offset with a variation span of 1.1 seconds. The interval in which it is heard is thus constrained to a box of a length corresponding to 2.8 seconds. The indeterminate parts of this interval (the initial and the final 1.1 seconds) are visually indicated by vertical stripes. Variations in volume (in this case between  $-47$  and  $+16$  decibels) and pitch (here between  $-11.80$  and  $-12.20$  semitones) are only represented textually in a separate window containing information about the currently selected objects on the timeline.



Figure 2.4: Screenshot from the timeline editor of CCC. The vertical overlap of molecule  $M_{943}$  does not have any sonic implication but is a mere artefact of compressing the vertical extent of the display.

The algorithms to choose between minimum and maximum values are WHITE, BROWNIAN, SEQUENCE, SERIES, ALEA, and ROTATION.<sup>28</sup> The first two are continuous pseudo random functions, while the remaining four take a list of discrete breakpoints over which they iterate in various ways.<sup>29</sup> In order to adjust the behaviour of individual atoms, molecules can be “frozen”, preventing the random number generator from advancing that determines which atom inside a molecule is realised.

The sounds realised by the four playheads are mixed together according to another algorithm which arranges them in layers and applies different filtering techniques to control the density and contrast. There is a separate visualisation for the layer arrangement which can also be frozen or switched off during composition. Only the momentary state of the arrangement is visible, so its temporal embedding is not represented.

### General Systems

General systems are independent of a particular composition. One can distinguish three approaches:

- (1) Similar to the editor constructed for *CCC*, the data is somehow coerced onto a linear timeline and enhanced by visual indicators representing the indeterminate elements. For instance the *Virage*<sup>30</sup> visual front-end for the interactive score system by Allombert et al. uses dashed stroking for boxes which are actually subject to temporal variations, despite being rendered as a rigid box at a particular position on the screen.
- (2) The idea of one coherent timeline is abandoned and instead it is deconstructed into pieces. An example of this kind is *Iannix*<sup>31</sup> which is tailored towards graphical scores, as musical structures are represented by polygons and curves. A number of cursors, which can

<sup>28</sup>These algorithms were adopted from the granular synthesiser *Grinder* by Gerhard Eckel, although SEQUENCE, ALEA, and SERIES seem to originate from Koenig’s *Project 2*, cf. Charles Ames (1987), ‘Automated Composition in Retrospect: 1956–1986’, *Leonardo* **20**(2), pp. 169–185

<sup>29</sup>SEQUENCE traverses the list and then repeats, ROTATION goes from front to back and then reverses back to the front. ALEA picks random list elements, and SERIES shuffles the list, uses each element once, and then shuffles anew.

<sup>30</sup>Antoine Allombert et al. (2010), ‘Virage: Designing an interactive intermedia sequencer from users requirements and theoretical background’, in: *Proceedings of the 36th International Computer Music Conference (ICMC)*, New York.

<sup>31</sup>Thierry Coduys and Guillaume Ferry (2004), ‘IanniX: Aesthetical/Symbolic visualisations for hypermedia composition’, in: *Proceedings of the 1st Sound and Music Computing Conference (SMC)*, Paris, 18:1–18:6.

be interpreted as concurrent playheads, “scan” the curves and output their trajectories’ visual coordinates, to which musical meaning must be assigned in the form of a program receiving Open Sound Control data. Furthermore, trigger points can be defined that produce an event when a cursor hits them. This is a purely procedural approach, as no method exists to declare temporal relations and time is only implicit in the dimensions of the curves and the speed of the cursors set into motion by starting a global transport. While this is a very playful tool, coordinated manipulation of time is difficult—for example, there is no notion of inserting an object between other objects *in time*, as generally there is no direct relationship between disjoint curves. On the other hand, it allows new means of expression because different cursors are independent of each other, allowing cyclical polyrhythmic structures or creation of parallel time layers. It is also possible to create and modify the graphical objects dynamically through programs, although it may require some skill and has the side effect that resetting the transport back to a previous offset does not restore the score to its previous state.

- (3) A rather conservative but still very useful approach is to see the score as a particular evaluation of a program, and thus more as a snapshot than the embodiment of the piece itself. An exemplar is the *Maquette* object for the *Open Music* visual patching environment.<sup>32</sup> The authors explicitly refer to the outside-time in-time dichotomy introduced by Xenakis. In a dedicated ‘evaluation’ stage, the in-time ascriptions are carried out and the visual representation as “temporal boxes” is updated. Objects may depend on each other functionally irrespective of their relative placements in performance time, although Bresson and Agon mention that it is possible to introduce a cyclical problem—if during evaluation an object *X* depends on the temporal position of another object *Y* and *Y*’s position changes during the evaluation, *X* has been using an obsolete representation of *Y*. The performance phase is strictly repeatable, objects will no longer change their temporal positions unless another evaluation is issued. This approach is useful because the visualisation is detached from the underlying algorithms for determining in-time positions. Furthermore, the visu-

---

<sup>32</sup>Jean Bresson and Carlos Agon (2006), ‘Temporal control over sound synthesis processes’, in: *Proceedings of the 3rd Sound and Music Computing Conference (SMC)*, Marseille, 9:1–9:10.

alisation truly reflects  $\mathcal{T}_{(P)}$ , as the sonic realisation is another independent stage which may apply additional transformations and interpretations to the temporal boxes.

In summary, the more a piece confines itself to “fuzzy” objects—objects which may vary in their actual positioning in  $\mathcal{T}_P$  only within certain bounds—the easier it is to visually represent its structure in a way that conveys the temporal succession. Taking again the stochastic music of Xenakis, each piece is fully calculated, so an unambiguous score representation is trivial. If the piece was defined to be the stochastic *process* carried out with a fixed set of parameters, at least the contours of the piece could still be visually represented. As Agostino Di Scipio notes, in these stochastic processes «... the unexpected, the singularity of events, does not become a source of information and transformation, but rather favors a levelling-off tendency reflecting the relentless increase of entropic disorder ...». <sup>33</sup> As soon as a recursive element is introduced—that is, the temporal development relies on a previously computed datum—a piece defined as process can only be abstractly represented but not as a tableau of temporal values laid out as simultaneous spatial positions.

It is for this last reason, and also because this direction leaks too much into the different research areas of notation techniques and interface design, that we conclude the brief review of graphical representation of musical time here. In the further course of the thesis, we restrict ourselves to textual description of musical data and musical processes, although a graphical front-end *Mellite* for our computer music system is introduced.

## 2.4 Creation Time

We now introduce a second layer of time, which shall be called creation time, denoted  $\mathcal{T}_K$ . This layer covers, but is not limited to, the process of writing a piece. The perspective is shifted, so that it no longer focuses on *what* is written—the musical datum—or re-written—the ascription of  $\mathcal{T}_{(P)}$  to the musical datum—but on the fact *that* something is written, and that this writing necessarily happens in time. Writing takes time just as reading takes time, and this symmetry is clearly seen in Fig. 2.1. The creation time spans the distance |——| overarching

<sup>33</sup>Agostino Di Scipio (1998), ‘Compositional Models in Xenakis’s Electroacoustic Music’, *Perspectives of New Music* 36(2), pp. 201–243.

the activities on the left-hand side. In a transitory simplification, this “interval” begins  $\vdash$  with the composer’s decision to write a piece, and it ends  $\dashv$  with him declaring the piece finished. A second simplification is to restrict this interval to contain only those activities which are directly *observable* by computer software.

For example, Fig. 2.2 may be the result of the steps depicted in Fig. 2.5. These are somehow slides or snapshots showing how the patch, starting from blank, is incrementally constructed. The sequence is arranged in zig-zag fashion with the new or modified objects and cables in each slide shown with darker borders. The most obvious thing conveyed is the order in which objects were created and connected. The interpretation is much more difficult. Hypotheses can be made, but only verified by asking the composer—and even that may only establish what the composer did consciously. It seems that the initial idea was to construct a random walk, because the *drunk* object is created first. But it may equally be that the initial idea was to create a triggered sine oscillator, and merely the execution of the idea began with the random walk. So if the aim was to retrace the conceptualisation process of the composer, clearly the induction from a sequence of visible actions is afflicted with speculation.

On the other hand, if the observation is plainly described as a motion, the psychological trap is avoided. It begins with the creation of several objects, forming two disconnected graphs (one on the left, one on the right side), which are eventually connected. Then a separate group formed of *20*, *mtof*, and *print* is added, and the message value undergoes changes to *40* and *140*, before they are deleted again, and an object *+ 30* is inserted after the previously created *mtof*. (Again, an intentional hypothesis would be that the effect of *mtof* was tried out on different input values, establishing a useful frequency range for the combination of *drunk* and *+ 30*.) It follows the substitution of *pack 1 10 0 0* for *1 10, 0 \$1 10*, and the substitution of *scale* for *expr*, finally the adjustment of the initial parameters for the *drunk* objects. (An interpretation may be that the *pack* object output was unsuitable to create the desired envelope in the *vline~* object, and that an exponential scale for the random durations was considered more appropriate than a linear scale.)

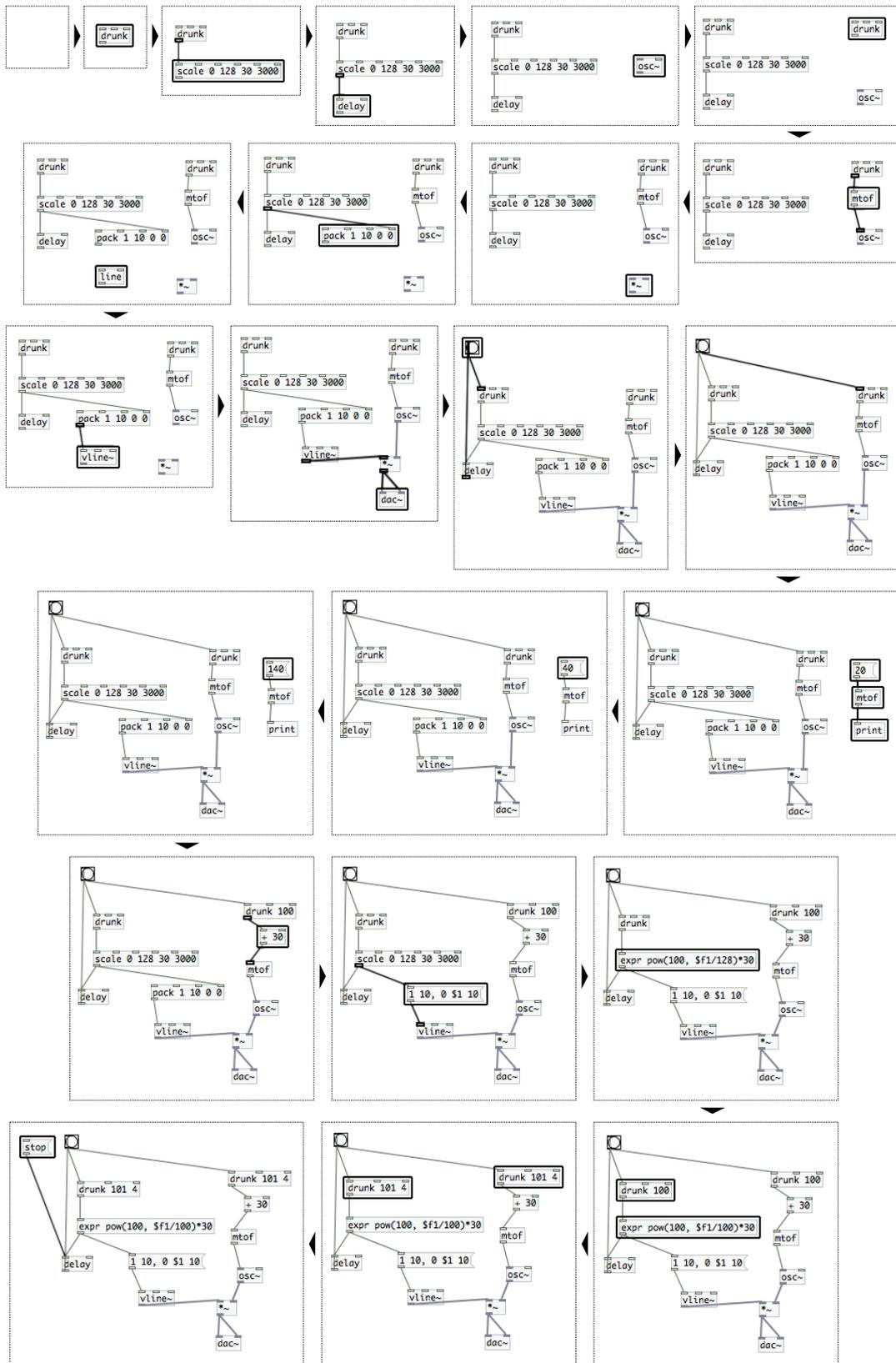


Figure 2.5: Possible evolution of a PD patch

In a more complex and realistic scenario, one might see many more deletions and modifications, as it becomes more difficult for the composer to anticipate the effect of an action within the ensemble of existing objects. Let us remember Koenig’s description of the composer in front of a composition program:

«A composer is more accustomed to being influenced by a spontaneous idea than by prepared plans; he decides and discards, writes down and corrects, remembers and forgets, works towards a goal; replaces it during his work by another . . . »<sup>34</sup>

A system that accounts for both qualities of musical time, the creation time and the performance time, may be used as an analytical tool for a musicologist to guide the determination of the psychological processes going on inside the composer—given the aforementioned premise that additional methodology is employed and found credible, which strengthens any hypotheses about such processes. But more importantly, it will equip the composer with means to observe his own activity and to *connect* these two time qualities in a recursive manner, such that a musical datum may reflect the trajectory of compositional work or that successive compositional work incorporates “performed work”—decisions made during a performance. In other words, it will allow an oscillatory movement between both sides of Fig. 2.1, thereby questioning the diaphragm itself.

## 2.5 Accessing Time

Now that it has become clear *which* temporal data should be represented in a composition system, one must ask *how* this accomplished; in other words, how this data is written and read. Two areas in computer science that are relevant in this context are algorithms and data structures as well as database systems.<sup>35</sup> Data structures are the more general field, while database systems specifically deal with the storage of data on so-called secondary memory (e.g. hard disks).

---

<sup>34</sup>Koenig, ‘Kompositionsprozesse’, p. 197.

<sup>35</sup>These are two areas defined by the CSAB, an accreditation organisation formed by the Association for Computing Machinery (ACM) and IEEE Computer Society, as reflected in Computing Sciences Accreditation Board (CSAB) (1986), *Computer Science as a Profession*, URL: [http://web.archive.org/web/20090117183438/http://www.csab.org/comp\\_sci\\_profession.html](http://web.archive.org/web/20090117183438/http://www.csab.org/comp_sci_profession.html) (visited on 25/05/2012); one might suggest human-computer-interaction as well, but it is not the focus of this thesis.

Since the system we are going to design should be able to store the musical data permanently, it is natural to look at possible approaches in database research. A traditional database has the notion of “tables”, a matrix in which each row corresponds to an individual entry or object, and within a row vector, the columns represent different parameters of the entry. In the *PD* example, an entry would correspond to a box inside the patcher, and the entry parameters could be the box’s type,  $x$  and  $y$  position on the screen, name and default arguments, and connected cables. Then, in an *ephemeral* setting, that is without taking into account  $\mathcal{T}_K$ , the steps in Fig. 2.5 would correspond to the addition, modification and deletion of rows in the table. Of course, this is all hypothetical, as in fact *PD* does not maintain a database, but an in-memory data structure, which is written out only when the patch is saved. Nevertheless, the format of that file, which is a plain text, resembles the speculative matrix:

```
...
#X obj 252 -215 osc~;
#X obj 252 -246 mtof;
#X obj 252 -270 + 30;
...
#X msg 33 -327 stop;
...
#X connect 0 0 5 1;
#X connect 1 0 0 0;
...
```

Indeed objects (*obj*) and messages (*msg*) are represented by said parameters. The cables, however, are separate entities of type *connect*, taking as parameters four integer numbers representing source box, source outlet, target box, and target inlet. In a table database, because of the different *shape* of the box and connector parameter columns, these would go into two different tables. When reading in a *connect* entry, its parameters must be translated to actual boxes. The integers representing the boxes are called identifiers or surrogates. In the above example, they might just indicate the row number in the file representing a particular box.

### 2.5.1 Transactions

How would the evolution of the patch over creation time be represented? A special database variant, called temporal database, is needed. R. T. Snodgrass and I. Ahn have established a

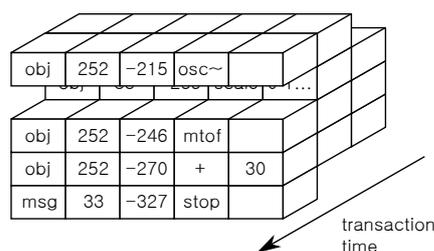


Figure 2.6: Transaction time database

taxonomy for time in databases.<sup>36</sup> They distinguish three types of time, called *transaction time*, *valid time*, and *user time*. The last of these is simply any datum which is not managed directly by the database, for example it is not subject to indexing. In a database, an index allows one to look up an entry by the index key. The two important dimensions are thus transaction and valid time. The transaction time is «the time the information was stored in the database.» Now if the composer performing the steps of Fig. 2.5 is seen as operating on the database, each step corresponds with a particular point in transaction time, and so the addition, deletion or modification of an object would be associated with such a point. This is illustrated in Fig. 2.6 (adapted from Snodgrass and Ahn) as a sort of growing cube. This type is also called “rollback” database because it is possible to go back in the history of the database.

In the illustration, time advances from back to front. Each slice along this axis then corresponds to a *view* of the patcher as it was at that time. In the oldest instant shown here, there are three objects in the patcher (three rows in the database); in the successive time step, the *osc~* is added, bringing the number of rows to four; in the third and last instant shown, a previously existing object (perhaps *scale*) was removed, and the *stop* message added. It implies that several actions can be carried out within the same *transaction*. Physically this is not possible: neither can the composer delete an object and create another one at the same time, nor can an algorithm achieve this, since it would need to break down this transaction into two separate steps, even if those steps were performed so fast that one was under the impression that they happened synchronously.

<sup>36</sup>Richard T. Snodgrass and Ilsoo Ahn (1985), ‘A Taxonomy of Time in Databases’, *ACM Special Interest Group on Management of Data (SIGMOD) Record* 14(4), pp. 236–246.

Transactions are thus a logical abstraction from physical time. They are said to be *atomic*, which means:

- › Transactions can be composed of a number of sub-actions. A transaction is opened, the sub-actions are encapsulated, and the transaction is eventually closed. The closure is also known as *commit*, and a committed transaction can no longer be broken up.
- › Transactions do not “take time”—all sub-actions are considered to happen logically at the same time instant.
- › Time leaps between two transactions, there is nothing to observe between two transactions.<sup>37</sup>

Furthermore, “wallclock” time can be attached to a transaction, denoting the time in the real world corresponding to the transaction, e.g. “at which date and time was an object created in the database?” The wallclock is usually read when the transaction is committed (it has been decided that no more sub-actions will be included). Wallclock time increases monotonically, and there is no way to issue a transaction which happens before another already committed transaction. It follows that the time-line formed by all the transactions can be further abstracted, as shown in Fig. 2.6, by being composed of equidistant points which are simply incrementally numbered ( $t_1, t_2, t_3$ ), even though in the physical world the time span that passed between  $t_1$  and  $t_2$  might be greater or smaller than the one between  $t_2$  and  $t_3$ .

A database supporting transaction time will never physically delete an object. Instead, a new transaction time slice is added where that object simply no longer appears. Therefore, the history of the creation of a structure (e.g. the patch) is preserved, and it is possible to “roll back” the database to any previous transaction time instant and view the stored structure as it has been at that time. If the term creation time,  $\mathcal{T}_K$ , is used to indicate those actions carried out by a composer in the process of composing a piece, which are observable by the computer system through association with transactions, then transaction time can be said to represent  $\mathcal{T}_K$ .

<sup>37</sup>In Marsden’s categories, the individuals of a transaction system are thus point events, and the texture is discrete.

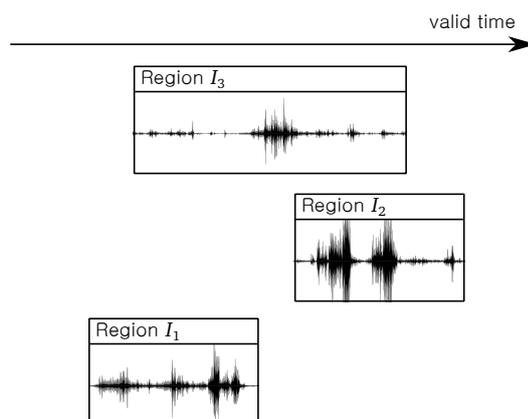


Figure 2.7: Schema of a multi-track editor

### 2.5.2 Valid Time

Another situation requires the expression of time independent of the action of entering it into the database. Snodgrass and Ahn call this ‘valid time’ because it is «indicating the points in time when the tuple accurately modeled reality.»<sup>38</sup> Without dwelling upon the status of “reality”, to give an example, in a business application where the database contains entries for the employees, such datum could be the validity of their contracts. The entry for an employee might be created before or after he actually starts to work for the company, so temporal information independent of the transaction time is needed. This information is not just “user time”, but treated specifically so the database can efficiently answer queries of the type “list all people who were employed on day  $X$ ”.

An obvious use case in music is the representation of virtual performance time,  $\mathcal{T}_{(P)}$ . The “reality” then is the situation of the performance, although a composer will not—except maybe in very particular pieces—encode a datum such as “play sound  $X$  at midnight on September 5, 2012”. Exactly because he encodes  $\mathcal{T}_{(P)}$  and not  $\mathcal{T}_P$ , this datum will instead refer to a *time base*, for example it might be relative to the beginning of the piece, no matter when the piece is played. Apart from this difference, the situation is quite similar to the business example. A typical query might be “list all objects which are sounding after 4 minutes from the beginning of the piece”.

<sup>38</sup>Snodgrass and Ahn, ‘A Taxonomy of Time in Databases’.

In a particular composition system, this might be a mediated query. For example, take the common tool used in fixed media electroacoustic composition, a multi-track editor. Its interface is schematised in Fig. 2.7. The composer typically works entirely inside this visual representation, moving around sound “regions” with the mouse. He may focus on a particular part, e.g. “move the timeline” to a window around four minutes into the piece. If the regions were stored in a database, the graphical interface would translate this movement into the query “. . . all objects which are sounding after 4 minutes . . .”, therefore the database is required to handle valid time, and valid time can be said to represent  $\mathcal{T}_{(P)}$ .

Note that the sound regions represent temporal successions in themselves, i.e. they refer to audio files representing samples of sound taken at a particular sampling rate (such as 44.1 kHz). An audio file could be seen as a temporal database, too, but its structure is much simpler than the regions list, where regions can appear at arbitrary positions and there may be overlaps on the timeline or empty parts with no regions. A “query” such as “all audio samples between one and two seconds” for an audio file can be directly translated to a logical offset into the file and a number of samples to be read, given that the audio file format is not using a compression scheme.

### 2.5.3 Bi-Temporality

Just as there is very little research on the creation timeline, none of the computer music systems known to us supports storage of the transaction timeline. The only option is to combine the music software with a generic software versioning system.<sup>39</sup> While this allows one to restore a previous version of a piece, the versioning information remains external: it is not possible to handle historical traces within the representation model of the music software itself.

When looking at a valid or  $\mathcal{T}_{(P)}$  timeline, such as Fig. 2.7, no information is preserved as to how the regions came into existence and ended up in their final positions.<sup>40</sup> Widely used multi-track editing systems, such as *Protools* or *Ardour*, only provide support for the standard desktop

<sup>39</sup>This is for example suggested by C. Burns, although he acknowledges that this might be useful for a language such as *Common Lisp Music*, and less for other applications such as *Protools*. Christopher Burns (2002), ‘Tracing Compositional Process: Software synthesis code as documentary evidence’, in: *Proceedings of the 28th International Computer Music Conference (ICMC)*, Göteborg, pp. 568–571

<sup>40</sup>Although there may be subtle indicators—observe the numbering of the molecules in Fig. 2.4.

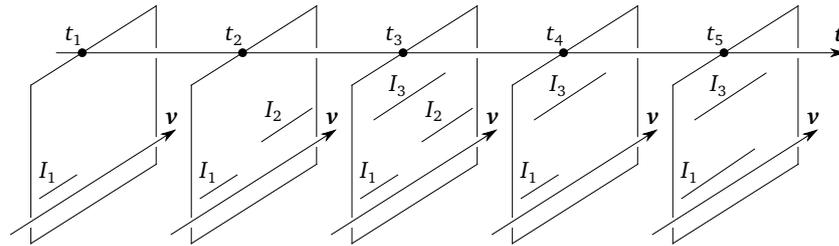


Figure 2.8: Schema of a bitemporal data structure

application metaphor of an undo-history. While the application is running, editing actions can be reverted step-by-step. When the application is closed, this history is lost. The purpose of the undo-history, as its name suggests, is merely to be able to correct recent mistakes by making the last editing steps undone.

Fig. 2.8 shows what a multi-track editor supporting transaction time might look like. The figure is adapted from an investigation into *bitemporal* databases.<sup>41</sup> A bitemporal database system is one which combines transaction time and valid time.<sup>42</sup> As depicted, both layers can be conceived as orthogonal to each other. The transaction time axis  $t$  runs horizontally, with individual points labelled  $t_1, t_2, \dots$ , and the valid time axis  $v$  extends into the  $z$ -plane, while the vertical space is merely used to handle the overlap between the valid time entities  $I_1, I_2, I_3$ . Two things can be noted:

- (1) Although not a requirement for bitemporality, in many cases transaction time is atomic (punctiform) whereas the valid time data often has the form of intervals.
- (2) The transactional timeline grows constantly, new transactions are appended. Objects on the valid timeline however may be freely adjusted. For example in transaction  $t_5$ , the duration of interval  $I_1$  is extended.

This second observation stems directly from the independence of both time dimensions. It places musical time at the unrestricted disposal of the composer, because it is not necessarily

<sup>41</sup>Anil Kumar, Vassilis J. Tsotras and Christos Faloutsos (1998), 'Designing Access Methods for Bitemporal Databases', *IEEE Transactions on Knowledge and Data Engineering* **10**(1), pp. 1–20.

<sup>42</sup>Christian S. Jensen et al. (1992), 'A Glossary of Temporal Database Concepts', *ACM Special Interest Group on Management of Data (SIGMOD) Record* **21**(3), pp. 35–43.

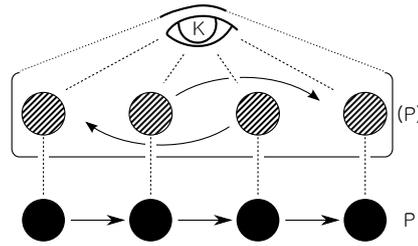


Figure 2.9: Decoupled creation and virtual performance time

a musical datum that causes another musical datum (requiring the latter to appear after the former); instead, the composer is the “prime mover”, able to rest as long as desired between transactions. This view is illustrated in Fig. 2.9. The composer has a bird’s eye view of the musical data and can relate it independent of any temporal ascriptions (in the case of “in-time” data). Very much like a god he embodies the role of creating *ex-nihilo*, starting with a blank screen or piece of paper. Transformations between  $\mathcal{T}_{(P)}$  and  $\mathcal{T}_P$  are fixed and the piece is clearly delimited. Compare this with a clock-wise rotated view of Fig. 2.1.

Xenakis reflected this role in his answer to the question “What is a composer?” in the chapter *Concerning Time, Space and Music*.<sup>43</sup> In his first remark, concerning entropy, he writes that «Indeed, much like a god, a composer may create the reversibility of the phenomena of masses, and apparently, invert Eddington’s “arrow of time.”» The god metaphor however is not so much used in the Aristotelian sense, but in reference to a theory of the evolution of the universe, having begun in complete order and moving towards increasing entropy. A. S. Eddington<sup>44</sup> explains the directionality of time (the “arrow of time”) from this increase in entropy, which is embodied in the second law of classical thermodynamics—a “secondary law” applying to groups of elements, as opposed to the “primary” laws of physics concerning only individual entities. The time of primary laws is the time of mathematics, Xenakis’ metric (symmetric) time, which appears as a reference, a functional argument, something that can be «undone» as Eddington says: A sequence of instants beaded on a chain, which can be traversed from either direction. In Xenakis’ works dealing with masses of events, the statistical methods can be used

<sup>43</sup>Iannis Xenakis (1992), *Formalized Music, Thought and mathematics in composition*, Revised Edition, Stuyvesant, NY: Pendragon Press.

<sup>44</sup>Arthur S. Eddington (1929), *The Nature of the Physical World*, Cambridge: Cambridge University Press.

to go from disorder to order at will, as the temporal algebraic functions used for establishing lexicographic order are a secondary process.

Xenakis' second remark addresses the problem of «creating something from nothing», which will appear a number of times in our text (e.g. Sect. 4.4.5).

## 2.6 Access Methods

With the bitemporal taxonomy established, it must be demonstrated how such data can be *accessed*. Accessing the data means to read, write or update an entry in the database. Each access comes at a *cost* in terms of *space usage* and *time*. Space usage describes how much storage space (e.g. bytes in memory or on hard-disk) is necessary to represent a datum. The time cost describes how long it takes the computer to perform a given operation. This time is not included in the bitemporal model, since being logically without duration is part of the abstraction of a transaction. And this is mostly sufficient, as in a system design we do not want to model temporality based on the speed of a particular computer or storage medium. Nevertheless, it means that assumptions are made about the insignificance of the access times—in an offline system, where the composer works “outside time”, there are no real restrictions on this access time, however they might disturb the process of composition if the composer experiences noticeable delays when carrying out an action. In a real-time system, where the computer performs the piece or parts of it during the compositional process, the access times need to be small enough not to introduce perceivable distortions during the production of musical events in  $\mathcal{T}_P$ .

The costs can be specified in the *O*-notation<sup>45</sup> as the asymptotic worst case (upper bound) behaviour function of a number of crucial parameters. A simple scenario would be an in-memory representation for valid time only, where valid time is given in points and not intervals. A binary search tree would be a suitable structure to efficiently store and query such values.

---

<sup>45</sup>Thomas H. Cormen, Charles E. Leiserson and Ronald L. Rivest (1990), *Introduction to Algorithms*, Cambridge, MA: The MIT Press, pp. 6–10, 23–32.

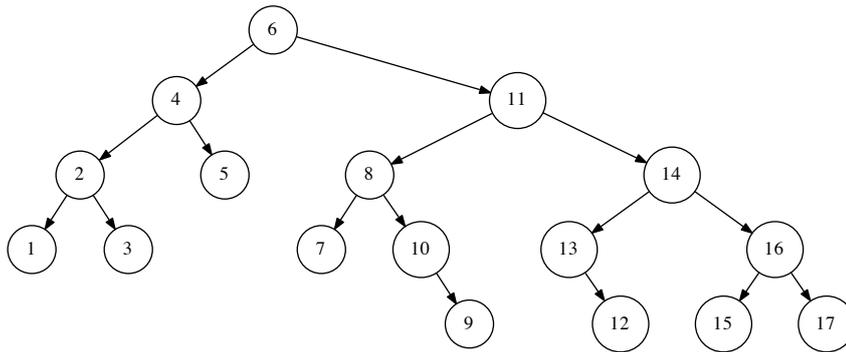


Figure 2.10: A binary search tree

The AVL tree<sup>46</sup> and the red-black tree<sup>47</sup> are two data structures with good performance (low costs): In both cases, the insertion and deletion of a point have a time cost of  $O(\log n)$ , where  $n$  is the current size of the tree, i.e. the number of entries in the tree.<sup>48</sup> The query, i.e. to answer whether a point is contained in the structure or to retrieve a point closest to a given point, has the same time cost of  $O(\log n)$ . The ideal time cost would be  $O(1)$  which means constant cost—no matter how large the tree is, a point can always be found in the same amount of computation time. AVL tree and red-black tree have optimal space costs of  $O(n)$  which means that to store  $n$  entries, no more than  $n$  memory cells are needed. An example tree is shown in Fig. 2.10. To find an element, one starts at the root, compares the query element to the root element and then either descends to the left or the right. The comparison is then repeated until either the query element has been found or the leaves of the tree have been reached.

Cost predictions are made difficult due to the following contributing aspects:

- › **Computation Model** The above values are assuming the RAM (random access memory) model of computation, which means that following from one node of the tree to a child node or the creation of a new node are constant time operations, and there are no requirements as to how the branches and leaves are localised in the memory. If these structures

<sup>46</sup>Caxton C. Foster (1965), ‘Information retrieval: information storage and retrieval using AVL trees’, in: *Proceedings of the ACM 20th National Conference*, Cleveland, OH, pp. 192–205.

<sup>47</sup>Leo J. Guibas and Robert Sedgwick (1978), ‘A dichromatic framework for balanced trees’, in: *19th Annual Symposium on Foundations of Computer Science (FOCS)*, IEEE, Ann Arbor, pp. 8–21.

<sup>48</sup>Since the  $O$ -notation only denotes the magnitude of performance, leaving out constant factors, the base of the logarithm is irrelevant. Some authors prefer to write  $\lg = \log_2$  instead, although  $\lg$  is ambiguous and may also be read as  $\log_{10}$ .

are stored on a hard-disk, a different model might be more appropriate which measures the amount of contiguous blocks read from or written to disk, giving way to alternative structures such as the B-tree.<sup>49</sup>

- › **Amortisation** Some algorithms only guarantee amortised costs.<sup>50</sup> This means that the worst costs are to be understood only as an average over a number operations, while a single operation may occasionally require more time, e.g. because the insertion of a datum requires the splitting of a node. These algorithms are thus problematic under strict real-time requirements.
- › **Dynamic Behaviour** Some data structures might have optimal bounds, but are static and cannot provide those guarantees under dynamic modification.
- › **Parallelism** Current computer processor development exhibits an increase in the number of cores which can work in parallel. Not all data structures are suitable for parallelisation and may thus perform inferior to well parallelised structures.
- › **Concurrency** As a result of parallel processing capability, a program may require that different threads access a data structure concurrently. Classic data structures often assume mutable memory cells, and their algorithms need to be adapted to cope with the concurrent situation. New approaches such as purely functional data structures<sup>51</sup> might be favoured, but these again interfere with the potential for storing them on hard-disk . . .

In conclusion, care must be taken in the choice of existing, and the creation of new, data structures with respect to their performance. Whenever feasible, structures that provide constant time operations  $O(1)$  should be given preference. However, due to the difficulty of taking all cost aspects into account—and since this thesis aims to explore a new creative approach instead of making claims about optimal bounds of the algorithms and structures used—generally

---

<sup>49</sup>See for example the following paper which applies the cache-oblivious model to B-trees, making them predictable under different conditions where memory is hierarchical (e.g. the structure must be transferred between main memory and a secondary memory such as the hard-disk): Michael A. Bender, Erik D. Demaine and Martin Farach-Colton (2000), ‘Cache-Oblivious B-Trees’, in: *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, IEEE, Redondo Beach, pp. 399–409

<sup>50</sup>cf. Cormen, Leiserson and Rivest, *Introduction to Algorithms*, pp. 356–377

<sup>51</sup>Chris Okasaki (1998), *Purely Functional Data Structures*, Cambridge: Cambridge University Press.

structures are accepted whose time costs are logarithmic in the size of the structure,  $O(\log n)$ . Then, even if due to transfer to secondary memory or *nesting* of data structures the costs rise to a polylogarithmic term, the rise is slower (“sub-linear”) than the growth of the data structure, becoming less and less significant as the database grows.

### 2.6.1 Multidimensional and Interleaved Structures

Since bitemporal databases merely provide a point of departure for our model, we do not wish to go into too much detail. A thorough comparison of access methods for temporal and bitemporal databases is given by B. Salzberg and V. J. Tsotras.<sup>52</sup> Nonetheless, an examination of the bitemporal systems of A. Kumar, V. J. Tsotras and C. Faloutsos<sup>53</sup> will help clarify some of the methods later used in our own design. Their initial observation is that naïve solutions, such as copying the whole data structure for each transaction step or just recording a log book of incremental changes to the initial structure, yield very bad update and query performance respectively.

An interesting idea is born directly from the visualisation of Fig. 2.8—transaction and valid time can be seen as two orthogonal axes of a plane in which the data is inscribed. Thus, any well performing spatial data structure can be used. A spatial data structure can encode multi-dimensional data and offers search algorithms such as finding all objects within an interval, within a rectangle or near a given point. Spatial structures have been well researched in the field of computational geometry, spatial databases and geographic information systems (GIS).<sup>54</sup> Kumar, Tsotras and Faloutsos use the R-tree<sup>55</sup> in one of their implementations. Fig. 2.11, adapted from their paper, shows this idea. An example query  $(t_i, v_j)$  finds any interval existing at (i.e. containing) transaction time point  $t_i$  and containing the valid time point  $v_j$ . Since interval  $I_1$  is changed in  $t_5$ , it is represented by two adjacent rectangles.

<sup>52</sup>Betty Salzberg and Vassilis J. Tsotras (1999), ‘Comparison of Access Methods for Time-Evolving Data’, *ACM Computing Surveys* **31**(2), pp. 158–221.

<sup>53</sup>Kumar, Tsotras and Faloutsos, ‘Designing Access Methods for Bitemporal Databases’.

<sup>54</sup>For an overview of spatial data structures see Hanan Samet (1995), ‘Spatial Data Structures’, in: *Modern Database Systems: The Object Model, Interoperability and Beyond*, ed. by Won Kim, New York: ACM Press and Addison-Wesley, pp. 361–385

<sup>55</sup>Antonin Guttman (1984), ‘R-trees: A dynamic index structure for spatial searching’, in: *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, Boston, pp. 47–57.

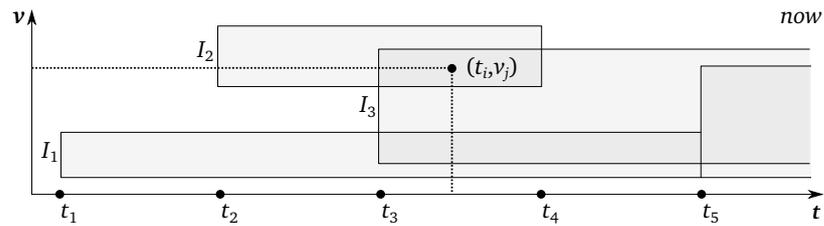


Figure 2.11: Modelling bitemporal data as an R-tree

The R-tree requires that all data is bounded by a root rectangle, putting an artificial restriction on the valid time and posing a problem to the representation of musical data that extends to the current transaction, i.e. exists in the most recent version. This problem of representing “now” can be solved by either using a special coordinate for “now”—e.g. the largest number representable with a given number of bits—or by maintaining two trees, one for past data and one for actual data.

A second method is to decompose the access into two strategies. In the first stage, again an off-the-shelf data structure is taken which performs well in the valid time case, but it is then interleaved in the second stage to provide the additional transactional layer. Good structures for representing the valid time intervals are the Interval Tree<sup>56</sup> and the R-tree (only one of the two dimensions of the R-tree is necessary, the other dimension could be used for an additional key). The trick is then to transform this structure according to the *partial persistence* approach developed by J. R. Driscoll et al.<sup>57</sup> In this transformation, a data structure is modified in such a way that each mutable field records the history of its “overwriting”. When a query is performed and the structure traversed, mutable fields are read by finding the version most recent with respect to a given query version. That is to say, looking at Fig. 2.8, a given transaction time is passed into the query of the spatial data structure which will then be *viewed as if* it was a single historic time slice. A clever methodology is guiding the persistent transformation to minimise spatial and temporal costs in contrast to the naïve scheme of verbatim copies of each time slice.

<sup>56</sup>Herbert Edelsbrunner (1983), ‘A New Approach to Rectangle Intersections (Part I+II)’, *International Journal of Computer Mathematics* **13**(3-4), pp. 209–229.

<sup>57</sup>James R. Driscoll et al. (1989), ‘Making data structures persistent’, *Journal of Computer and System Sciences* **38**(1), pp. 86–124.

The advantage of this second approach is that  $\mathcal{T}_K$  and  $\mathcal{T}_{(P)}$  are decoupled. Musical data might be heterogenous, and not all data might have temporal ascriptions (cf. “outside time”). A uniform approach to modelling creation time based on ‘persistence’ can be combined with multiple representations of performance time. Since a variant of persistence is also used in our implementation, a longer discussion is given in Sect. 5.5.

## 2.7 Branching and Multiplicities

It seems that a suitable model for the representation of the compositional process has been found: In a bitemporal database  $\mathcal{T}_K$  corresponds with the transaction timeline, and  $\mathcal{T}_{(P)}$  is represented by valid time data. But is it sufficient to view each dimension as a *time-line*?

### 2.7.1 Branching as Parallel Motion

It has already been noted that a composer’s behaviour may not be led by a clear goal, but that the strategy may shift during the compositional process, and moreover that he might make decisions which he later revokes as errors. Schaeffer, too, acknowledges a “groping” which characterises at least the first phase of the process, in which establishing groupings and relationships between sounds only become gradually clear.<sup>58</sup>

In one of the few studies systematically observing composers at work, D. Collins traces the actions of a professional composer working with a MIDI- and synthesiser-based system.<sup>59</sup> He confirms that the composer works with incremental goal adjustment, and most importantly detects a *concurrency* of strategies. The composer in his case study would develop different themes independently and eventually chose not to settle on either of them, but to combine them into one piece. Moreover, «he took the unusual step of looking outside the composition to material he had written elsewhere and ‘imported’ a new theme ...». In conclusion, Collins advocates a model of the “problem solving” process which includes *branching*.

An exhaustive tracing of the compositional process thus needs to be able to represent this concurrency. A regular bitemporal system is limited, as transactions can only be appended

<sup>58</sup>Schaeffer, *Traité des objets musicaux*, p. 381.

<sup>59</sup>David Collins (2005), ‘A synthesis process model of creative thinking in music composition’, *Psychology of Music* 33(2), pp. 193–216.

to the most recent state of the data structure. In contrast, concurrency is well addressed in collaborative software and revision control systems.<sup>60</sup> Here, a group of people needs to be able to work concurrently on a project—this might be a text document, or a software development project—which implies that multiple “timelines” branch off from a common root. In software development, a branch that evolves more or less into a self-contained project is also called a ‘fork’. Yet, just as important as branching, a mechanism is needed through which the different threads are merged again into one body (of text, of software). Returning to the music composition system, the composer must be enabled to start working on an idea from any previous situation (point in  $\mathcal{T}_K$ ), either

- › in order to dismiss the most recent actions on a branch
- › in order to develop concurrent (alternative) versions of a piece
- › in order to develop variants within a piece which are then merged back into the “main” piece

It goes without saying that a fourth perspective indeed is to allow multiple composers to work on the same piece, although this scenario is not investigated in this thesis.

It should further be clarified that “branching in  $\mathcal{T}_K$ ” is not in conflict with the linear advancement of time. The transactions may still be ordered linearly on a timeline, however the musical structure is multiplied and must be logically represented as a tree or—when branches are merged together—as a graph. It is thus not sufficient to ask how the piece looked “last Monday at noon”, but an additional information denoting the branch is needed, e.g. “last Monday at noon in the main branch”. If transactions are restricted to include only sub-actions within one branch, then instead of associating them directly with wallclock time, they can be given incremental logical numbers  $(t_1, t_2, t_3, \dots)$  which uniquely identify them. A ‘version’ in the graph of transactions then denotes the view of the data structure that results from traversing the graph from its root up until and including a particular transaction which “concludes” the

---

<sup>60</sup>For an early account on revision control see Walter F. Tichy (1982), ‘Design, Implementation, and Evaluation of a Revision Control System’, in: *Proceedings of the 6th international conference on Software engineering ICSE*, IEEE, Tokyo, pp. 58–67

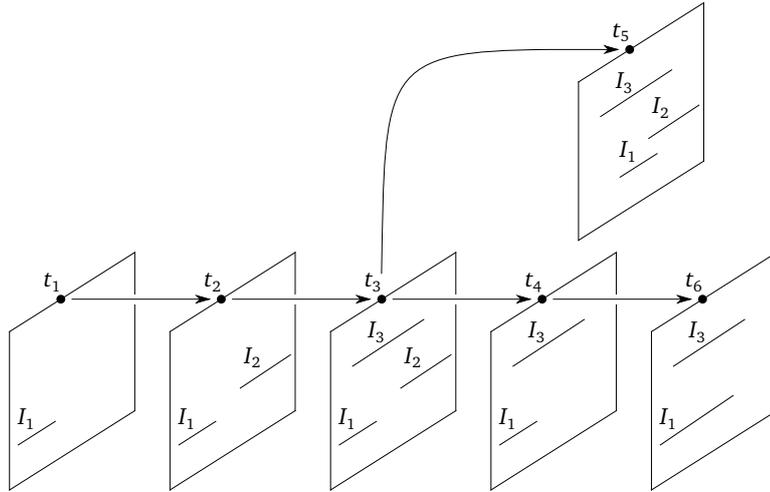


Figure 2.12: A branched and temporal structure

version. We can then ask how the piece looked “in version  $X$ ”, and wallclock time and branch are attributes of the version’s concluding transaction. This is illustrated in Fig. 2.12, where version  $t_5$  initiates a new branch, and hence when referring to object  $I_1$  in  $t_5$ , we refer to that branch along with a transaction belonging to that branch.

This tuple indexing is also called “branched and temporal”, and L. Jiang et al. have developed a structure called the BT-tree implementing this index.<sup>61</sup> However, valid time data is not directly supported, and performance degrades with the number of branches. An example of a direct versioning approach, indexing by the unique version identifier instead of the tuple  $(Branch, Time)$ , is the *fully persistent* B-tree.<sup>62</sup> This variant of the persistence method of Driscoll et al. allows transactions to branch off any past version of the data structure. None of these branching structures support the merging of branches, however.

### 2.7.2 Branching as Serial Motion

The translation from  $\mathcal{T}_{(P)}$  to  $\mathcal{T}_P$  might not be as direct as in a tape composition for which a single performance timeline is sufficient. In the example of the *CCC* installation, two tricks were used to be able to use a single timeline: First, multiple real-time cursors allowed the production of

<sup>61</sup>Linan Jiang et al. (2000), ‘The BT-Tree: A Branched and Temporal Access Method’, in: *Proceedings of the 26th International Conference on Very Large Data Bases (VLDB)*, Cairo, pp. 451–460.

<sup>62</sup>Gerth Stølting Brodal et al. (2012), ‘Fully Persistent B-Trees’, in: *Proceedings of the 23rd Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, Kyoto, pp. 602–614.

several independent playbacks of the timeline which were then overlaid, furthermore the cursors were instructed to jump at the end of a section, dividing the monolithic timeline into a multitude of short looping timelines. Second, the aleatoric variations of the ‘molecules’ and ‘atoms’ were individual, so no sound region would depend on the position of another region but only relate to an absolute position on the timeline, making it possible to graphically represent the region and encode it in an efficient underlying search structure.

The questioning of the timeline goes back to what has been said about visual presentability in Sect. 2.3.2. The more the composition is based on algorithmic elements, the more the notion of a linear timeline becomes meaningless or at best blurry, and the composer is reasoning more about actions and reactions, behaviours and so forth. It is that the writing has become meta-writing. As D. Harel notes, «programming is not about doing; it’s about causing the doing».<sup>63</sup> The composer writes instructions for the inscription in time. As a hypothetical example, let us imagine a stochastic composition along the lines of Xenakis. The interesting (primary) process is the writing of the parameters that control the stochastic movements, not the (secondary) individual calculations of these movements. Nevertheless, the secondary process might be a crucial element in the composition process: It is observed by the composer as the verification of the designed algorithm.

In this scenario, the “groping” becomes manifest in two stages—first in the rewriting of the algorithm, under the assumption that the system can trace this somehow; and second, as the successive “rendering” of the algorithm as lexicographic temporal inscriptions in each development cycle. For example, Di Scipio notes about Xenakis that he «... tends to create a mechanism that, once started, exhibits itself in time, rendering auditorily explorable the potential of knowledge captured in the theoretical premises and assumptions behind the model mechanism».<sup>64</sup> Now, even if he does not talk about his process of finding the parameters of the models he uses, the mastery evident in his work strongly suggests that he thoroughly tested and tuned them to arrive at their final values.

---

<sup>63</sup>David Harel (2008), ‘Can Programming Be Liberated, Period?’, *Computer (IEEE)* **41**(1), pp. 28–37.

<sup>64</sup>Di Scipio, ‘Compositional Models in Xenakis’s Electroacoustic Music’.

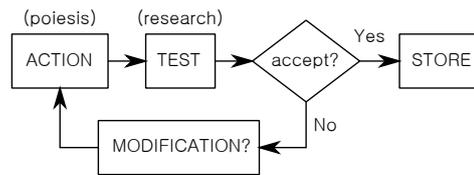


Figure 2.13: Simple model of the compositional process

Such a test-cycle model has been proposed by Emmerson<sup>65</sup> as a first simplified approximation of the compositional process, and it is shown in slightly modified form in Fig. 2.13. The TEST node would degenerate into the rendering of the algorithm output, observable by the system, and the listening and judgement of the composer, not observable by the system, although once more an implicit speculation may be made by looking at the successive action taken (which again is inscribed and observed by the system).

In this iterative approach, the rendering can be seen as the production of stretches of  $\mathcal{T}_{(P)}$ . Although each in a way replaces its predecessor, keeping them as traces—almost like a palimpsest—can shed light on the process in which the algorithm was developed. It may sound redundant, as it has been said that the system ideally traces the formulation of the algorithm itself, and therefore would be able to re-render each iteration at a later point for review. However

- › the rendering might involve unique material from outside of the system (e.g. a live sound signal being recorded as part of the process)
- › the “redundancy” is reflected in two orthogonal representations, a procedural description and an explicit temporal condensation<sup>66</sup>
- › if identifiers are given to individual sound objects, it may be possible to follow their motion across the iterations (e.g. view them simultaneously in space)

All these points will become more clear in the course of this text. It suffices here to emphasise that the branching in the version tree can be exploited to account both for parallel (concurrent, forking) activities as well as for the preservation of sequential iterative processes.

<sup>65</sup>Simon Emmerson (1989), ‘Composing strategies and pedagogy’, *Contemporary Music Review* 3(1), pp. 133–144.

<sup>66</sup>cf. Honing, ‘Issues on the representation of time and structure in music’

## 2.8 A Comprehensive Model of the Compositional Process

Until the last section, the assumption has been upheld that  $\mathcal{T}_K$  and  $\mathcal{T}_{(P)}$  can be clearly separated, and indeed this is true for many approaches to composition and is reflected in many software systems («AlgoScore has a non-realtime perspective, where the composer can relate freely to time and construct the composition outside of time.»<sup>67</sup>) This model is again depicted in Fig. 2.14a. The freedom with respect to manipulation of time during the compositional process is contrasted with the product character of the work—«...closed with regard to the result, but open with regard to the making ...»<sup>68</sup> It is a paradoxical situation: The *random access* which for example a tape composition system offers, allows the composer to explore the virtual performance time in a very concrete form, as he can listen directly to the piece. Before his mind's eye, many different paths may appear, some of which he may try out in the process. But the closure of the piece means that only one designated path in the decision space is left to be presented.

So if a transaction time registration is used to capture the exploration of this horizon of virtualities, it still hits the diaphragm which separates composition from performance. Why should this observation process remain restricted to the composer himself and perhaps a musicologist analysing his work? The closure of the piece is described by Emerson as a “masterwork syndrome”.<sup>69</sup> The masterwork's paradigms include that the composer only creates one version of a work which is then a monolith, unaffected by the time and site at which it is performed, and that the test-cycle of experimentation only takes place once until a final form is found for the work.

Dissent is uttered from different sides, challenging the boundary between composition and performance. One side is “interactive composing”, as coined by J. Chadabe. This type of composing is still founded on an asymmetry of steps:

---

<sup>67</sup>Jonatan Liljedahl (12th Nov. 2008), *AlgoScore user guide*, URL: <http://download.gna.org/algoscore/Help/algoscore-manual.pdf> (visited on 10/05/2012).

<sup>68</sup>Koenig, ‘Kompositionsprozesse’.

<sup>69</sup>Emerson, ‘Composing strategies and pedagogy’.

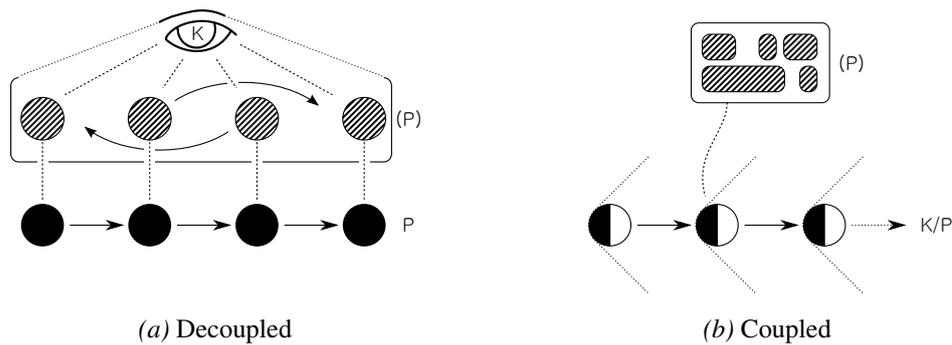


Figure 2.14: Relationships between the two time layers. The horizon of potential actions in  $\mathcal{T}_K$  is indicated by dashed lines, actual actions by white circles. Prospective and realised performance time are indicated by grey and black shapes respectively.

- «(1) a design stage, where the composer designs a specific compositional process, using any of the modules available in the program, and  
 (2) an operation stage, where the composer’s process plays back and the composer interacts with the playback according to the design»<sup>70</sup>

Yet it is clear that the piece is not static but augmented in each performance. Although one writing process has terminated—the design of the interactions—the secondary writing process in which the composer interacts with the piece in a performance (note that Chadabe does not use the word ‘performer’ here) is kept indefinitely open.

More radically departing from the idea of the “piece” is the discourse on improvisation. G. E. Lewis criticises the Eurocentric view that a composition is embodied by a structure which «... inevitably arrives in the form of a written text, a coded set of symbols, intended for realization in performance by a “performer.”»<sup>71</sup> Instead, in his own software, *Voyager*, which he alternatively calls an “environment”, a “program”, a “system”, and a “composition”, he tries to dissolve the difference between composing and improvising. This resonates well with the prompt by D. Charles to conceive «a music that takes care to consider the composer not as the organizer of a technological ritual but, more modestly, as the first listener».<sup>72</sup>

<sup>70</sup>Joel Chadabe (1984), ‘Interactive Composing: An Overview’, *Computer Music Journal* **8**(1), pp. 22–27.

<sup>71</sup>George E. Lewis (2000), ‘Too Many Notes: Computers, Complexity and Culture in “Voyager”’, *Leonardo Music Journal* **10**, pp. 33–39.

<sup>72</sup>Daniel Charles (1965), ‘Entr’acte: “Formal” or “Informal” Music?’, *The Musical Quarterly* **51**(1), pp. 144–165, p. 152.

What these approaches have in common is that they convolve the time in which a piece is perceived and the time at which decisions are made. This convolution is the essence of ‘real-time’. This is shown in Fig. 2.14b. The composer-performer here is embedded in the real-time of the performance, and in each moment the “future” (horizon of virtualities) appears as a more or less wide cone with fuzzy borders. Each decision is embedded in the play, and  $\mathcal{T}_K$  and  $\mathcal{T}_P$  coalesce. This experience of time is described by Marsden as being

«... inexorably drawn along. We move continuously in one direction and cannot voluntarily change our ‘viewpoint’. This is clearly true of ‘actual’ time ... It is hard to imagine any other kind of time in which, if we (or any other agent, whether human or machine) are ‘in’ the time, we would not similarly and necessarily find our viewpoint changing as time advanced.»<sup>73</sup>

The hierarchy of Chadabe is not needed: both of his stages are equally potent *writing processes*, and they can both be represented as pending transactions. A situation of composing-performing may be preceded by a variable amount of composing “outside-time”, creating structures that become then available as building blocks during the real-time situation. This is indicated by the enclosed box in the top part of Fig. 2.14b. The extent to which this prescription of structures constrains the freedom of the composer-performer is reflected by the angle of “possible futures” at each instant of the performance.

It must be made clear that, although the  $\mathcal{T}_{(P)}$  box in the top of Fig. 2.14b looks like the miniaturisation of 2.14a, the coupled model is not something that *follows* the decoupled model, but the decoupled model is rather a special case in a system which oscillates between the two access modes of time. If the diaphragm has been used as a metaphor for a strictly decoupled model, in this oscillatory model the mirror would be an appropriate image. Assuming the “piece” in Fig. 2.1 to be a mirror, it means that each side can infinitely reflect each other; the piece is just the frame for an ongoing writing process in which the human composer «... or any other agent,

---

<sup>73</sup>Marsden, *Representing musical time: a temporal-logic approach*, p. 17.

whether human or machine . . . »<sup>74</sup> observes and reacts *in time*, either coupled or decoupled to a real-time flow.  $\mathcal{T}_K$  translates into  $\mathcal{T}_{(P)}$  and vice versa.

The mirror model most closely matches the direction of impact devised by Di Scipio. In his concept of “audible eco-systems”,<sup>75</sup> such a system is characterised by self-observation, determining «. . . its own internal states based on the available information on the external conditions – including the traces of its own existence [!] left in the surroundings.» We will misread the second part as traces of its own existence left observable in the *data structure*. This type of system reminds us of the self-regulating aspect of process that has so far been neglected, and reiterates one of Marsden’s remarks, the unimportance of attributing agency to either human or machine. Since an event can become—within the performance—a singularity, a new source of information and transformation, it is a model of generative music pieces or sound installations. ‘Generative’, strictly defined, means that the actual is not just a choice from the virtualities, but the virtualities themselves are constantly renewed by the actual.<sup>76</sup> This topic will reappear in Sect. 4.4.6 as the differential reproduction of a medium through the succession of forms.

A system unifying both time layers inside a model which provides for their reciprocal translations will thus make tape composition, algorithmic composition, live electronics, live improvisation, and generative sound installation become just locations within a common plane.

It is the responsibility of the joint between the two layers, providing the source of these translations, to keep the different forms liquid, in order to avoid tape pieces turning into monoliths (flattening the solution space to one path), but also to enrich real-time systems with temporal controls (a looser coupling) for an emancipation beyond “music instruments”.

## 2.9 Conclusion

With the contours of the compositional process becoming more and more acute, its product, the “piece”, becomes more and more specious. Is the structure the piece? The diaphragm

<sup>74</sup>Marsden, *Representing musical time: a temporal-logic approach*, p. 17.

<sup>75</sup>Agostino Di Scipio (2003), ‘Sound is the interface’: from interactive to ecosystemic signal processing’, *Organised Sound* 8(3), pp. 269–277.

<sup>76</sup>cf. Gilles Deleuze and Claire Parnet (1996), ‘L’actuel et le virtuel’, in: *Dialogues*, Paris: Flammarion, pp. 179–181

model provided the triple assurance of the piece: It is process product, a refractor of time, and a transportable object. But if refraction is changed for reflection, if the performance “writes back”, is the piece now the sum of structure plus performance(s)?

The second-order, deferred writing is character trait of the algorithm. It may be fully exhausted on the composer’s desktop, translated to lexicographically fixed data. But it may also be kept in motion, adding fragility to the transportation; threatening the medial fixation, most obvious in tape composition. A fixation which is dubious, because it trades the agility of random access, *total recall*, the bird’s eye perspective of the composer, for a withering of the solution space. The reproducibility of the work in the composer’s studio ignites his desire of complete *control* over the result, a sort of teleology.

The algorithmic composer might also not be immune to this trap, but he is inevitably and constantly thrown back to an algorithm’s potentiality, its possibilities and limitations, the struggle with its implementation in a programming language; that is to say, here the arrow of time takes a more curvilinear shape, the tools of production are unstable and meld with the actual works of art. The eigen-motion of the process surfaces.

This does not mean that a distinction between composing and performing might not ever be useful, nor that one could not speak of “a piece” and that this piece has first been composed, then performed. But we propose a shift in perspective where these are simply specialised views of the activity of composing which often obstruct more than they reveal. What is needed is a deconstruction of the compositional process in order to uncover a new understanding of it.

There are interesting new approaches emerging that are beginning to change the view of what composing means. An example has been given with the “audible eco-systems” of Di Scipio, although a mere turnaround of “interactive composing” to “composing interactions” might not be enough. In order to make the tracing of the compositional process productive *for the unfolding of the process itself*, the notion of interaction should be completely dissolved, leaving only a plain system of traces.

The endeavour is to reattribute time to what is mistakenly called a reasoning “outside-time”. While several models of the compositional process have been proposed in the literature, this very

manifest notion of a compositional time is almost entirely absent.<sup>77</sup> This time is depersonalised and void of psychology, and we will justify this approach in the following chapters through recourse on Hans-Jörg Rheinberger's epistemic thing which peels off from the researcher and is primarily constituted as material trace and not linguistic signification.

---

<sup>77</sup> A notable exception is the integration of "cycle time" into Emerson's model, although it is put into the context of psychology and cognition and not interwoven with performance time (Emerson, 'The Relation of Language to Materials').

## Chapter 3

# Beyond Control and Communication

The purpose of this chapter is to develop an appropriate methodology to research compositional process. It begins by reviewing some existing models of this process and questioning its contextualisation as a problem-solving activity. We argue that the idea of goal-directed behaviour, rooted in cognitive psychology and cybernetics, contributes to the disappearance or neglect of the traces of the process. Revealing how the definition of goal-directedness is tautological brings us to the core problem: the requirement for and limitations of representational forms. Often representations are based on the postulation of similarities, analogies and imitations. But representations also go beyond that which is represented. For example, operational closure is an important property of creative systems. Establishing connectivity in the action repertoire, it allows the process to be kept in motion. Having an efficient representation of (infinitely) concatenated transformations through the notation of “powers” gives us an identity handle of a process, a symbolic notation for a temporal suspension which can even expose the uselessness of relating to the origin of a process. An alternative deparadoxification is the construction of hierarchies, also noticeable in the distinction between the researcher and what is observed. Here the operational closure is broken, bringing into question the validity of research which does not feed back into the artistic system. We prepare instead for a methodology which employs the artistic process and self-observation, focusing on the material trace instead of the deciphering of a communication chain.

The previous chapter introduced creation time as the domain in which compositional processes take place. And while musical time has been extensively studied inside and outside of computer music, only the temporal values eventually heard as performance sequence have been accorded musical status; the creation time has been studied only peripherally and implicitly without naming it as such. Possible explanations arise from two related aspects which have been

addressed. The “masterwork syndrome” (Emmerson) affords the idea of the perfected work, with regard to which the different tested paths are external. The iterations of a perfection process are just incomplete approximations that are *superseded* by each new cycle. More profanely, a composer might not realise, give importance to or admit groping and adjustment of rules during the process.<sup>1</sup>

This type of work also peels off from the composer, so the composer’s physical or mental motions might simply be seen as extra-musical. It leads to the other aspect, the reduction of the composer to a “bringer of structure” (Lewis), whereas any embeddedness in real-time is discounted. Existing research does not have much empirical evidence of the traces of the compositional process, relying mostly on interviews with the composers, or snapshots of computer artefacts such as MIDI files. In an ex-post case study by C. Burns, there are severe gaps in the snapshots of one of the music programs he looks at, and he notes that:

«Composers are generally more interested in producing work than in documenting it. Sketches and drafts are often saved only if their continuing availability is necessary for the completion of a project, and mistakes and false starts are unlikely to be preserved.»<sup>2</sup>

We suspect, however, another, further-reaching reason for the negligence. It stems from a sort of “double bind” embodied in the seemingly opposite sides of goal-directed models of the composition process. These two sides could be identified as ‘control’ and ‘communication’, and the aim of this chapter is to study their appearance and context. It will be demonstrated that, in this context, traces function merely as warrantors of repetition or indicators of originating goals. Being seen purely as *signifiers*, they are omissible and therefore have not caught much

<sup>1</sup>B. Becker and G. Eckel who were investigating the interaction of composers with computer systems conducted a number of interviews. They found that many composers tend to refuse to explicate or they mystify their practice, and the accounts they give are strongly formed by the social and cultural discourse in which they are embedded. Barbara Becker and Gerhard Eckel (1995), *Künstlerische Imagination und Neue Medien: Zur Nutzung von Computersystemen in der Zeitgenössischen Musik*, tech. rep. Arbeitspapiere der GMD No. 960, St. Augustin: German National Research Center for Information Technology (GMD); for a short English summary, see Barbara Becker and Gerhard Eckel (1996), ‘On the Use of Computer Systems in Contemporary Music’, in: *Proceedings of the 22nd International Computer Music Conference (ICMC)*, Hong Kong, pp. 118–120

<sup>2</sup>Christopher Burns (2002), ‘Tracing Compositional Process: Software synthesis code as documentary evidence’, in: *Proceedings of the 28th International Computer Music Conference (ICMC)*, Göteborg, pp. 568–571.

attention in previous research. The next step then is to bring the traces out of the shadows, and reinstate them as the manifestation and condition of a reproductive process driven forward by connectivity.

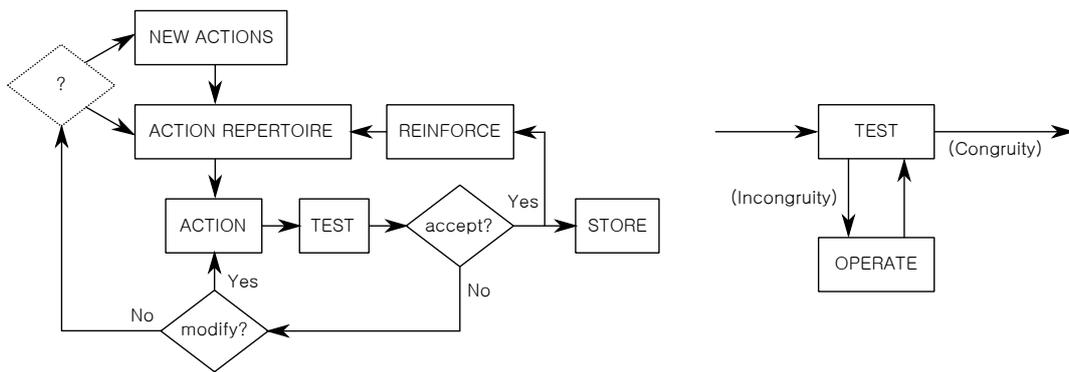
### 3.1 Composers in Control

When B. Eaglestone et al. formulated the requirements for a composition system, they found two functions:<sup>3</sup> Firstly, it must allow the composer to create, manipulate, store and retrieve musical artefacts—this is what has been discussed under the rubric of ‘access’. This function they call “service”, and it must be complemented by a second function, which is to «provide an environment within which those services can be used creatively.» Their hypothesis is that there is a discrepancy between the advances in general software engineering, reflected in the development of computer music systems, and the enabling of creativity, which they link to the idea of “divergent thinking” coined by psychologist J. P. Guilford. Software engineering accordingly would focus on “convergent” logic, meaning that it tries to implement the single optimal solution to a given task, whereas the idea behind divergent thinking is that multiple answers from different areas (thus divergent) can be found and synthesised to complete a given task.

Their interest in the support for creativity in computer music (in particular electroacoustic music) led them to study the compositional process, and their paper contains a review of related work of such studies and possible methodologies. As has been said before, not many systematic studies have been conducted, and those which exist are limited since they mostly focused on traditional instrumental composition, took the angle of musicology and education, or concentrated on a particular composer with the danger of producing results which reflect that composer’s idiosyncrasies rather than general approaches.

---

<sup>3</sup>Barry Eaglestone et al. (2001), ‘Composition Systems Requirements for Creativity: What Research Methodology’, in: *Proceedings of Mozart Workshop on Current Research Directions in Computer Music*, Barcelona.



(a) Elaborated model of the compositional process (Emmerson)

(b) TOTE unit (Miller, Galanter and Pribram)

Figure 3.1: Test-cycle based models

Two of the studies they highlight have been mentioned before: the case study of Collins<sup>4</sup> and the theoretical reflections of Emmerson.<sup>5</sup> The former is closer to the agenda of Eaglestone et al., focusing on creativity and education and using similar research instruments (case studies with a mix of qualitative and quantitative empirical methods), and the latter is closer to our own agenda, focusing on the praxis and the development of the language of electroacoustic composition. Yet both produce diagrammatically represented models of the compositional process which share some similarities.

### 3.1.1 Test Operate Test Exit

The discussion will start from Emmerson’s diagram, depicted in Fig. 3.1a, which is an elaborated version of the simple model that had been shown in Fig. 2.13. It is still built around the core loop of the simple model which closely reproduces the Test-Operate-Test-Exit (TOTE) cycle (Fig. 3.1b) postulated by G. A. Miller, E. Galanter and K. H. Pribram (1960) in their outline of cognitive psychology.<sup>6</sup> The TOTE was introduced as an alternative to the previously dominant Stimulus-Response pattern by the behaviourists, in an attempt to generalise the test mechanism

<sup>4</sup>At that time they referred to the early 2001 PhD thesis of Collins, while we will refer to the newer article, David Collins (2005), ‘A synthesis process model of creative thinking in music composition’, *Psychology of Music* 33(2), pp. 193–216.

<sup>5</sup>Simon Emmerson (1989), ‘Composing strategies and pedagogy’, *Contemporary Music Review* 3(1), pp. 133–144.

<sup>6</sup>George A. Miller, Eugene Galanter and Karl H. Pribram (1960), *Plans and the Structure of Behavior*, New York: Holt, Rinehart and Winston, Inc.

internal to an organism and overcome the otherwise plainly reflex (threshold) based concept of S-R.

The test stage is basically a comparison of an input with a *desired output*, the TOTE unit thus becomes a sort of *state switch*. The organism is previously in state of incongruity,<sup>7</sup> and the operation is performed until the test indicates that a state of congruity is reached. Characteristic is the mainly mechanical exemplification, such as hammering a nail until it is flush with the wall.

At first glance then the diagram in Fig. 3.1b is a temporal chart, where time flow corresponds with motions along the arrows. But what exactly “takes time”? Miller, Galanter and Pribram distinguish three levels of abstraction with regard to the transport of the arrows. The lowest level is *energy* such as in a physiological perspective, a view which neither the authors nor we are interested in. The other two levels are *information* and *control*. This comes as no surprise, as the authors try to apply the ideas of cybernetics, a young discipline at the time, to psychology, and cybernetics in turn draws inspiration from C. E. Shannon’s mathematical theory of communication.<sup>8</sup> The three possible interpretations also appear in an introduction to cybernetics by W. R. Ashby, whereby cybernetics can be defined as «the study of systems that are open to energy but closed to information and control».<sup>9</sup>

Control is the most “intangible” force, and Miller, Galanter and Pribram also seem uneasy with the term, suggesting one should revert to using ‘temporal succession’. The notion of control is needed for their approach, as one can easily see that a single TOTE unit is of limited descriptive use. Referring back to Emerson’s model, it could be said that ACTION is some form of manipulation of the musical material and structure by the composer, followed by a TEST which makes judgements over these recent changes. If, according to the judgement, the changes have contributed positively towards some imagined aesthetic goal, they are kept (STORE), otherwise they are rejected and a new adjusted ACTION performed. Now this does not account for any

---

<sup>7</sup>Miller, Galanter and Pribram use the term in favour of “difference” as it allows the conception of TOTEs where operations not necessarily require a difference.

<sup>8</sup>Claude E. Shannon (1948), ‘A Mathematical Theory of Communication’, *Bell System Technical Journal* 27(3), pp. 379–423.

<sup>9</sup>W. Ross Ashby (1956), *An introduction to Cybernetics*, London: Chapman & Hall.

further structure of the compositional process, e.g. the motion from macro to micro structure or vice versa. And therefore, implicit in the TOTE model is the possibility of concatenating units or, more prominently, the hierarchical nesting of units. The nesting occurs within the operate (ACTION) phase, so a more differentiated model of the process could be that one imagines an outmost TOTE test “Is the composition finished?”, and if the answer is ‘No’ and control returned to the operate phase, we may find here another test such as “Is the material exhausted?” If the answer to the subordinate test is ‘No’, operate by developing some material, if the answer is ‘Yes’, operate by arranging or rearranging the existing material, etc.

In this process, essentially driven by *feedback*, time elapses (and energy might be consumed), but control is conserved—it is handed over to successive stages and eventually returns after a full cycle. But what is control exactly? An alternative definition of cybernetics is the study of “control and communication in the animal and the machine” (the title of N. Wiener’s monograph<sup>10</sup>), the word cybernetics deriving from the Greek word for *steersman*. We shall look closer at this take on the matter. Here, control is based on regulation within a machine,<sup>11</sup> where a regulator’s function is to shield the machine’s state from disturbances from the machine’s environment. This is described in detail by Ashby,<sup>12</sup> and his diagrammatic arrangement is shown in Fig. 3.2a. It is important to remember that cybernetic diagrams are based on functional relations and not in any sense physical parts of a machine. Indeed, depending on the *observer* or the question at hand, the functions and relations may change.

Nevertheless, the components are most easily understood with a mechanistic machine, for example a thermostatically controlled water-bath. The regulator  $R$  is designed to keep the essential variable  $E$  of the system, here the water temperature, within a defined range of permissible values  $\eta$ . The value of  $E$  is threatened by a disturbance  $D$  affecting the machine.  $D$  is outside the control of the machine (no arrow points towards it); in the example it could be ambient temperature changing due to weather conditions, or cold or hot objects entering the bath. Be-

<sup>10</sup>Norbert Wiener (1948), *Cybernetics, or Control and Communication in the Animal and the Machine*, New York: John Wiley & Sons.

<sup>11</sup>Ashby uses the term machine very inclusively as «all possible machines», which may comprise anything from living organisms to theoretically postulated machines; Ashby, *An introduction to Cybernetics*, §1/3

<sup>12</sup>*Ibid.*, chap. 10–11.

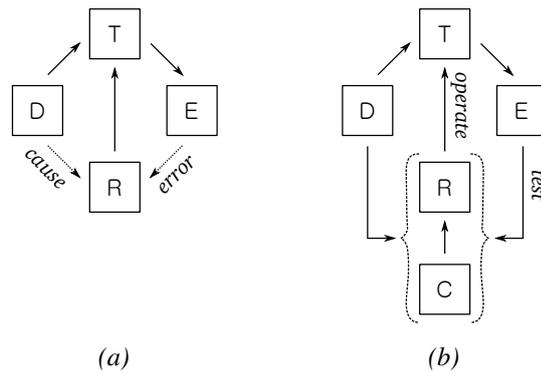


Figure 3.2: Regulation and control (Ashby)

yond the passive shielding of  $E$  from  $D$  (e.g. by having an insulator),  $R$  takes an active role by inducing counter-measures  $T$  which transform the effect that  $D$  has on  $E$ , such as starting to heat or cool the water. The regulator can be controlled by pre-embedded knowledge about the effect of the ambience (possible disturbances) on  $E$ —this is called *cause*-controlled. But often the knowledge about  $D$  is received by  $R$  indirectly as a feedback from the outcome  $E$ , in which case the term *error*-controlled regulation is used, since  $R$  behaves in reaction to the discrepancy (error) between the observed value of  $E$  and the desired subset  $\eta$ .

Based on information theory, Ashby is now able to establish a relation between disturbance, regulator and outcome of the essential variables, casting it into the ‘law of requisite variety’. He calls *variety*  $V$  what Shannon calls information entropy—the distinct number of states a variable can take on and measured logarithmically as bits. The law has the simple form of  $V_E \geq V_D - V_R$ . It states that the variety (uncertainty) transmitted by a disturbance onto the essential variable is bounded by the variety of the regulator. The more information the regulator has about the disturbance, the more varied its response and counter-actions are, and the more strongly it can block the influence of the environment on a system. If for every possible input from the outside, the regulator has an effective counter-action, the system would be under total control ( $V_E = 0$ ).

How does this relate to the compositional process? We are trying to elucidate what drives the process and how it can be observed, and we have come to see a model of process informed by the psychological construct of a TOTE cycle which in turn is animated by the “flow of control” (control is what takes time). An explanation is required as to how the control loop

of the regulator comes into being. Ashby explains the a priori of the desired subset  $\eta$  of the essential variables that guides the regulator through the addition of a new “input” to the system, the controller  $C$ . It is included in Fig. 3.2b, where also the two phases of TOTE are marked as the communication from regulator to transformation (operative instruction) and the feedback of the loop’s outcome for successive comparison (test instruction). The function of the controller is to decide on a target or goal to set ( $\eta$ , not explicitly shown). The communication from  $C$  to  $R$  is the action of setting the target. The curly braces hint at the fact that another feedback cycle involving the controller may be considered, in which it adjusts the target.

For the moment it can be assumed that the controller is the composer instructing a computer system to manipulate musical data. The disturbances describe the collection of aspects of the system not under his control. They are therefore—speaking strictly in functional cybernetic terms—actually external to the system, which may be defined provisionally as «a list of variables» under the control of an experimentator/controller.<sup>13</sup> Given some preconceived or incrementally refined ideas about the desired musical output, he would instruct the regulators provided by the system (e.g. using tools or algorithms in a software) to come as close as possible to the desired result.

Emmerson is aware of the implications of building the model around the TOTE unit, noting that it «has an inherently conservative streak» because «a successful group of actions would be reinforced and a stable repertoire established for endless use . . . »<sup>14</sup> If a system’s behaviour is defined as the trajectory of states (outcomes), the iterative testing would tend towards an equilibrium. In the case of the system appearing as complete black box (the composer has no knowledge of how it works and what effects his actions have) an initial strategy would be to give it random input. But even then, behaviourists have observed that at some point a desirable outcome will occur and due to the feedback (some form of gratification) that particular behaviour would be reinforced and appear with increasing frequency.<sup>15</sup> This is of course but a specific case,

<sup>13</sup>Ashby, *An introduction to Cybernetics*, §3/11.

<sup>14</sup>Emmerson, ‘Composing strategies and pedagogy’.

<sup>15</sup>The account of B.F. Skinner’s experiments, as given in Arthur L. Loeb (1991), ‘On Behaviorism, Causality and Cybernetics’, *Leonardo* 24(3), pp. 299–302

since feedback is interpreted as successive stimulus which strengthens a reflex<sup>16</sup>, and artists certainly exhibit more complex behaviour than a collection of reflexes. If a hierarchical TOTE is assumed, there may be many factors contributing to the test, including social aspects such as subscribing to a certain movement or aesthetic programme, but also trying to innovate, the desire of the classical avant-garde to distance itself from the previous programmes. Therefore:

«... the TEST procedure is neither absolute nor stable. The model is embedded within the social psychology of its real time. The term ‘judgement’ has a paradoxical component: we must establish sufficient agreement to allow communication, but build in the ability to evolve to suit changing situations ... To survive in an evolving world the variability of TEST demands an input to ACTION REPERTOIRE called ‘NEW ACTIONS’.»<sup>17</sup>

Emmerson refuses to explain where the new actions come from, since «it doesn’t matter»; of greater importance is the fact that the model is kept operationally open to this intrusion from the environment. There is, however, another remark that needs to be made within the context of cybernetic modelling. The last quotation using the terms “survive” and “variability” again strongly alludes to Ashby’s language. It has already been said that the TOTE model allows the hierarchical nesting of units, and thus it is also possible to turn the table around in Fig. 3.2b, in which case testing might be an internal function of the regulatory part of the control loop. Perceptiveness and adaptiveness to an evolving world formulated as artistic goal therefore still strangely translate to a highly varied regulator, where  $\eta$  becomes the avoidance of “lagging behind” development in the world.

The functional agility of the cybernetic approach is thus both its weakness—because almost any hypothesis can be valid, given the right formulation, and its strength—taking a constructivist approach which emphasises the dependency on the observer. For example, Fig. 3.3, adapted from F. Heylighen and C. Joslyn,<sup>18</sup> is a symmetrically drawn version of a control system. From the

---

<sup>16</sup>cf. Miller, Galanter and Pribram, *Plans and the Structure of Behavior*, p. 30

<sup>17</sup>Emmerson, ‘Composing strategies and pedagogy’.

<sup>18</sup>Francis Heylighen and Cliff Joslyn (2001), ‘Cybernetics and Second Order Cybernetics’, in: *Encyclopedia of Physical Science and Technology*, ed. by Robert A. Meyers, vol. 4, New York: Academic Press, pp. 155–170.

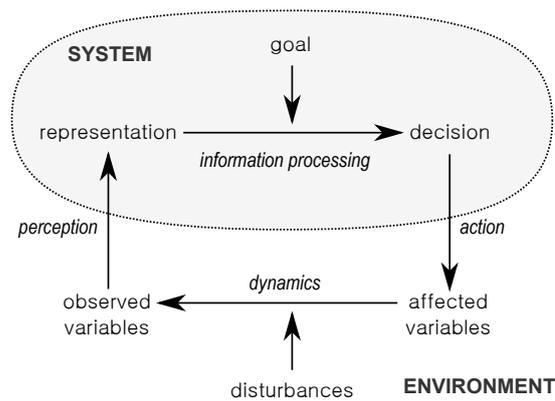


Figure 3.3: Control system (Heylighen and Joslyn)

system's perspective, its environment is just a generalised external system for which boundaries other than to itself have simply not been established yet. What it perceives as disturbances (uncertainty) might well be the flow of information within this hypothetical externalised system. In a careful consideration of our setup, there are at least four systems: Composer, composition software, composition software designer, and computer music researcher. Each is the other's mutual environment, theoretically requiring the elaboration of six pairwise relationships between them.

### 3.1.2 Goal-Directedness

Another point of interest concerning Fig. 3.3 is that the controller has become *implicit*, while the goal set by the controller is *explicitly* included. This is peculiar because previously the goal could have been seen as a future state: the controller sets the goal as target, the state to reach, the controller is the origin of the goal; but now it emphasises the goal as the origin of the system's dynamic; it is as if the goal has always been there and seems to indicate the past of the system. The setting and attaining of the goal essentially appear to annihilate time: they form an "outside-time" operation. In an early article by A. Rosenblueth, N. Wiener and J. Bigelow,<sup>19</sup> predating the coining of the term cybernetics, the authors are implicated in this unclear temporal orientation. They try to scientifically rehabilitate the term 'teleology', defining it as "purpose controlled by feed-back". Any Aristotelian connotation is disavowed by the statement that «The

<sup>19</sup> Arturo Rosenblueth, Norbert Wiener and Julian Bigelow (1943), 'Behavior, Purpose and Teleology', *Philosophy of Science* 10(1), pp. 18–24.

concept of teleology shares only one thing with the concept of causality: a time axis.» This is however wishful thinking, as they get caught up in tautological definitions. First, the definition of purposefulness:

«The term purposeful is meant to denote that the act or behavior may be interpreted as directed to the attainment of a goal – i.e., to a final condition in which the behaving object reaches a definite correlation in time or in space with respect to another object or event. Purposeless behavior then is that which is not interpreted as directed to a goal.»<sup>20</sup>

It is clearly a restatement of a *final cause*, although remarkably under the condition that purpose is an attribution made by an observer. In an attempt to reestablish some form of objectification, Rosenblueth, Wiener and Bigelow base the concept of purpose on «the awareness of “voluntary activity”», describing the latter as follows: «When we perform a voluntary action what we select voluntarily is a specific purpose, not a specific movement.»—unequivocally a circular definition. Indeed, a review of the goal construct in psychology by A. J. Elliot and J. W. Fryer<sup>21</sup> begins with Aristotle, for whom «the origin of action . . . is choice, and that of choice is desire and reasoning with a view to an end.»<sup>22</sup> There is therefore no human behaviour without purpose. In psychology itself, the usage of goal (in German: *Ziel*) can be traced from the beginning of the 19th century, although at that time it was not yet used systematically. The frequent combination with *movement*, such as “movement towards an end”, shows an interesting intersection with the conceptual history of *process* (cf. Sect. 2.1.1). After the turn of the 20th century, goals were often seen—e.g. by S. Freud or W. McDougall<sup>23</sup>—as auxiliary to human instincts which directed all human behaviour. In the behaviourist perspective of the 1920s, the attempt was made to objectify goals through the observation of something “persistent”. E. C. Tolman writes:

---

<sup>20</sup>Ibid.

<sup>21</sup>Andrew J. Elliot and James W. Fryer (2008), ‘The Goal Construct in Psychology’, in: *Handbook of motivation science*, ed. by James Y. Shah and Wendi L. Gardner, New York: The Guilford Press, pp. 235–250.

<sup>22</sup>From *The Nicomachean Ethics*, cited in *ibid.*

<sup>23</sup>McDougall is also seen by other authors as the first exponent of 20th-century psychology to create a goal taxonomy: James T. Austin and Jeffrey B. Vancouver (1996), ‘Goal Constructs in Psychology: Structure, Process, and Content’, *Psychological Bulletin* **120**(3), pp. 338–375

«It is this purely objective fact of persistence until a certain type of goal-object is reached that we define as a goal-seeking.»<sup>24</sup> Tolman furthermore lays the foundation for a hierarchical conception of goals, by distinguishing ultimate goals (physiological equilibrium of the organism, e.g. “cease hunger”) from subordinate goals, which can be observed as persistent behaviour (“find food”).

Elliot and Fryer see the term ‘goal’ fully established in psychology in the 1930s, with additional conceptualisations then appearing in the 1950s through cybernetics. According to their review, in the contemporary literature at least three distinct definitions of goal are used, which can be seen as three degrees of indirection:

- (1) Goal as a network of variables which create an orientation towards behaviour (double indirection)
- (2) Goal as the purpose *for* behaviour (single indirection)
- (3) Goal as the aim *of* behaviour

While Elliot and Fryer try to find a uniform definition of goal, arriving more or less in the middle while attempting an outright rejection of any cybernetic ideas («Goal-directed behavior is proactive, not reactive»<sup>25</sup>), other authors have used this divergence to create a taxonomy of goals. For example G. Pask defines goal-setting as necessary elements of theory building:<sup>26</sup> To be able to make predictions about a model which is used as a prescriptive blueprint for a system, one implies a purpose *for* the system. If a theory is built by creating a model from an existing system, one implies a purpose *of* the system. Moving from observer to designer,<sup>27</sup> goal in a narrow sense is the purpose *in* the system. Either this goal is built into it but can only be indirectly discovered by an observer who equates purpose *in* and purpose *for*; or we are dealing

<sup>24</sup>Cited in Elliot and Fryer, ‘The Goal Construct in Psychology’. Tolman calls this an objective fact, because the sentence is stated within an experiment of a hungry rat chasing for food in a labyrinth, and the “physiological condition of hunger” is taken as an additional granted fact. Even if we accept hunger as a physiological condition, it becomes clear that the transfer to cognitive “behaviour” is far less convincing if no substitution for this objective physiological fact is postulated.

<sup>25</sup>Ibid.

<sup>26</sup>Gordon Pask (1969), ‘The meaning of cybernetics in the behavioural sciences (The cybernetics of behaviour and cognition; extending the meaning of ‘goal’)', *Progress of Cybernetics* 1, pp. 15–44.

<sup>27</sup>See Fig. 3.5 later in the chapter.

Descriptive Model Purpose <i>of</i>	Prescriptive Model Purpose <i>for</i>	Goal-directed System Purpose <i>in</i>
--	--	---

Table 3.1: Goal taxonomy of Pask

with a language oriented system which has a *general purpose*, being able to accept new goals that the user must state in a programming language. This classification is shown in Table 3.1.

Despite all of this, the ontological status of ‘goal’ remains unsolved—a typical indicator of a problem of self-reference. J. T. Austin and J. B. Vancouver have conducted another study<sup>28</sup> in which they try to tackle this problem by analysing the *structure* and *process* of goal concepts. In the review of dimensions used to define goals, they again find divergent positions. A subject could conceptualise goals along the dimensions of ‘expectancy’, ‘value’, ‘ease’, and ‘clarity’, whereas an external observer might use the dimensions ‘commitment’, ‘origin’ and ‘self-efficacy’. Beyond the idea of “desired state” (which has been shown as tautological above) they even postulate “virtual goals” for which no internal representation of desired state exists.<sup>29</sup> They take as an important indicator of the existence of goals the property of *equifinality*, which means that a state of equilibrium may be reached through different strategies or behaviours, while at the same time stating that equifinality does not *require* an underlying goal. A discrimination is proposed where goals «must drive some processes in the organism» to be of interest, which again is circular reasoning—only goals are researched which drive processes, and the observation of processes that are “driven” (towards some direction) guarantees the existence of goals.

### 3.1.3 Problems and Solutions

A possible exit strategy might be to begin at an initial stage where there are no goals (yet). It seems permissible to assume that there are no predefined goals that cause the desire in a human being to start a compositional investigation.<sup>30</sup> There might be various motivations, but not much

<sup>28</sup>Austin and Vancouver, ‘Goal Constructs in Psychology: Structure, Process, and Content’.

<sup>29</sup>The example of a virtual goal is the desire to avoid collision between multiple goals which a system wishes to attain.

<sup>30</sup>*Intrinsic* goals, leaving aside more external goals such as satisfying a client who has commissioned a composition.

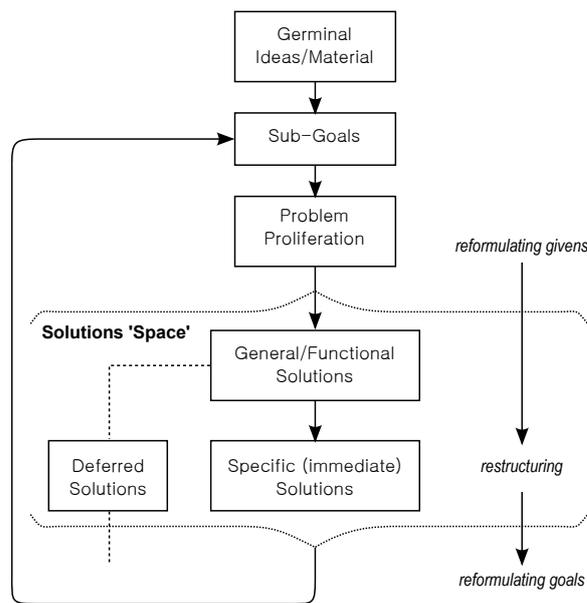


Figure 3.4: Synthesis process model (Collins)

can be said except that the composer aims to create *something*,<sup>31</sup> and that ‘something’ might initially be largely undefined.

Austin and Vancouver call this perspective “goal processes”, processes in which goals must first be established, will then be pursued, but may be continuously revised. The domain in which goal processes are operationalised is problem-solving, and it involves iterations of decision-making. The sequence of transactions which forms the creational timeline can be seen as the manifestation of decisions, and thus this perspective at first seems a fair match for the compositional process. A model based on problem-solving was developed by Collins, and an adapted diagram is shown in Fig. 3.4.<sup>32</sup>

Collins describes his model as a synthesis of different approaches to the compositional process which have been established in psychology and cognition research, namely a linearly progressing stage model and parallel and recursive problem-solving strategies. The model was constructed from a case study where a composer was asked to compose an instrumental score. Heterogeneous methodology is applied, combining interviews with data gained from incrementally saved MIDI

<sup>31</sup>So again, obviously any social and economic aspects are ignored for the sake of simplicity. For a brief account of these, see Becker and Eckel, ‘On the Use of Computer Systems in Contemporary Music’, §3.1.3

<sup>32</sup>Collins, ‘A synthesis process model of creative thinking in music composition’.

files and screenshots of the composition software. In particular, it was possible to observe how the composer developed two initial “themes” and how they moved *across* time  $\mathcal{T}_{(P)}$  *in* time  $\mathcal{T}_K$ , how variants of the themes were produced and interleaved, yielding what Collins called a “structural mapping”.<sup>33</sup>

Four aspects of this model are noteworthy. (1) Although sub-goals play an important role in the process, there is no explicit overarching goal, and in fact there is no explicit condition which indicates when a composition is “finished”—although the recursive nature of the process was accented by the way we redrew the diagram (subsequent stages are vertically distributed in the original paper). Collins explains the missing overarching goal by musical composition being an «ill-defined, as opposed to well-structured, problem-solving activity: the initial start and end states of a musical composition are imprecise.»<sup>34</sup> (2) The revision of goals occurs in cycles. A cycle begins with the *proliferation* of a problem, meaning that what the composer considers musical problems results from the currently given materials and structures, and as the materials and structures are developed, some problems are “solved”, others disappear, yet others shift their focus, and new ones appear. Thus the cycle is entered by a clarification of what the current “givens” are. Then, after a working phase—developing the “space” in which solutions directly derived from the recent problem statement as well as perhaps dormant “deferred” solutions and solutions coming from the restructuring of the givens emerge—the composer takes a look at the recent outcomes and reformulates the “goals”. (3) The situation is therefore not necessarily one where a problem incrementally shrinks, but creating new problems is an essential part of the creative process, and so Collins proposes to speak of a “solution space” instead of a “problem space”. (4) Finally, it can be seen that the composer may decide to put a particular solution (either developed or just hypothesised) aside for later consideration, creating a sort of reverberation along the creational time axis.

---

<sup>33</sup> A diagram is included in David Collins (2007), ‘Real-time tracking of the creative music composition process’, *Digital Creativity* **18**(4), pp. 239–256

<sup>34</sup> *Ibid.*

### 3.2 Models

Next, we wish to formulate a critique of the *modelling* methodology itself. What is the function of a model and how did it come into existence? Models share a close relation to systems, and they too are observer-dependent constructs. A straightforward case is the model of a system whose components or internal structures are known, for instance a model of the application of a theory—it associates the constants in the theory with elements in the applied domain, and the relations defined in the theory with relations between elements of that domain, while preserving the axioms of the theory (all statements that the theory makes, hold in the domain).<sup>35</sup>

Such a bijective or isomorphic association is not possible for opaque and observed systems. The requirement is relaxed, and now models are “only” required to consist of homomorphic mappings—elements and relations are defined that can then be identified in the observed behaviour of the system.<sup>36</sup> From the cybernetic perspective, a good model is a good regulator, because the experimenter as a system in its own right is coupled to the observed system *as his environment* (cf. Fig. 3.3), so the attempt to model that externalised system is an attempt to minimise disturbance (uncertainty) in the behaviour it exhibits.

Models are thus a re-entry of control at the level of the researcher, and the persistence and stability of a model are a quality measure in a positivist discipline. The establishment of a model is an important advancement in knowledge: «The creation of a model is proof of the clarity of the vision. If you understand how a thing works well enough to build your own, then your understanding must be nearly perfect.»<sup>37</sup> But then a composer is a highly complex system, and the behaviour to be modelled (which due to the complexity of the system must be *partial*) becomes even more dependent on the observer.<sup>38</sup> The model explains only a certain aspect of the system and is equally a representation of the system and of the researcher, whose hypotheses are a necessary element for building the model which must now be called *heuristic*.

<sup>35</sup>See Alan Marsden (2000), *Representing musical time: a temporal-logic approach*, Lisse: Swets & Zeitlinger Publishers, pp. viii–ix

<sup>36</sup>Heylighen and Joslyn, ‘Cybernetics and Second Order Cybernetics’.

<sup>37</sup>Miller, Galanter and Pribram, *Plans and the Structure of Behavior*, p. 46; a very similar statement is found in the introduction of Lars Löfgren (1968), ‘An axiomatic explanation of complete self-reproduction’, *Bulletin of Mathematical Biophysics* 30, pp. 415–425

<sup>38</sup>Cf. Ashby, *An introduction to Cybernetics*, §6/14

An example of such a hypothesis in the cognitive research field highlighted so far is given by Nick Collins: «It is perfectly possible that models devised by humans can capture something important about human music.»<sup>39</sup> It could hardly be more cautious. David Collins changes the hypothesis for an underlying motivation, speaking with the voice of Otto Laske: «I wish we could read people’s brains when they are engaged in music-making.»<sup>40</sup> It is this motivation which *configures* the model-building of the investigation, and as such the model owes as much to the findings of the case study as to this preconfiguration, within which goal-directedness serves as an axiom. There is however a slight distortion in this quote, since in the original context—Laske interviewing M. Minsky<sup>41</sup>—it served instead as a trigger for Minsky who more strongly articulates this wish to look directly into the brain.

### 3.2.1 Simulation and Analogies

A closer look at this interview is beneficial, because it clarifies the aspects of methodology and research objectives raised so far. The interview moves around the interaction of music understanding and artificial intelligence, of which Minsky as former director of the MIT’s artificial intelligence group is an authority. At the heart of AI is problem-solving, and in this context music-making can be seen as a vehicle for studying human action. Action and decision-making in turn are linked back to cognitive processes which it primarily aims to study. Minsky names Freud as the first to introduce a usable theory of thought based on goal-seeking, which was then operationalised in the 1960s in the form of computer *simulations*.

The strong emphasis on simulation makes—from today’s standpoint—the argumentation of Miller, Galanter and Pribram obsolete, since they were contrasting simulation with artificial intelligence.<sup>42</sup> For machines to be useful, they should implement, for example, chess games or language translations by closely *imitating* how a human actor would play chess or translate a text, as opposed to implementing the most efficient solution to playing chess or translating, and the latter they associated with “artificial intelligence”. The validity of imitation is an implicit assumption stemming from equating similarity with the homomorphism of the model. The

---

<sup>39</sup>Nick Collins (2009), *Introduction to Computer Music*, Chichester, UK: John Wiley & Sons, p. 298.

<sup>40</sup>Used as an epigraph in Collins, ‘Real-time tracking of the creative music composition process’

<sup>41</sup>Marvin L. Minsky and Otto Laske (1992), ‘A Conversation with Marvin Minsky’, *AI Magazine* **13**(3), pp. 31–45.

<sup>42</sup>Miller, Galanter and Pribram, *Plans and the Structure of Behavior*, p. 54.

validity of similarity is preserved even under duplication—the world is reflected in the mental *image* (image and imitation sharing the same etymological root, and both denoting “likeness”<sup>43</sup>), a model of the mental image created, translated into a simulation of the operations on the mental image, the simulation observed and compared with the theory, and thereby knowledge obtained about how humans act upon the world and make decisions, e.g. how a composer composes.

The fact that a simulation replaces “real” experimentation does not change the epistemological underpinnings.<sup>44</sup> Both are faced with problems of abstraction, inference and artificiality (design), and try to arrive at the same result—understanding how the brain works. For example, here Laske himself describes an experiment he conducted where both adults and children were given a composition task to be completed with the help of computer software that created a protocol of the actions:

«For the first time in music history, we are able to produce empirical traces of a musical process; we can then study such a process in terms of the actions it is composed of. Common sense tells me that a musical form derives from the process that produced it, and I would think, therefore, that the *control structure* of that process is intimately linked to the musical form emerging from it.» [Emphasis added]<sup>45</sup>

Minsky simply takes a more radical approach, in which the sketches and physical traces of composition will be made superfluous, «if, over another few decades, we find ways to more directly record a composer’s actual brain activities.»<sup>46</sup> Neither the composer’s body nor his emotions would be relevant any more, once «high-resolution brain activity imaging instruments» are available.

The idea of similarity and image is all pervasive in Minsky’s thinking. Strangely, it is his answer to tackle inductive reasoning: no actual thing can be abstracted, but «real things can be seen as

<sup>43</sup>Cf. Jacques Derrida (1981), ‘The Double Session’, in: *Dissemination*, trans. by Barbara Johnson, London: The Athlone Press, pp. 173–286, p. 188

<sup>44</sup>For example, see Roman Frigg and Julian Reiss (2009), ‘The Philosophy of Simulation: Hot New Issues or Same Old Stew?’, *Synthese* 169(3), pp. 593–613

<sup>45</sup>Minsky and Laske, ‘A Conversation with Marvin Minsky’.

<sup>46</sup>Ibid.

related—at least in an observer’s mind—by apparent similarities of structures, effects, or useful applications.»<sup>47</sup> As a result, while Laske uses common sense as a trampoline for developing a model, Minsky sees it as an end, claiming that to understand music a “common sense database” must be constructed using analogy (which he links to our mode of experience). The crux of the analogy becomes clearer when music is likened to language communication based on a transmission-reception model:

«The idea is that when a speaker explains something to a listener, the *goal* is to produce in the listener a structure that *resembles* a certain semantic network in the speaker’s brain.» [Emphasis added]<sup>48</sup>

Although an application of this model to music is not explicitly developed, in other places he uses “speaker” and “composer” conjointly, and furthermore proposes to apply a model of (pseudo-) storytelling to composition. The idea of transmittal of patterns and their observed similarity can be found in K. Krippendorff’s catalogue of metaphors used for communication.<sup>49</sup> The model is that of *cognitive sharing*, where sharing not only means to be part of something, but «being in some respect the same» or «thinking alike». Metaphors for this are the German “Mitteilung” and the Greek “Symbolon”. Mitteilung rather inaccurately translates to “message” or “notification” in English, but literally means “sharing (dividing) with”. The symbolon is a coin broken in half and shared between two friends with the purpose of reuniting them eventually. The prevalence of this model is reflected in the term “communication” itself, relating to “common”.

This model dresses as a «wolf in sheep’s clothing»,<sup>50</sup> in that it seemingly is symmetric (“community”) but ultimately requires the authority of a single observer: the “communicator” who judges whether the communication was “successful”, i.e. whether a thinking-alike can be ob-

---

<sup>47</sup> Ibid.

<sup>48</sup> Ibid.

<sup>49</sup> Klaus Krippendorff (1994), ‘Der Verschwundene Bote; Metaphern und Modelle der Kommunikation’, in: *Die Wirklichkeit der Medien; Eine Einführung in die Kommunikationswissenschaft*, ed. by Klaus Merten, Siegfried J. Schmidt and Siegfried Weischenberg, Opladen: Westdeutscher Verlag, pp. 79–113; an English manuscript exists as Klaus Krippendorff (1990), ‘Models and Metaphors of Communication’, *Annenberg School for Communication Departmental Papers (ASC)* (276), URL: [http://repository.upenn.edu/asc\\_papers/276/](http://repository.upenn.edu/asc_papers/276/) (visited on 13/08/2014)

<sup>50</sup> Krippendorff, ‘Der Verschwundene Bote; Metaphern und Modelle der Kommunikation’.

served. The illusion that a single authoritative observer might exist is embodied in the Laplace demon by which Minsky is chased—in *some decades, we will be able to spare a composer's disclosures as we have a complete image of the brain and its workings*. The brain is the key to the lost origins of music; evolution does not bear traces of the origin, as animals do not compose, but when the brain is understood, we will have found the origin.<sup>51</sup>

### 3.3 Notes from the Metalevel

A theory of the transmission of patterns from one medium to another is the most widespread in science, because it enables explanation and the building of generalities. H. von Foerster also described this desire for explanation as the root of the “cognitive blind spot”, operationalised through causation and deduction.<sup>52</sup> The presumed perspective, a non-interfering observer of entities interacting with each other, is nonetheless not without an alternative. Krippendorff contrasts it with the perspective of cognitive *autonomy* which sees human beings as “becomings”, as entities who continuously construct and deconstruct their non-shared realities. Instead of controlling and controlled subjects, these “auto-poets” may change rationality for aesthetics.<sup>53</sup> A resulting theory of communicative competence might thus be better suited to dealing with artists as agents, and also shift the weight from ‘goal’ to ‘becoming’ in the description of process.

But how can anything be stated without being observed? And how can that statement have scientific value without a form of objectification? An approach of cybernetics itself to solving these issues is known as “second-order cybernetics”. Von Foerster characterises it by a change of perspective from observed systems to observing systems. Since the researcher as observer is necessarily an active agent within the system that he observes, the situation can be improved by reflecting the role of observer, asking the researcher to observe himself in the role of the observer, or to observe other observers. This reflectivity is most palpable in several titles by von Foerster, e.g. *Understanding of Understanding* or *Cybernetics of Cybernetics or the Control of Control*

<sup>51</sup>Minsky and Laske, ‘A Conversation with Marvin Minsky’.

<sup>52</sup>Heinz von Foerster (1979), ‘Cybernetics of Cybernetics’, in: *Communication and Control in Society*, ed. by Klaus Krippendorff, New York: Gordon and Breach, pp. 5–8.

<sup>53</sup>Krippendorff, ‘Models and Metaphors of Communication’.

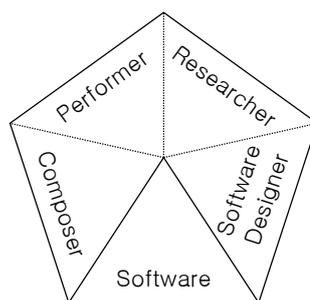


Figure 3.5: Observer roles in the thesis project

and the *Communication of Communication*. Instead of abandoning control and communication, it is elevated to a meta-level.

The theory of observing systems has been developed beyond cybernetics, the most important authors in this context being biologists H. Maturana and F. Varela, and sociologist N. Luhmann, who in turn took on the writings of the former. He describes the approach of second-order cybernetics as getting to know the distinctions guiding «the observations of the observed observer and to find out if any stable objects emerge when these observations are recursively applied to their own results. Objects are therefore nothing but the *eigenbehaviors* of observing systems that result from using and reusing their previous distinctions.»<sup>54</sup>

It seems that this second-order observation, the climbing of the ladder to the next meta-level, has not bought anything but breathing time. The problem still persists in “getting to know the distinctions”. If observation is tentatively equated with making distinctions (“the composer chose *that sound* and not another”), this activity is nothing but another distinguishing process which, to become tangible, must again be observed, and so on. The only way to confine these coupled observing systems within some boundaries is to include at some stage a self-observation, which indeed yields Krippendorff’s “auto-poets”.

### 3.3.1 Self-Observation and Boundaries

Fig. 3.5 shows a possible set of observers involved in our project. The technological manifestation of observing (or *facilitating observing*) is the software framework which will be described

<sup>54</sup>Niklas Luhmann (1993), ‘Deconstruction as Second-Order Observing’, *New Literary History* 24(4), pp. 763–782.

in detail in Chap. 5. It connects to multiple human actors or perhaps a single actor taking different roles, such as software developer, composer, performer, researcher. It is clear that the boundaries between these “sub-systems” are merely an abstraction,<sup>55</sup> for instance I as the person who drew the figure decided that a distinction between composer and performer is *significant*. Moreover, different people when faced with deciding whether an observed act is compositional or performative will define the boundaries differently—or draw a different distinction. And finally, the boundaries shift over time as the composer/performer changes roles or our perception of those roles changes. The fact that boundary establishment is a necessarily active ongoing process within a system is a key element in Luhmann’s theory and indeed closely linked to self-observation.

It can be said that the difficulties with self-reference are rooted in language, more precisely in the specific use of language, where an actor operates on or indicates an operand, both of which collapse in the self-indicating situation.<sup>56</sup> A classical example is the liar’s paradox, the statement “I am lying”. Speaker and subject in the utterance are the same but contradict each other. The mathematician G. Spencer-Brown has called this situation *re-entry* in the book *Laws of Form*<sup>57</sup> which led to an elaboration of self-referentiality by Varela and became an important reference for Luhmann as well. Since re-entry as a concept strongly resonates with our further discussion of the unfolding and traceability of the compositional process, it is worth reviewing it in greater detail here.

### 3.3.2 Re-Entry

Spencer-Brown develops a binary arithmetic and algebra—which might seem rather remote from our endeavour, however they are embedded in a “narrative” of observation. The narrative begins with an originary act of distinction, the drawing of a boundary. Spencer-Brown’s notation system is geometrically informed and often metaphorically interpreted and reveals surprising connections, which might explain why it has not been successful in the core realm of logic itself,

<sup>55</sup>cf. Robert L. Flood (1988), ‘Unleashing the “Open System” Metaphor’, *Systemic Practice and Action Research* 1(3), pp. 313–318

<sup>56</sup>Francisco J. Varela (1975), ‘A Calculus for Self-Reference’, *International Journal of General Systems* 2(1), pp. 5–24.

<sup>57</sup>George Spencer-Brown (1969/1979), *Laws of Form*, New York: E.P. Dutton.

where it was seen rather as a reinvention of Boolean algebra with an obscure notation.<sup>58</sup> Instead it worked as a generator of ideas in other fields such as biology, systems theory, philosophy and sociology. The two aspects of interest here are the form of distinction and the paradox of the re-entry of the form into itself.

The book begins with ‘distinction’ and ‘indication’ as givens (it thus begins with a distinction itself). Consequently, the act of distinction allows indicating either side of the distinction: that which has been distinguished, the ‘marked space’, from everything else, the ‘unmarked space’. Geometrically it can be represented as a circle, whereas for better compatibility of expressions later formed with the layout of written text, a mark  $\sqcap$  is used, the concave side of which is the “inside”. It is also called ‘cross’ to emphasise that it is not only the result of a distinction but the possibility for action (to cross). Two axioms are needed to construct what Spencer-Brown calls primary arithmetic: The spatial nesting or “order”, with the equivalence of  $\sqcap \sqcap = (\text{void})$  and the spatial sequence or “number”, with the equivalence  $\sqcap \sqcap = \sqcap$ .<sup>59</sup> These equivalences seen as equations enable them to act as two basic operations by which expressions (arrangements of crosses and voids) can be rewritten, typically reduced to conduct logical proofs, but also grown.

---

<sup>58</sup>cf. John Mingers (1995), *Self-Producing Systems: Implications and Applications of Autopoiesis*, New York: Plenum Press, §4.2; on the other hand, Luhmann points out that it is centred around a paradox which goes beyond logical contradiction, and therefore the core ideas (what he considers the core ideas) cannot really be handled by a logical discourse; Niklas Luhmann (1999), ‘The Paradox of Form’, in: *Problems of Form*, ed. by Dirk Baecker, Stanford: Stanford University Press, pp. 15–26, § VI

<sup>59</sup>It is beyond the scope of this text to go into more detail. But a somewhat intuitive explanation of the axioms is to see the nested mark as a double crossing, a going back from image via reflection to the source of the image; the sequential marks are a “a call made again” (a name called twice), thereby just confirming a value and not changing it. Luhmann questions the identity of the observer after the double crossing, however, cf. *ibid.*, § III



### 3.3.3 Power (of) Representation

This is remarkably similar to the state changes described by Ashby.<sup>60</sup> Here, a transformation can be subject to *closure*, meaning the application of an operator to a set of operands does not produce any elements which are not already contained in the set of possible input operands. Under closure, a sequence of transformations may be repeatedly invoked, and the repeated application of the transformations may be represented in the simplified form of powers, such as  $T^2$  meaning that transformation  $T$  is applied twice. The notation of powers resembles the substitution of the form for  $f$  and its reappearance on the other side of the equation in the *Laws of Form*, the difference being that in the former case operators are notated and in the latter case the operands. With Ashby's power notation, and using the appropriate transformations applied, the re-entry could be written  $(C1(C4(J2(C1(C5))))))^\infty$  or  $T^\infty$  if the composition of the five steps was substituted by  $T$ .

Having a stable representation,  $a$  and  $b$  can be tentatively assumed to reduce to either cross or void, and thus solutions of the second-order equation for  $f$  in all four possible combinations can be sought. Three solutions are determinate, but in one case  $f$  has two possible solutions and thus remains *indeterminate*:  $\overline{f|} = f$ . My own interpretation is as follows: The condensation of an ever-evolving arrangement (a pure process) into a form of representation may lead to a *useless* representation in terms of tracing the origin of this process—it is impossible to know whether  $f$  was originally void or distinction. Nevertheless, it may still be a *useful* representation in terms of showing the mode of production, i.e. indicating the possibility of future development.

With this representational form *as a tool*, on the other hand, it is possible to construct a paradoxical situation:  $\overline{f|} = f$ . Like the liar's paradox, it seems to have no solution.<sup>61</sup> Again, it is useless in terms of the origin of  $f$ . We observe a form, although we “know” that the coming into being of it is “impossible”. And again, it may be a productive form. Different strategies of de-paradoxicalisation<sup>62</sup> may be applied, all of which make use of the asymmetry<sup>63</sup> or directivity

<sup>60</sup> Ashby, *An introduction to Cybernetics*, ch. 2.

<sup>61</sup> Indeed the equation can be read as “ $f$  is equal to an indication of  $f$ .”

<sup>62</sup> This term was used in Niklas Luhmann (1989), ‘Law as a Social System’, *Northwestern University Law Review* 83(1 & 2), pp. 136–150

<sup>63</sup> Luhmann, ‘Deconstruction as Second-Order Observing’, § II.

of observation: First, a statement is understood as a *command*, and its form is the decision whether to obey it or not. Second, different *observers* are attributed to each side of the equation without them necessarily sharing intersubjective agreement, leading rather to an interaction of observers within a social system. Third, distinction as a timeless event which simultaneously yields both sides of the spatial cleft is distinguished from indication which *requires time* to cross the boundary,<sup>64</sup> and therefore the equation is rather a state machine or transformation directive.

### 3.3.4 Oscillation

While we are less interested in the first two possibilities, the last seems to reveal the motion aspect of process. Indeed, Spencer-Brown himself chooses to resolve the paradox by introducing imaginary states by way of a temporal “tunnel”. Already in the introduction, the pair distinction-indication is given self-referentially as a possibility of motion: «There can be no distinction without motive, and there can be no motive unless contents are seen to differ in value.»<sup>65</sup> With a temporal tunnel between inside and outside a cross, instead of growth in space, the representation (“outside-time”) as given above remains stable, while an observation “in-time” perceives an oscillatory pattern. It is a rather unusual approach, but complements the spatial metaphor used previously and may stem from Spencer-Brown’s background as an electronic engineer.

It is also not new with respect to other disciplines. Where the oscillations in the *Laws of Form*, due to its binary conception, are pulse trains, the general systems theory (GST)<sup>66</sup> developed by L. von Bertalanffy in the 1930s is based upon differential equations—also called “motion equations”<sup>67</sup>—and oscillations are patterns of continuous cycles. As a major influence in the foundation of cybernetics, GST introduced the notion of asymptotic equilibrium. According to von Bertalanffy, systems of differential equations (some of which are “general” in that they find application in different disciplines such as biology, physics and economics) can lead to

<sup>64</sup>Luhmann, ‘The Paradox of Form’, § III.

<sup>65</sup>The contents, e.g. inner and outer space severed by the mark, receive their value through “naming”, i.e. indication; thus the whole construction is circular. Spencer-Brown, *Laws of Form*, ch. 1

<sup>66</sup>Both spellings *general system theory* and *general systems theory* appear in the literature and von Bertalanffy’s work, perhaps due to the translations from German. Since it highlights the similarities across multiple systems, we keep using the plural here.

<sup>67</sup>Ludwig von Bertalanffy (1972), ‘The History and Status of General Systems Theory’, *The Academy of Management Journal* 15(4), pp. 407–426.

three different types of solutions: (1) Differences become arbitrarily small in the limit, thus like a damped oscillation settle on a stationary value; (2) the differences grow and no stability is obtained; or (3) a neutral stability is embodied by *imaginary* solutions, whereby «the system contains periodic terms, and there will be oscillations or cycles around the stationary values.»<sup>68</sup>

The last case, comparable to Spencer-Brown's re-entry, is characteristic of *open systems* (such as living organisms) in steady state. Here, despite material exchange with its environment, the system remains stable through continuous differentiation (oscillation). This is very similar to the concept of differential reproduction of H.-J. Rheinberger which will be introduced in Sect. 4.4.7, and the maintenance of systems in Luhmann's theory. For him, social systems are autopoietic as opposed to being controlled via coupled systems, meaning that they are coherent through self-observation. Self-observation is nothing but a re-entry, it is «the operation of distinguishing system and environment *within the system*.»<sup>69</sup> The unmarked space along with its possibility to be distinguished is called medium by Luhmann, conceived as a loose coupling of elements. The medium that keeps a system together is “meaning”, the paradoxical simultaneity of presentation (actuality) and potentiality: meaning is the same from the inside and the outside, before the re-entry “is” and “is not” after the re-entry. Meaning both distinguishes between what is actual and what else could be and performs the actualisation (indicates the potential) itself. Operational closure allows meaning, because actual operations can be potential operations—they can appear on the horizon from which meaning in-forms and is kept in motion.<sup>70</sup>

### 3.3.5 Connectivity

It is important to note that Luhmann sees this “meaning” in opposition to hermeneutics, although apparently it shares the property of being unstable and needing continual actualisation. The crucial difference is that hermeneutics retains the idea of a «boundary between the external and the internal», that an interpretation needs to «penetrate the surface of an object (that is, a text) or a subject (in other words, a mind)» to reveal a truth.<sup>71</sup> With the radical approach of

---

<sup>68</sup>Ludwig von Bertalanffy (1950), ‘An outline of General System Theory’, *The British Journal for the Philosophy of Science* 1(2), pp. 134–165; also von Bertalanffy, ‘The History and Status of General Systems Theory’

<sup>69</sup>Luhmann, ‘Deconstruction as Second-Order Observing’, § II.

<sup>70</sup>Niklas Luhmann (1995), ‘The Paradox of Observing Systems’, *Cultural Critique* 31, pp. 37–55.

<sup>71</sup>Luhmann, ‘Deconstruction as Second-Order Observing’, § I.

autopoiesis, there is no longer any outside other than a distinguished outside on the inside of the form. The boundary of the “system” is an instable ignorance function of the system itself.<sup>72</sup> The system needs to oscillate between the two sides of the distinction in order to preserve «the undecidability of whether something is inside or outside a form»,<sup>73</sup> because otherwise there would not be any *real* decisions, based on “free choice”, but everything would just have to be “calculated”.<sup>74</sup>

So operational closure, the property that a system’s output may again be subject to its transformations, is a double-edged sword. It allows representation of something basically unrepresentable (‘becoming’), thereby constructing a false “identity” of process. But it is at the same time the condition that allows process to be kept in motion. Operational closure in the form of re-entering “meaning” as boundary establishment resolves the apparent contradiction between closure and openness of the process. It indeed accommodates Emerson’s “new actions” entering the process (Fig. 3.1a), *because* the system—of art production in general—is opaque and indeterminate. And finally, operational closure reappears on the level of individual compositional processes; here it stimulates new permutations of the material traces of the process, as the transformations are “uninterested” in the material and may be applied to arbitrary elements, which I have discussed as ‘random access’.<sup>75</sup> Another way to address the general system’s opacity is to describe the motion from individual process to general process, the establishment of “new actions”, as a form of emergence or ‘conjunction’ (Rheinberger).

### 3.4 Injection

Because the argument has been spun as a long thread, a moment should be taken to recapitulate and to decide which aspects of it should be injected into the design of the observing software

<sup>72</sup>This radical system view, which abandons the idea of an outside, will return in Sect. 4.3 as a critique by J.-F. Lyotard who focuses particularly on that which is ignored or not-indicated, yet “among us”.

<sup>73</sup>Luhmann, ‘The Paradox of Observing Systems’, § III.

<sup>74</sup>This paradox is developed as a “metaphysical postulate” in Heinz von Foerster (1991/2003), ‘Ethics and Second-Order Cybernetics’, in: *Understanding Understanding: Essays on Cybernetics and Cognition*, New York: Springer, pp. 5–8; cf. Niels Åkerstrøm Andersen (2003), ‘The Undecidability of Decision’, in: *Autopoietic Organization Theory: Drawing on Niklas Luhmann’s Social System Perspective*, ed. by Tore Bakken and Tor Hernes, Oslo: Abstrakt Forlag, pp. 235–258

<sup>75</sup>Hannes Holger Rutz, Eduardo Miranda and Gerhard Eckel (2011), ‘Reproducibility and Random Access in Sound Synthesis’, in: *Proceedings of the 37th International Computer Music Conference*, Huddersfield, pp. 515–522.

system. The chapter started by reviewing models of the compositional process and thereby methodologies to tackle it, and showed how they are based on the assumption of observable goal-directed behaviour. And the assumption goes across disciplines. *I* did not decide to talk about Marvin Minsky, but David Collins did. And Eaglestone likewise chose to talk about Collins. Did Emerson pick Miller, Galanter and Pribram? Not explicitly, but the construction of the model is too perfect a match for the TOTE, the «crux of systemic monism»,<sup>76</sup> even when adjusted by the possibility of “new actions” to counter its conservative streak.

In summary, three main problems can be named: (1) The establishment of causes and effects, (2) hierarchical nesting as the chosen means of de-paradoxification, and (3) the dismissal of the material reproduction of a system’s boundaries. These interact with each other, but for the sake of clarity will be recalled separately.

### 3.4.1 Arrows

Often arrows are falsely turned around. Closure can appear as stability, but neither does stability imply closure nor must a system under closure remain stable.<sup>77</sup> Likewise, goal-directedness may result in stable behaviour, but stability may not be caused by goal-setting, nor does perceived instability imply that a system might not behave stably with respect to another, unobserved agenda.

Furthermore, it is not only the direction of arrows that is problematic, but the fact that they embody what Heylighen calls an epistemology of *correspondence*, where every «...conceptual object (symbol) in the knowing subject’s model is supposed to correspond to one or more physical objects in the environment.»<sup>78</sup> This is presented as the “mainstream” view of Artificial Intelligence, and we have shown how the terminology of image, reflection, and transmission of patterns belongs to this theory of knowing. The symbolon as metaphor had been connected by

---

<sup>76</sup>Pask, ‘The meaning of cybernetics in the behavioural sciences (The cybernetics of behaviour and cognition; extending the meaning of ‘goal’).

<sup>77</sup>Operational closure (Luhmann) and organisational closure (Varela) are problematic categories in themselves, for they are ultimately either axiom or observed constancy, and furthermore do not specify all the other terms in which a system is *not* closed. Stability as a particular rendering of closure is stated in Francisco J. Varela (1981), ‘Autonomy and Autopoiesis’, in: *Self-organizing Systems: An Interdisciplinary Approach*, ed. by Gerhard Roth and Helmut Schwegler, Frankfurt and New York: Campus Verlag, pp. 14–23

<sup>78</sup>Francis Heylighen (2001), ‘Bootstrapping knowledge representations: From entailment meshes via semantic nets to learning webs’, *Kybernetes* 30(5/6), pp. 691–722.

Krippendorff to the idea of cognitive sharing, which in turn assumes a transmission metaphor of communication, where a «...*code* describes the process of “translation” by establishing a correspondence between the motions, changes, or choices made in one medium and motions, changes, or choices subsequently occurring in another», and which is built on an understanding of ‘information’ in the mathematical terms of Shannon such that it «measures the extent to which coding processes are reversible and thus preserve a pattern.»<sup>79</sup> Information as a third quantifiable substance besides matter and energy has the paradoxical property of being transmitted and at the same time remaining with the sender, producing this apparent symmetry (co-respondence) which is only the sheep’s clothing covering an underlying asymmetry (authority of the sender). The superficial symmetry was also encountered in the pair goal-setting/goal-attainment. As a couple, they function outside-time, they *constitute* (sym)metric time. The arrow seems to spin around, but actually it is just a diagrammatic trick (how to draw the two sides), like walking around a compass, and thereby seeing the needle rotate, whereas the process or landscape is hidden under the surface of the compass. The symmetry layered on top of an asymmetry is extensively studied by G. Deleuze as two forms of repetition. The superficial or “bare” repetition lies in the “effect”; it is the symmetrical repetition of the Same, the identical reflection as «difference between objects represented by the same concept». In contrast, the “deep” repetition lies in the “cause”, it is asymmetry and origin.<sup>80</sup>

The constructivist critics of the structuralist correspondence-as-truth propose instead to base an epistemology on *coherence* (Heylighen<sup>81</sup>) or *consistency* (Varela<sup>82</sup>). However, as Heylighen points out, it is quite difficult to exactly define coherence. The constructivists rely on another metaphor, that of keyhole and key. Those items are processed or “selectively retained” which “fit”. Another opposition to correspondence would thus be complementarity (key and hole

<sup>79</sup>Klaus Krippendorff (1993), ‘Major Metaphors of Communication and Some Constructivist Reflections on their Use’, *Cybernetics & Human Knowing* 2(1), pp. 3–25.

<sup>80</sup>Gilles Deleuze (1968/1994), *Difference and Repetition*, trans. by Paul Patton, New York: Columbia University Press, p. 23f.

<sup>81</sup>Heylighen, ‘Bootstrapping knowledge representations: From entailment meshes via semantic nets to learning webs’.

<sup>82</sup>He writes that in the observation of an autonomous system, «what we could call a representation is not a correspondence given an external state of affairs, but rather a consistency with its own ongoing maintenance of identity.» Varela, ‘Autonomy and Autopoiesis’

complement each other).<sup>83</sup> If second-order approaches can be described as a change from the question as to what an observer is (ontological status) to who the observed and the observing are (functional differentiation),<sup>84</sup> we will take another step backward and focus on the materiality of the written/observed. This is not a simplification, but a protection from the pitfalls of language. In fact we cannot come closer to understanding process than by beginning with the relations between graphemes, its material traces.<sup>85</sup>

Cybernetics has been both obstructive and useful for approaching the nature of process. On the obstructive side, there is a distraction from the trace due to the interest in «tracing long chains of cause and effect, so that we can relate a set of possible initial causes to a set of final machines issuing as consequence»; on the useful side we find the concept of difference and differentiation, and the functional composition of systems which are re-definable, so that for example the question of “initial states” (origin) in a system can be dissolved (de-paradoxified) by factoring them out as noise in the environment.<sup>86</sup>

### 3.4.2 Levels

Spencer-Brown has shown that the two basic modes of geometric arrangement are sequence and “order” (nesting, hierarchy). These are also the two possibilities of combining TOTE units. Each nesting level is the transgression of a cross, one additional act of distinguishing/observing. Accordingly, the classic scientific observation is situating the observer on a meta-level with respect to the level of the observed object. This dissolves the observer paradox as long as «... one does not address the unity of this distinction».<sup>87</sup>

This view can still be found even in areas which acknowledge the artificiality of system borders, such as soft systems methodology (SSM), a branch of systems theory formulated by P. Checkland which questions classical problem-solving strategies because it deals with problems that cannot

---

<sup>83</sup> cf. Krippendorff, ‘Der Verschwundene Bote; Metaphern und Modelle der Kommunikation’, § 6.10

<sup>84</sup> Luhmann, ‘Deconstruction as Second-Order Observing’, § III.

<sup>85</sup> See Rheinberger’s idea of an epistemic semiosis which is a motion between the material traces of experimentation in the first place, and before it may become a linguistic process: Hans-Jörg Rheinberger (2nd July 2008), ‘Epistemische Dinge—Technische Dinge’, *Bochumer Kolloquium Medienwissenschaft*, URL: <http://vimeo.com/2351486> (visited on 28/08/2012), 48’

<sup>86</sup> cf. Ashby, *An introduction to Cybernetics*, § 3/11, § 11/19

<sup>87</sup> Luhmann, ‘The Paradox of Form’, § VI.

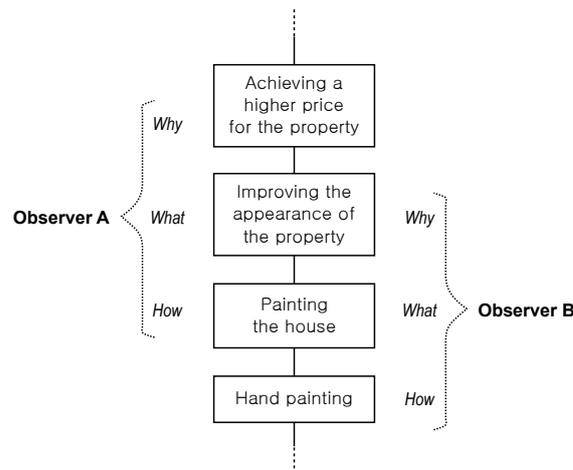


Figure 3.6: Observer-defined levels of thinking (Checkland)

be formulated in terms of goals and purposes. Its foundations are emergence and hierarchy taken from general systems theory and control and communication taken from cybernetics.<sup>88</sup> It tries to address “soft” problems as found in management of organisations.<sup>89</sup> Although perspective is observer-dependent, observers are situated in a common space where they observe different layers or levels of nested systems. This is illustrated in Fig. 3.6 which was adopted from Checkland.<sup>90</sup>

We have chosen this diagram because it shows at least two problems both in the current research into the compositional process and the possible design of such an observation. First, the alignment of the observers based on shared “objectives”, with the implicit arrow being a reading of motivation or motives—observer *A* can be interpreted as observing observer *B* for a higher purpose. If we think of the studies of Eaglestone or Collins, the ‘why’ is something like “understanding creativity”. The ‘what’ is the “sampled subject” coupled to his objective, that is *B* being for instance a composer whose goal is to create a new piece of music (and not to understand his own creativity, at least not explicitly or assumed in this model). His ‘what’ are then the actions in Emmerson’s model (Fig. 3.1a) or the development of “solutions” in Collins’s model (Fig. 3.4), again objectively manifest in order to create the axis with *A*’s ‘how’.

<sup>88</sup>P. Checkland, cited in Alex Ryan (2008), ‘What is a Systems Approach?’, *Arxiv preprint arXiv:0809.1698*

<sup>89</sup>For an overview see Peter Checkland (2000), ‘Soft Systems Methodology: A Thirty Year Retrospective’, *Systems Research and Behavioral Science* 17, S11–S58

<sup>90</sup>Ibid.

Second, the open axes. Again there are two of them: *A*'s 'why' has no connection to *B*, and *B*'s 'how' is unobserved by *A*. The second case is a problem of generalisation in (and possibly invalidation of) the research, the first case is a problem of relevance (or feedback) of the research. The generalisation problem can be equated either with the problem of representing process (*B*'s 'how' is lost), but also—if *A* assumes the role of software designer in Fig. 3.5—with a problem of prescribing usage of a system (*A*'s 'how' determines 'what' *B* can do, so *B*'s 'how' is irrelevant or redundant).

For example, a common view in computer-aided composition (CAC) is to see the compositional process going top-down from conceptualisation to formalisation to eventual sound realisation (cf. Sect. 2.1 and Sect. 2.2). The following quotation from a framework for *OpenMusic* by J. Bresson and C. Agon is a clear instantiation of the researchers and designers speaking about the imagined composer, standing one step above him on the staircase:

«First we must consider sound as the *intentional* object of a compositional process, and therefore as a structure likely to be represented by a program. Through the general structuring of the programs via the *abstractions* or *hierarchical constructs*, compositional *models* are created, made of organized [components] and structural aspects and *corresponding* to particular situations or compositional approaches. In this context, sound is therefore not considered only from the acoustic signal production point of view, but rather as the *product* of a compositional process that aims to create this signal ... The implementation of these models ... might therefore allow for the symbolic access and *control* of the corresponding sound representations, whether they concern symbolic or *subsymbolic* structures. The synthesis processes are then integrated within a network of structural relations and components, uniting the *low-level* sound synthesis aspects and the symbolic layers of composition, and providing ... a representation allowing one to formulate compositional intentions. The *musical* sound is made explicit by *recovering* (mentally and/or physically)

the object of the model as a formal structure . . . or eventually as a sound signal, according to a synthesis process.» [Emphasis added]<sup>91</sup>

While abstraction is a necessary part of using programming languages, and we welcome the employment of programs, the more problematic hierarchy here is the distinction between sound and musical sound, where the “plain” sound is referred to as “low-level” or “subsymbolic”, only contributing to the understanding of the music as a reference whose signified needs recovering, and the signified being the original intentions of the composer. Consequently, “sound synthesis processes” are considered an «abstract variable element» in their compositional models<sup>92</sup> and delegated to external programs, thus not observed by the system and given the status of an uninteresting ‘how’ (craftsmanship). The authors argue that thereby their system gains openness, and indeed findings by B. Eaglestone et al. support this in the sense that composers’ creativity is boosted when they are faced with switching between applications.<sup>93</sup> However, the approach is devalued by the additional statement by Bresson and Agon that it allows a stronger concentration on “compositional issues” as well as by the fact that connectivity is only considered in one direction and misses the complementary motion from sound to structure.

We shall give only two short antitheses here. The first is the perspective of Di Scipio, for whom «the array of DSP algorithms, and the methods by which they communicate among themselves, should be seen as the material implementation of a compositional process or concept».<sup>94</sup> The second comes from the just mentioned study of Eaglestone et al. They find that users work with «software tools at all levels of abstraction and which stimulate and challenge, rather than reflect, their perceptions», suggesting that a software should implement a combination of “micro and macro views” and an interface allowing the user to access data across all levels of abstraction.<sup>95</sup>

<sup>91</sup>Jean Bresson and Carlos Agon (2007), ‘Musical Representation of Sound in Computer-Aided Composition: A Visual Programming Framework’, *Journal of New Music Research* 36(4), pp. 251–266, § 4.1.

<sup>92</sup>Ibid., § 5.1.

<sup>93</sup>Barry Eaglestone et al. (2007), ‘Information systems and creativity: an empirical study’, *Journal of Documentation* 63(4), pp. 443–464, § 5.

<sup>94</sup>Agostino Di Scipio (2003), ‘“Sound is the interface”: from interactive to ecosystemic signal processing’, *Organised Sound* 8(3), pp. 269–277.

<sup>95</sup>Eaglestone et al., ‘Information systems and creativity: an empirical study’, § 5, §6.

Even if one does not impose a specific hierarchy to the process of composition, the problem of the axes correspondence in the levels view is not resolved. The problem lies in separating form and content. With the levels, we believe we can observe without interference; the researcher's 'what' (study object) does not influence the composer's 'why' (motivation), the researcher's 'how' (methodology) does not influence the composer's 'what' (working object). To guarantee such transparency on the part of the observer, researchers advocate "naturalistic settings". For example, Collins prefers video-recorded observation and collection of computer artefacts to «enable the researcher to withdraw from the participant's 'space' allowing a naturalistic, constraint-free setting in which creative work could take place».<sup>96</sup> Likewise, Eaglestone et al. felt obliged to ensure that the «situation was "pretty natural"» and the composers «didn't feel they were taking part in a scientific experiment».<sup>97</sup>

On the encouraging side of the so-called 'naturalistic inquiry', its proponents E. G. Guba and Y. S. Lincoln do criticise the positivist assumptions such as objectivity, cause and effect, explicable hypotheses, and so forth.<sup>98</sup> They propose the interesting concept of the inquirer as "smart instrument", who operates by «by virtue of his sensitivity, responsiveness, and adaptability». Yet, Luhmann's critique of hermeneutics still applies here as the boundary between internal and external is asserted, i.e. the clear separation of the left and the right side of Fig. 3.6. The inquirer is *instrumental* and never becomes a *conspirator*.

Our solution to the conflict between the experimental approach—which assumes that a situation is clearly defined and controlled—and the "naturalistic" approach—which asks for unhindered movement of the observed subject—is to redefine what constitutes experimentality and to assign the composer the additional role of researcher. This assignment also addresses the problem of relevance. The result is a situation of self-observation, perfectly possible in Checkland's model which indeed states that a single observer should move to different levels, and in line

---

<sup>96</sup>Collins, 'A synthesis process model of creative thinking in music composition'.

<sup>97</sup>Eaglestone et al., 'Composition Systems Requirements for Creativity: What Research Methodology'.

<sup>98</sup>Egon G. Guba and Yvonna S. Lincoln (1982/2002), 'Epistemological and methodological bases of naturalistic inquiry', in: *Evaluation Models: Viewpoints on Educational and Human Services Evaluation*, ed. by Daniel L. Stufflebeam, George F. Madaus and T. Kellaghan, Second Edition, New York: Kluwer Academic Publishers, pp. 363–381.

with von Foerster saying that each observation is ultimately self-referential.<sup>99</sup> It also follows from Eaglestone et al.'s assumption «that one must create [a] constructivist, experimental and individually tailored research situation, in order to investigate a process which is highly experimental, uncontrollable and personal.»<sup>100</sup>

What the composer-as-researcher is given is a computer music framework, developed as part of this thesis, which is his experimental system and serves as a secondary observer. This constellation can be seen in the topmost part of Fig. 3.7, which was taken from a previous paper.<sup>101</sup> The framework provides abstractions for the representation of sound objects along with a tracing mechanism for the memorisation of the coming-into-being and evolution of such objects. An interface is provided by which observers can be attached to this framework and the production of sound objects. This can be either an algorithmic coupling or a human switching between the roles of Fig. 3.5.

As an *actual* software this framework is necessarily one of control and regulation, as this is the foundation of computer science and all the bits and pieces from which the framework is constructed. Also inevitably the observation of “process” through control and regulation is limited by the observation horizon. For example, the framework is implemented in a general programming language, providing an embedded domain specific subsystem responsible for the representation and registration of sound objects. The technical boundary between this internal domain and the general domain of the hosting programming language is permeable, making it easy for someone programming this framework to include code or actions which escape the internal horizon.

One strategy could be to extend the tracing system to include the hull that was formerly its environment: In the second step of Fig. 3.7, the hypothetical observation would extend over the entire programming language, perhaps by tracing the evolution of a program's abstract syntax tree. The composer easily transgresses this boundary, too, as he incorporates artefacts

---

<sup>99</sup>In a self-referential situation, «... one is encouraged to speak about oneself. What else can one do anyway?» von Foerster, 'Ethics and Second-Order Cybernetics'

<sup>100</sup>Eaglestone et al., 'Composition Systems Requirements for Creativity: What Research Methodology?'

<sup>101</sup>Hanns Holger Rutz (2011), 'Limits of Control', in: *Proceedings of the 8th Sound and Music Computing Conference (SMC)*, Padova, 132:1–132:6.

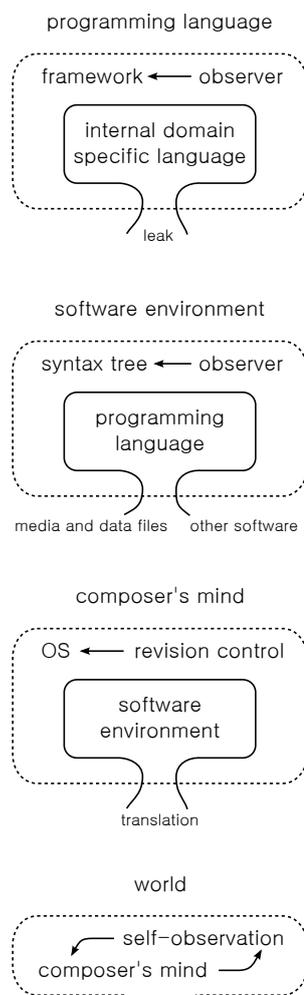


Figure 3.7: Computer based tracing of the compositional process, and the limits of control

produced with other software systems. The observation could be extended by using a general artefact tracking system, such as a file-based revision control system. But then, a major part of the compositional process takes place outside of the computer, within the composer's mind, in his sketchbook, etc. Ultimately, the only possible tracing of this horizon is through the self-observation of the composer.<sup>102</sup>

The other strategy would be to try to close the leaks, i.e. to restrict the composer in the repertoire of actions. While this may work in the first two layers, it does not eliminate the leak where things are translated between computer and human (third layer). Furthermore, it violates the requirement formulated by Eaglestone et al. to allow elements of non-control and access across all levels of abstraction. Essentially, we would create pure technology which cannot be destabilised to become epistemic. The failure of control is necessary for the emergence of "new actions" in the corner of one's eye (cf. Sect. 4.3ff).

### 3.4.3 Double Bind

Throughout this chapter we have tried to show how the methodological background layer that is present in all of the prior research reviewed has created what could be called a double bind situation, to borrow the term from cyberneticist G. Bateson. It was originally used in a research project on psychotherapy where it signified a situation which Bateson assumed to be a cause of schizophrenia.<sup>103</sup> The patient trapped in a double bind is exposed to a recurrent experience in which he can only lose, no matter which behaviour he chooses. The double bind is created by two concurrent instructions in conflict with each other, for example by the parents disagreeing or by a contradiction between an action such as punishment and a verbal utterance such as "do not see this as a punishment". An additional force prevents the victim from escaping this situation, and eventually he becomes habituated to the ongoing conflict.

I want to use the term double bind here more generally in order to describe the problem that each of the approaches is exposed to such a conflict between two opposites, but because the concepts

<sup>102</sup>This idea of the hierarchical nesting of languages and meta-languages up to the «ultimate human observer» can be found for example in Lars Löfgren (1992), 'Complementarity in language; toward a general understanding', in: *Nature, Cognition and System II*, ed. by Marc E. Carvallo, Dordrecht: Kluwer, pp. 113–153, §1/3

<sup>103</sup>Gregory Bateson et al. (1956), 'Toward a Theory of Schizophrenia', *Behavioral Science* 1(4), pp. 251–264.

underlying these approaches have been so persistent over decades and still form the mainstream methodology in many disciplines, one has “learned” to accept and somehow ignore this conflict. Any attempt to question the premises is fiercely rejected. To give an example, in a peer-reviewed academic psychology journal, psychologist D. J. Kruger accuses a colleague of his who dared to propose the inclusion of “postmodern” elements in his discipline of promoting «speculation for the sake of intellectual discourse rather than the pursuit of truth or knowledge.»<sup>104</sup> How he managed to pass the peer review despite including a comparison with the Dada movement, I do not know.

In the area of musical research, there is a new trend to establish “cognitive science” as the opinion leader. A quite aggressive partisan of this movement is M. Pearce. Besides selling the paradigms established in the 1950s and 1960s as «suggestions for future research»,<sup>105</sup> he devotes quite some energy on dismissing other disciplines as invalid for research on music composition. Oddly disguised as an “artist’s statement”<sup>106</sup>—oddly, because in a list of eleven people contributing their statements, he is the only one who does not have artistic work—he claims that there are different parties who have different “goals” and «different methodological requirements for demonstrably achieving those goals»; of these parties, those working on algorithmic composition are devalued as not having «rigorous criteria for success»; those who design compositional tools can be divided into worthless engineers and those who base their tools on a music theory so that their “success” can be measured by comparing computer-composed pieces with «existing music»; within those modelling existing musical styles, a personal assault is launched against a researcher who dares to also be a composer; and finally—it is always good to be last in a list—those who computationally model music cognition (no critique found).

There are alternative pairs of end points between which what we are seeking to trace—the process—is lost: control/communication is such a double bind. The antagonism is summarised in Fig. 3.3 and was discussed with respect to the law of requisite variety; minimising versus maximising the variety of possible expressions of the environment and the system respectively.

---

<sup>104</sup>Daniel J. Kruger (2002), ‘The Deconstruction of Constructivism’, *American Psychologist* 57(6–7), pp. 456–457.

<sup>105</sup>Marcus Pearce, David Meredith and Geraint Wiggins (2002), ‘Motivations and Methodologies for Automation of the Compositional Process’, *Musicae Scientiae* 6(2), pp. 119–147.

<sup>106</sup>Marcus Pearce (2009), ‘To Beep or Not to Beep’, *Contemporary Music Review* 28(1), pp. 125–126.

Another pair is goal-setting/goal-attainment which we have said constitute an outside-time operation in the sense that its representation is foldable. A third pair would be music/mind in the sense that the body and material trace in between are made superfluous. Finally, the pair conceptualisation/perception with “description” in its middle: When a description is produced, it is «... always finitely representable and locally independent of time, whereas *what* the descriptions describe, the interpretations (models, computer behaviours, phenotypes), may be infinite of any order as well as dynamic»<sup>107</sup> (my emphasis). The “fragmentation” or partiality in the description must appear as a failure to contain the “whole” language, yet we still do not see an escape other than repeating the desperate attempt to reconstruct and assimilate the communication ends, to complete the symbolon. We thus unfold and unfold the finite representation, but we do not look at the fragmentation itself.

### 3.5 Resolution

To resolve the double bind we have to employ a methodology which frees itself from the habitual expectations of “demonstrably achievable goals”. We *do* demonstrate and achieve, but by using a re-entry of the form of the thesis’ object—tracing process. We use writing *processes* to observe writing processes. The form of the writing, the way it grows across the different chapters, is as essential as the apparent end point or conclusions of a “representation of process”. Chap. 4 connects to artistic writing processes—a selection of sound works—by analysing them but at the same time translating them and revealing through the translation the essential movements and shifts which characterise process. A number of scholars have also encountered these difficult-to-observe movements and we show how by gathering them we gain a more acute contour of our observed.

We also introduce into the artistic writing process a software framework, a piece of technology, but the text demonstrating the design of this framework, Chap. 5, is itself again a crucial element which is interlocked with the material manifestation of that framework. It oscillates between a computer science discourse and (subtly) the poetic texture of algorithmicity as such. It is not just

---

<sup>107</sup>Löfgren, ‘Complementarity in language; toward a general understanding’, §1; the conceptualisation or translation of the *what* into the finite set of symbols eventually perceived and interpreted, will return as vertical axis composing/analysing in Fig. 4.35.

a product-description, but in its entirety the starting point for further experimentation. Ultimately, all chapters become a hypertext in which the motion of process can be retraced, the very same motion that is subject to the analysis in Chap. 4. In order to observe the motion, delimitations are drawn between philosophical, methodological, artistic and technological discourse—without them, we would perhaps end up like Pedro Camacho, the scriptwriter in Mario Vargas Llosa's *La tía Julia y el escribidor*, who begins increasingly to mix up the different characters and facts that had previously been kept neatly separated.

The project was an *involved* approximation. I am the writing machine, and as practitioner-researcher<sup>108</sup> I am subjecting myself to experimentation which tries to carve out what the role of tracing in composition is, and to what extent the traces can be observed. Self-observation: «... What else can one do anyway?»<sup>109</sup> Eaglestone et al. felt the need to report that the composers they observed, «... didn't feel they were taking part in a scientific experiment».<sup>110</sup> But being aware of this was my only guarantee of success, which is to produce connectability both with the scientific research on computer music and the praxis of computer music. This double relation is what should define artistic research.<sup>111</sup>

To understand process, we can think of it for a moment as the interface between a system, a form or actuality—that which has been marked or distinguished—and an environment, a medium or virtuality. This exchange has been described by Varela as a *dance* between autonomy (law-from-the-inside) and control (law-from-the-outside).<sup>112</sup> But then I believe what defines this dance ultimately is not the dancing partners, but something integral to the form of dancing itself. I have illustrated this idea in Fig. 3.8, combining the metaphor of the bifurcation occurring in a dynamic generator with J. Derrida's idea of an irreducible dynamic of writing.<sup>113</sup>

---

<sup>108</sup>Cf. Michael Biggs and Daniela Büchler (2011), 'Communities, Values, Conventions and Actions', in: *The Routledge Companion to Research in the Arts*, ed. by Michael Biggs and Henrik Karlsson, Abingdon and New York: Routledge, pp. 82–98

<sup>109</sup>Von Foerster, 'Ethics and Second-Order Cybernetics'.

<sup>110</sup>Eaglestone et al., 'Composition Systems Requirements for Creativity: What Research Methodology', § 6.2.

<sup>111</sup>For the difficulty of this term, see Henk Borgdorff (2011), 'The Production of Knowledge in Artistic Research', in: *The Routledge Companion to Research in the Arts*, ed. by Michael Biggs and Henrik Karlsson, Abingdon and New York: Routledge, pp. 44–63

<sup>112</sup>Varela, 'Autonomy and Autopoiesis'.

<sup>113</sup>Jacques Derrida (1972/1988), 'Signature Event Context', in: *Limited Inc*, ed. by Gerald Graff, trans. by Samuel Weber, Evanston, Illinois: Northwestern University Press, pp. 1–23.

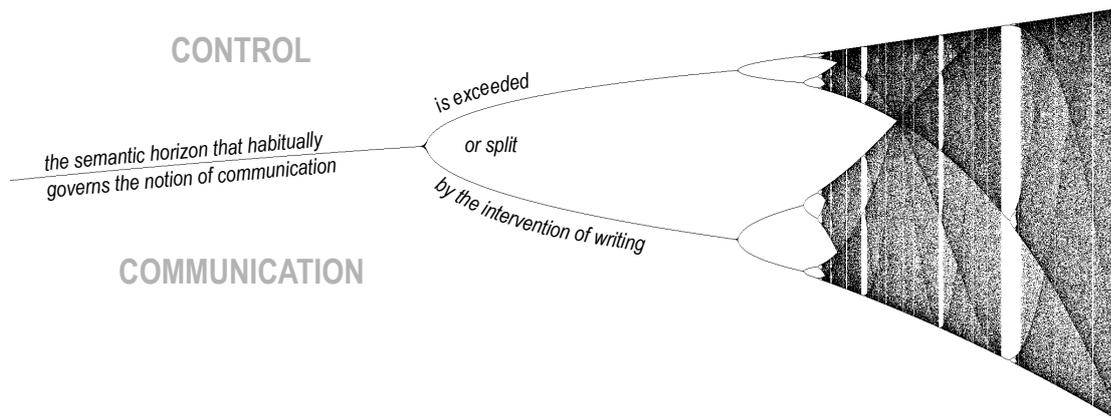


Figure 3.8: Bifurcation opening a space for the dance of material traces

To observe process, we need to observe the material traces of composing. The material reproduction must be embedded in our “system design”. This embedding will be the facilitator of experimentality in the sense that the epistemic thing, that which experimentation tries to approach, is articulated by the material traces.<sup>114</sup> Within a pure technology—based on a solid foundation of computer science—the system must integrate a critical interface for the generation of an epistemic surplus value; an interface that allows the material trace of the compositional process to be subject to itself in a sort of “structural feedback”. To access the «remnants of older narratives as well as fragments of narratives that have not yet been told» and to allow for «unprecedented concatenation of the possible(s).»<sup>115</sup>

We also believe that the analysis of compositions should be primarily based on the descriptions, translations and rewritings of these traces. We need to bring into computer music a thinking which still seems rather foreign when looking at the common platforms of dissemination in this discipline, such as journals and conferences, whereas it is accepted in the general arts discourse and in philosophy. To quote from the latter field, I want to conclude this chapter with G. Böhme, mostly noted for his concept of atmospheres as the foundation of aesthetics. A critic of the dominance of language, communication and semiotics in the aesthetics, he reminds us:

<sup>114</sup>Cf. Hans-Jörg Rheinberger (1994/2005), ‘Alles, was überhaupt zu einer Inskription führen kann’, in: *Iterationen*, Berlin: Merve Verlag, pp. 9–29

<sup>115</sup>Hans-Jörg Rheinberger (1994), ‘Experimental Systems: Historiality, Narration, and Deconstruction’, *Science in Context* 7(1), pp. 65–81.

«It is not, however, self-evident that an artist intends to communicate something to a possible recipient or observer. Neither is it self-evident that a work of art is a sign, insofar as a sign always refers to something other than itself, that is, its meaning.»<sup>116</sup>

---

<sup>116</sup>Gernot Böhme (1993), 'Atmosphere as the fundamental concept of a new aesthetics', *Thesis Eleven* **36**, pp. 113–126.



## Chapter 4

### Traces

The purpose of this chapter is to understand my compositional process by staking out a field of various realised works. Three different angles are used to plough through this field. First, starting with the sound installation *Dissemination* as an example, we establish threads which link various aspects of it to preceding and succeeding works. One method is to name a concept, such as the ‘sound mobile’, and trace its materialisation beyond the boundary of a piece, showing how the way the configuration of the concept changes becomes the important indicator of the process rather than a constancy promised by using the same name. Second, this ‘differential reproduction’ can re-enter the inner discourse of the pieces themselves, something that is shown using the example of operationalising sound similarity. In this regard, we call sound similarity—or more broadly ‘signal processing’—an *interface*, something that mediates between human and computer in composition without submitting to the position of a utility that imposes the composer’s intention on the computer system. Instead, the interface is understood as a crucial suspension of the short circuit implied by ideologies such control and communication. It is the enabler of a proper material discourse which informs the compositional process in a substantial way and thus should be the centre of study. Finally, the third angle looks at pieces which have been composed with the software framework developed in the thesis, investigating how its use enables the composer to work with the trace of the process as well as providing a source for data-based analysis of pieces.

#### 4.1 Introduction

The interest in the works presented in this chapter lies in observing and retracing processes in which computer-based sound works are created and unfolded. When viewing these processes—

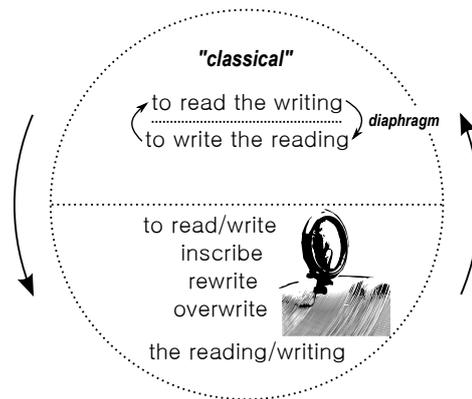


Figure 4.1: Phase model of relations between writing and reading

including their observation—under a systemic lens, one always finds nested layers of writing. In the literal sense, writing is the act of composing, beginning with the cliché of the blank paper which must be filled through *invention* and *intention*. This leads to a perspective in which art is seen as an excessive writing activity, antipole to scientific research which—again, following the cliché—begins with excessive reading, only eventually adding an arduously produced contribution.

In the discourse of artistic research, which seeks to combine both artistic and scientific practices, the relations between the two poles must be found. To begin with, the classical ideal of scientific research—still very much in effect—after which the observer should avoid influencing the observed, i.e. reading neatly isolated from writing, henceforth becomes a half-cycle in the phase diagram of the experimental oscillation (Fig. 4.1). It denotes a point of standstill, in which we look at our own traces, gathering a picture of how they can lead to something new in the next step.

The second half-cycle contains writing not as invention but as a reading/writing; inscription, rewriting, overwriting. This is where observer and observed cease resistance and relinquish their dissociation, they become one machine, the needle which at once scans the wax cylinder and incises it. This melded writing is driving and being driven; it is pure motion in the sense that what counts primarily is the act of writing and not what is being written.

What these pieces aim to do is resolve the conflict between exposing the processes and creating a produced “piece” and sensual surface. To work with algorithms so that the computer can carry on the writing, and also to insert a non-dramatic and non-intellectual layer which permits the listener or viewer to guard his or her proper time.

All pieces discussed in this chapter are represented on the accompanying DVD with sound examples and, if applicable, photographic material.

## 4.2 Dissemination

The first work is an audio-visual installation called *Dissemination* (2010), a collaboration between me and my partner, Nayarí Castillo. It was first conceived for an artist-in-residency programme in a rural area in Portugal, but only realised the following year in the context of the festival “Sounding Code” in Berlin, and shown again in a modified version in a gallery in Graz. In roughly the year that lay between the first sketching and the inauguration, part of the framework described in detail in Chap. 5 was developed and so this was a first testing ground for it. Some of the findings have been published previously.<sup>1</sup>

As will be seen, it is arguable where the working process begins and where it ends, but to make a first indication, connections can be drawn to two or three particular previous works by the artists. In 2009, Nayarí was working in a residency in Argentina. In *Natural Palimpsest*, located on a former forestry station, she created a parcours of twelve interventions, drawing from the memories of workers who had been employed at the station for more than twenty years. One of the interventions consisted of a tableau of seeds of different colours and shapes that were collected from around the station (Fig. 4.2). Seeds were chosen from those plants that did not originate from the place but were in some way “imported” to the region, as Nayarí’s work centres around ideas of travelling and migration.

On my part, there are two notable sound installations. In *Kalligraphie* (2007), shown in a gallery in Thuringia, glass plates are installed in a window situation, inviting the visitor to sit on the windowsill in their middle (Fig. 4.2). The headspace is expanded by the plates which are excited

---

<sup>1</sup>Hanns Holger Rutz (2011), ‘Limits of Control’, in: *Proceedings of the 8th Sound and Music Computing Conference (SMC)*, Padova, 132:1–132:6.

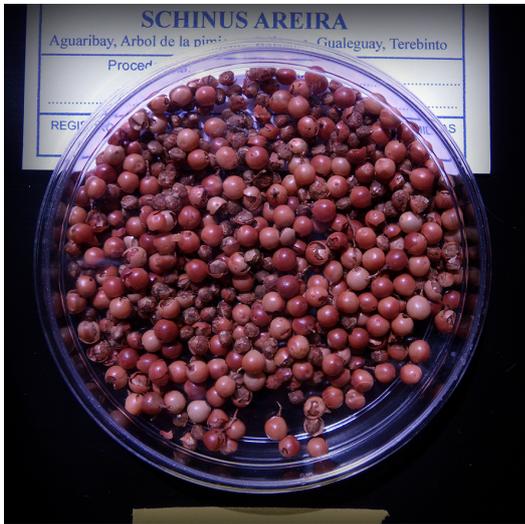


Figure 4.2: Left: *Natural Palimpsest*. Right: *Kalligraphie*

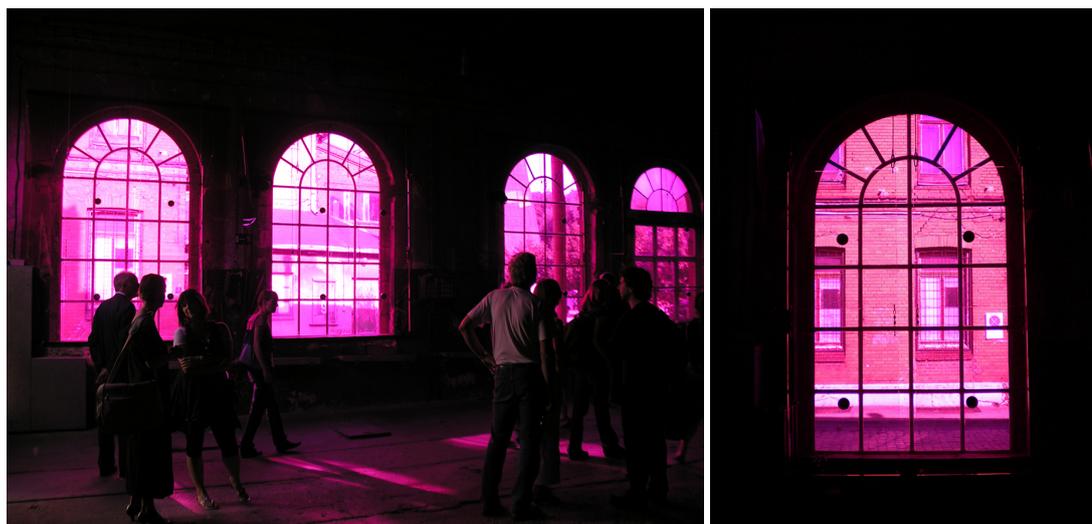


Figure 4.3: Sound installation *Amplifikation*, former tram depot Weimar

by small transducers, applied by the mere force of gravity. With their horizontal orientation, the plates evoke the image of specimen slides, dissecting the sound into vertical layers. The second installation, *Amplifikation* (2009), took place in the former tram depot of the electricity works in Weimar. Again glass plates were used in combination with transducers, but as the large space was dominated by old window panes, these plates were suspended vertically and in groups of four, essentially creating a duplicate image of the original windows, but shifted towards the inside of the hall (Fig. 4.3).

The diffusion of sound from the surface of glass plates creates a very particular effect. On the one hand, the characteristic resonances of the glass are superimposed on the sound, on the other hand the sound is emitted rather homogeneously from the whole surface, leaving behind the almost punctiform shape typical for speakers. These two phenomena provide the sound with a very physical or “material” presence.

*Dissemination* uses glass plates—each 90 by 70cm—in a combination of horizontal and vertical suspension. They function both as membranes of sound diffusion and as specimen holders for flying seeds, confined to a regular arrangement of petri dishes. Fig. 4.4 shows the model produced in the planning phase, Fig. 4.5 shows the actual installation in its two exhibition sites. The space is conditioned by filtering the daylight with yellow gels—a similar approach had

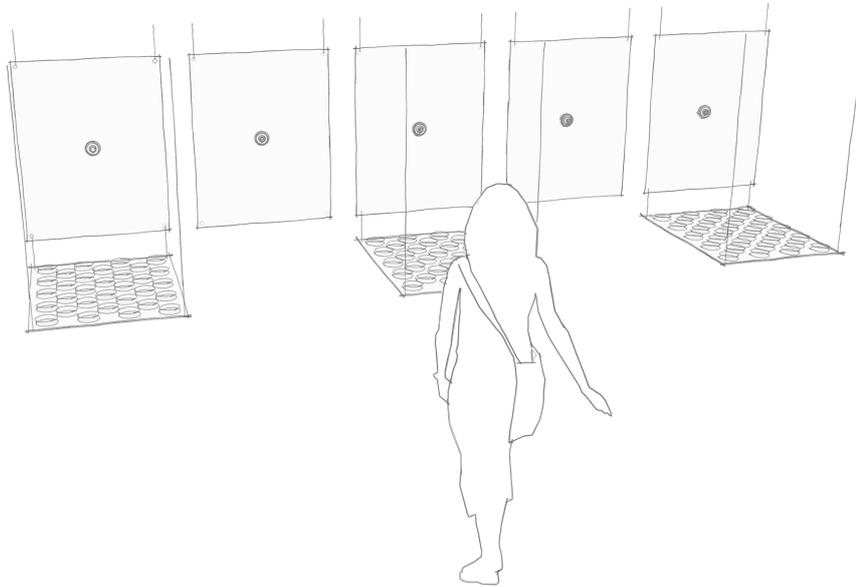


Figure 4.4: Model of the installation *Dissemination*. The vertical plates are excited by transducers, the horizontal plates carry the petri dishes with seeds.

been tried in *Amplifikation*, where pink gels produced a dreamlike but at the same time highly attentive effect in the sun-drenched depot.

Before deciding on the arrangement of the plates, I had imagined different possibilities, as the sketches in Fig. 4.6 show: Pure vertical or horizontal series, a long triptych format, a grid of squares suspended from a ceiling or arranged as an ‘X’ on the wall. The triptych format was used in a different work, and for *Dissemination* the decisive factor was to provide surfaces for the petri dishes and to be able to place the glass inside the space, so people could walk around them.

#### 4.2.1 Sound Mobile

Recognising the “origins” of the ideas that shape the sound composition is difficult, but one can always trace them back to previous steps. One such idea is the *sound mobile*, a sounding structure composed of different elements which are “suspended” in the space; on one hand a set of givens, on the other hand something in motion that would appear in ever so slightly different combinations, producing a stream of sound which would never repeat itself.



*Figure 4.5: Dissemination, as exhibited in Berlin (top) and Graz (bottom)*

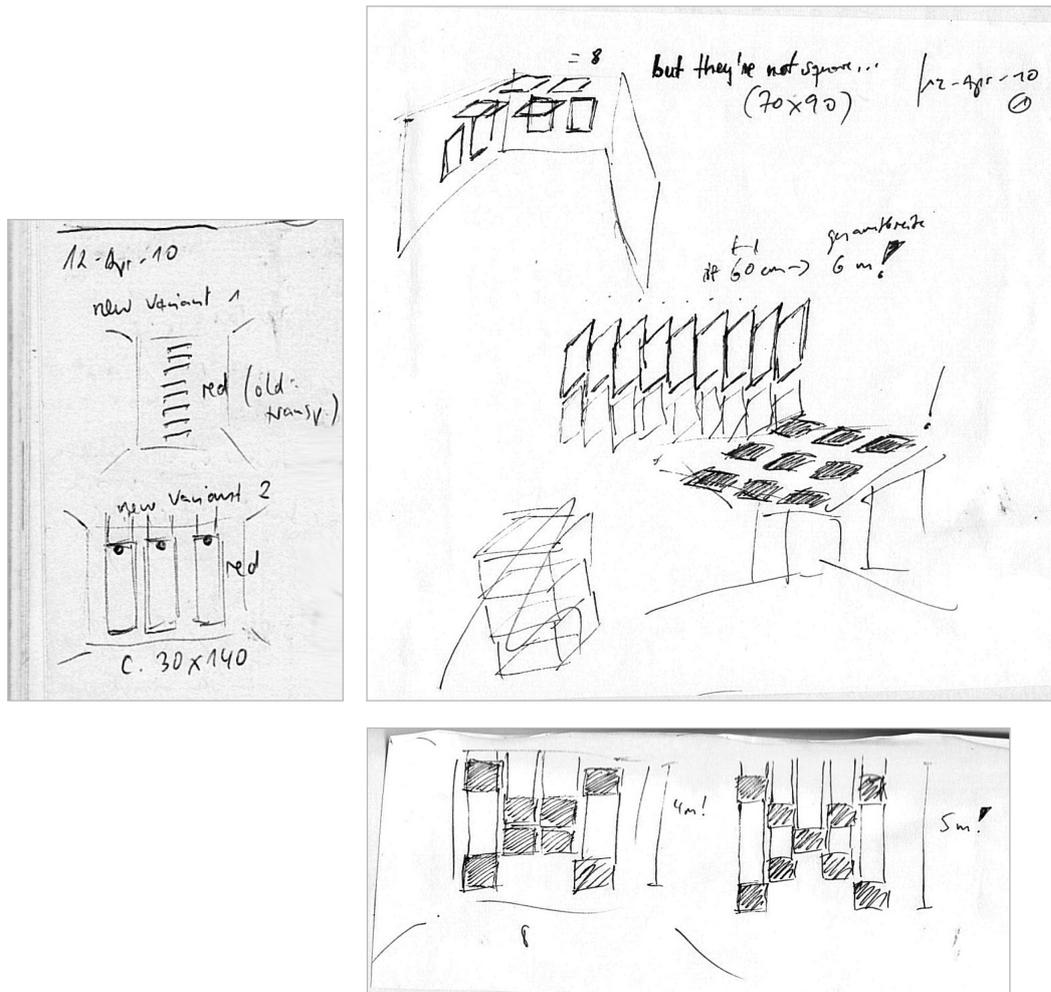


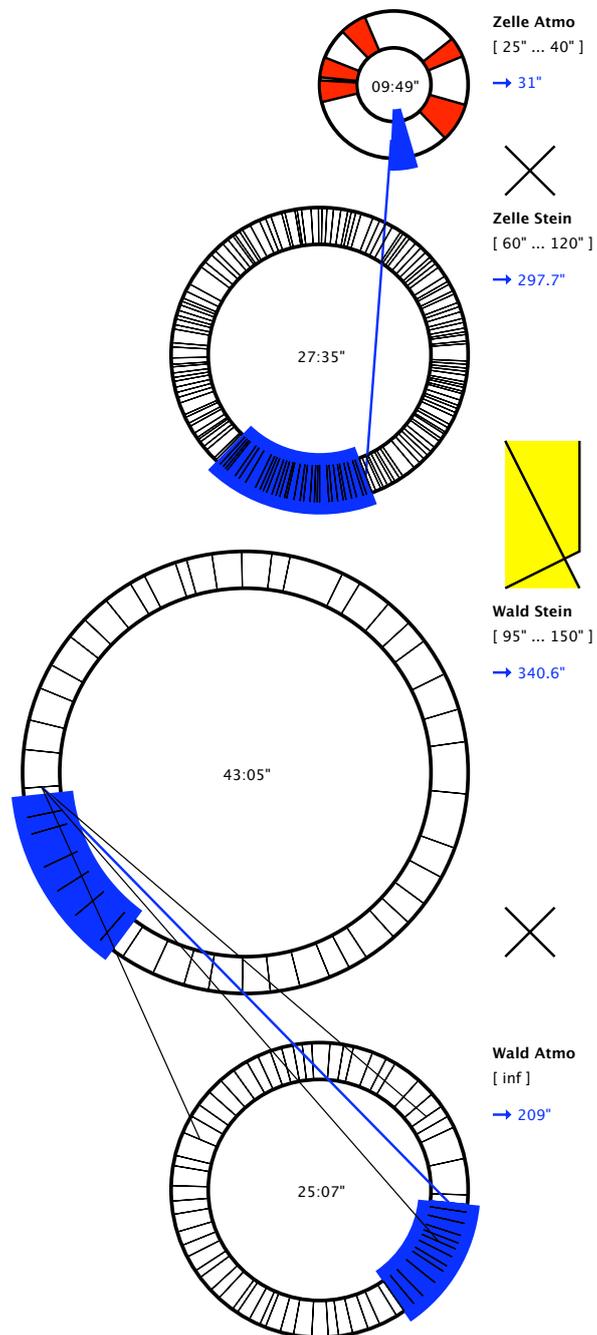
Figure 4.6: Sketching out variants for a new glass plate based installation. Unless specified, sizes are in cm.

With few exceptions, sounds are always produced based on electroacoustic—concrete—material, either atmospheres and ambient sound recordings picked up in “nature” or deliberately produced sound gestures, isolated as a “foreground” which is possible in a soundproof recording booth. To be able to mould these sounds, strategies must be found to cut them up, transform and recombine them in an “organic” manner that preserves the impact of the source sound, where the transformations and recombinations remain “credible” as if they might have occurred in the original context of these sounds—precisely not to stress the cut *as a cut in itself*.

In this respect, a sound installation from 2006 is noteworthy. *Zelle 148* was an intervention in cell no. 148 of a former Stasi prison in Erfurt, Germany (Fig. 4.7a). A metal seating surface was installed on top of the original plank bed, inviting the visitor to sit down and take headphones off the wall. A sensor would pick up the gesture of putting on the headphones to initiate the sound piece, which was synchronised with a subtle change in light atmosphere using MIDI controlled lights below the bed. The piece followed a rather linear development, going through four successive stages, as shown in Fig. 4.7b, an ex post score.

The simple form is  $A-A'-B'-B$ , where situation  $A$  takes place within the holding cell and  $B$  is located in a forest outside. The first and last section consist of the plain atmosphere of those two spaces without additional performative elements, whereas the two middle sections introduce the sound of a rock.

When the listener puts on the headphones, the atmosphere of the cell itself is faded in. The recording was done with a binaural technique at exactly the position of the seat, producing a rather strange effect which is hyperreal—the sound perception is alienated by the fact that an apparatus, the headphones, is worn, but at the same time everything sounds as if it was coming directly from the cell and its corridors. You hear the distant city outside the cell, shaped by the horrific acoustics of the confined space which bears some strong resonances. The beginning of the second section comes as a surprise. A large rock is heard sliding relentlessly across the cell floor, moving forward and backward between the short distance from the cell door to the tiny window. The actual rock *used to produce that sound* is left as a visible artefact of this action in one corner of the cell. Also visible are the scratch marks of the rock on the floor. The



(a) Installation cell (top) and prison corridor (bottom)

(b) Form scheme. The four stages proceed from top to bottom. Circle diameter corresponds to sound file length, spokes to markers in the file. Shaded sectors and lines connecting sections indicate one particular rendering of the piece.

Figure 4.7: Installation *Zelle 148*, former Stasi prison Erfurt

third section slowly fades into the outside space, using a sliding frequency filter as transition. The violent nature of the rock changes its character, as it is heard now sliding across dried leaves in the forest. Some wind and birdsong is heard, and the contrast of going from closed to open acoustics is particularly strong. In the final section the rock rests, and the pure outside atmosphere is heard until the listener decides to leave the cell.

Each section corresponds to a sound file recording, indicated by the four circles in the score, meaning that these sounds could be conceived as endless loops. Their total durations vary between 10 and 45 minutes. The computer used to play the sound installation would pick a duration for each section within given boundaries. For example, the first section lasts between 25 and 40 seconds, the second section lasts between 60 and 120 seconds, etc. In order to make the “sound mobile” work, there are constraints on the beginning and ending of each section. The spokes in the score indicate segmentations of the files. The first section may not begin or end in any of the shaded segments, because some contiguous event would occur, such as a pronounced car passing by or bells ringing. Section two is constrained by starting and ending exactly in one iteration of the rock moving. The same goes for section three. For each gesture in section three, there is a number of possible connecting points in the last section—schematised as a bundle of rays in the score—which are chosen to provide a seamless transition, e.g. in terms of wind or birdsong.

The technique of using long sound files as source material, along with lists of markers and regions, possibly enriched with meta data, became a common element in all the subsequent sound installations. The interpretation of mobility became more perspicuous in two different ways. In the sound installations *CCC* (Sect. 2.3.2) and *Kalligraphie*, the focus shifted towards a play of foreground and background with no linear direction. Whereas in electroacoustic compositions, such as *Zeichnung* (2005; so actually pre-dating *Zelle 148*), the focus shifted towards the idea of difference and repetition.

The translation of the foreground/background concept into actual sound processes forks again. In *CCC*, shadowing elements in the background is done on the signal level, using for example frequency domain filtering or dynamic processors. In *Kalligraphie*, an implicit timeline is

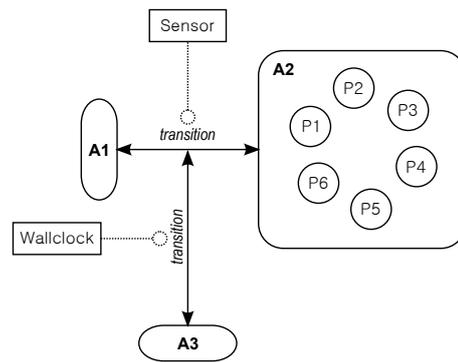


Figure 4.8: Schematic of *Kalligraphie*. A hierarchy of two transitions blends between three algorithms,  $A_1$  being “absence” and  $A_2$  being “presence” of a visitor, as determined by a sensor. After a particular wall-clock interval, a distinct intermission  $A_3$  is played.

maintained that tracks “articulated” regions across the sound layers. The layers are observed and faded out as future “collisions” between different articulations are detected.

Fig. 4.8 shows a schematic view of *Kalligraphie*. The sound layers  $P_1$  to  $P_6$  are controlled by algorithm  $A_2$ . Each of the layers consists of one particular and contrasting sound colour, spatial and conceptual connotation: For example, MAUERSEGLER is an open space sound recorded on a rooftop with groups of swifts (*Apus apus*) occasionally passing by very close to the microphone, which picks up their cries. Due to the high speed at which they pass the roof and their varying distance, the gestures have a strong spatial depth with the feeling of high velocity. In contrast, APFELESSEN is a recording of someone munching apples, referring to the body and the head of the listener inside the installation. PLEXISCHLEIFEN is a somewhat ironic layer which consists of the sound produced by polishing the acrylic glass which holds the glass plates, referring to the acoustic space and time in which the installation was constructed. And so forth. The combination of the sound colours is purely driven by preventing the co-appearance of articulations in these layers, such that at a time predominantly the articulations of only one layer are heard, whereas the other layers are allowed to persevere in a section in which they produce background sounds—the birds being distant, the munching paused, etc.

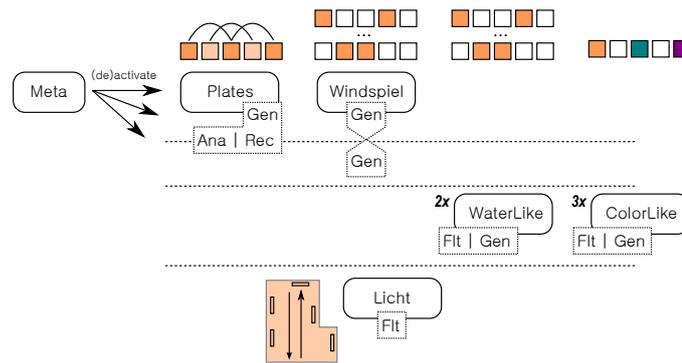


Figure 4.9: Schema of the constituent sound processes in *Dissemination*. Vertical lines partition sound layers where lower layers can filter or shadow upper layers. Small squares indicate spatialisation modes, corresponding to the five-channel diffusion in the first exhibition, and with the mode for LICHT showing the floor plan in Berlin. Each process provides different components which can act as sound generators, filters or analysing and recording stages.

#### 4.2.2 Mobility in ‘Dissemination’

In *Dissemination* the meta process, the algorithm that controls the different sound layers, uses these ideas from *CCC* and *Kalligraphie*. A schematic view is shown in Fig. 4.9. The layers are less constrained, only few co-appearances are forbidden, such as the WATERLIKE processes being mutually exclusive, and each of the COLORLIKE processes is attached to a particular plate, so they do not overlap in space. There are two motions, one—PLATES—using a generative and feedback-driven strategy, the second—used by all other processes—having the more non-directional static form of the mobile as in *Kalligraphie*.

Fig. 4.10 shows an example evolution in which the processes occur over a time span of approximately 40 minutes. The compositional form is clearly a “moment form”, with an interplay of slower and faster rhythms. The sonic complexity is hidden by the fact that motions in the parametrisation of each process are not visualised, but only the times at which sounds appear and disappear are shown. This is particularly important for the process depicted on the far left, PLATES, which is continually changing its sonic material.

The processes are inspired both by the metaphor of ‘dissemination’ as well as the experience gathered from previous works. In fact, many sounds found in this work had been used previously, as shown in Fig. 4.11. The sound of munching the apples reappears—it was used before in

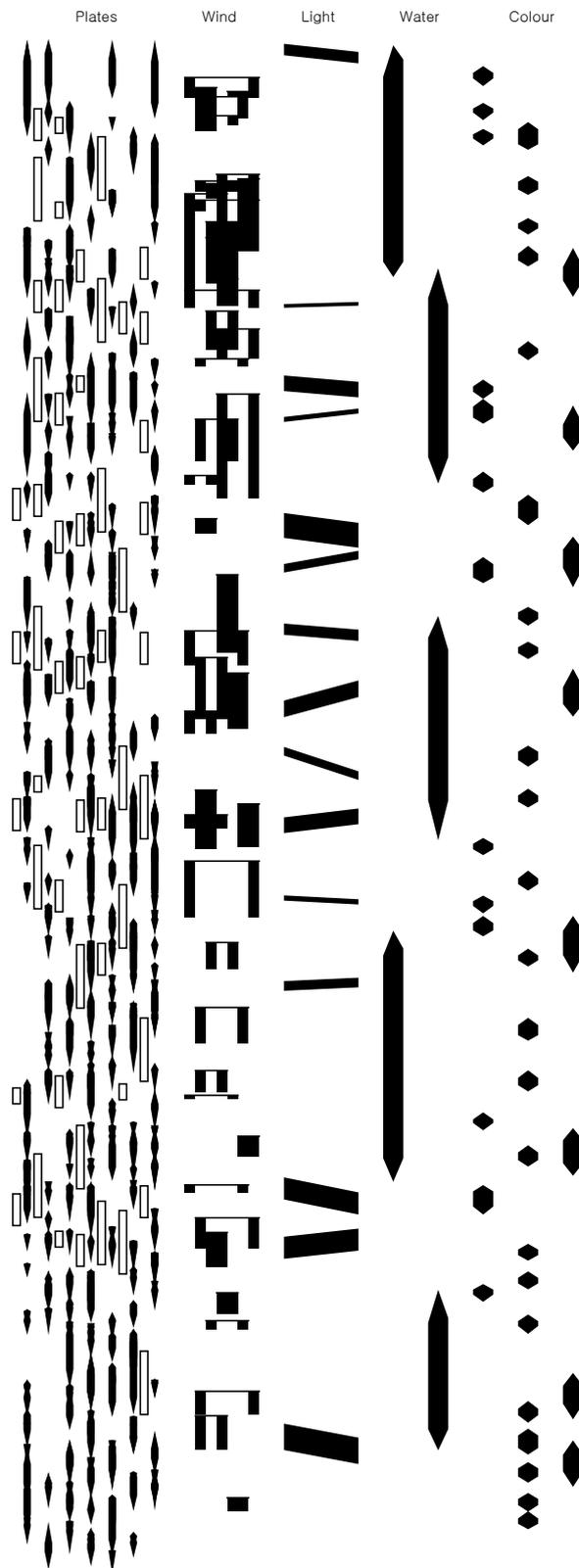


Figure 4.10: Score rendering of *Dissemination*.  
 A stretch of c. 40 minutes in  $\mathcal{T}_P$  runs from top to bottom.

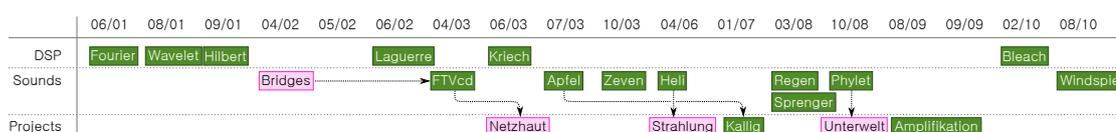


Figure 4.11: Selection of external references for *Dissemination* and their establishment over time: special digital signal processing algorithms, sound files, and previous works. Items with light background do not have a direct link with the piece, arrows indicate dependencies between these references.

*Kalligraphie*—but equally undergoes a connotational *shift*, as it now relates to the display of the seeds and alludes to the food chain, whereas the body of the listener is decentred and attenuated. While the sound of a WINDSPIEL is newly recorded, the way it is used, connecting two plates, is not unlike the use of a repetitive sound taken from radiators in *Amplifikation*'s depot, where an invisible line is drawn between the windows and the opposite wall. Indeed, the more I look at Fig. 4.11, which was created at the beginning of 2011, the less convinced I am that the manifest links are comprehensive. Although they constitute visible material traces of the compositional process beyond the limits of individual pieces, the network of connections—still material and not logical or “semantic”—might actually be aporetic.

The signal processing tools, shown in the top most row of Fig. 4.11, are a good example of this. It is futile to connect them to particular pieces, because although they often came into existence during the creation of certain pieces, they can be endlessly combined and parametrised. I have developed them over a period of twelve years—grouped together as the open source software *FScape*—reused them over and over again, so they became highly idiosyncratic, yet continuously unfolded their potential as I learned to predict their effect and experimented with their interaction. In other words, they exhibited what H.-J. Rheinberger described as the paradoxical relationship between experimenter and experimental setup: The more the setup is tied to and dependent on the skills and experience of the researcher, the more independent it becomes in his or her hands.<sup>2</sup>

<sup>2</sup>Hans-Jörg Rheinberger (1992), *Experiment–Differenz–Schrift: zur Geschichte epistemischer Dinge*, Marburg: Basiliken-Presse, p. 21.

An instance of this DSP is the WAVELET filter.<sup>3</sup> This process applies a spectro-temporal multi-resolution decomposition to a sound file, whereby a compact finite impulse response filter kernel is recursively applied, first yielding a high frequency and a low-frequency signal at half of the original duration, then using the latter in the next iteration (yielding two signals at a quarter of the duration, and so forth), until the decimation is complete. This is conventionally used as an analysis tool or to produce compact vector representations for signal compression, but it may become a creative tool: One can apply transformations to the signal *in the wavelet domain* and, like the Fourier transform, go back to the time domain, or one might just use the subsequently decimated bands as sound signals *in their own right*, an approach used in the WINDSPIEL process of *Dissemination*. Since we begin with the lowest band and proceed to the highest, this results in interesting accelerandos especially if the source material is rhythmic (in WINDSPIEL, the strokes of a windspiel are assembled to rhythmical sequences before entering the DSP chain). Due to the decimation, each band in itself is perceived as having a broadband spectrum.

The horizontal dimension within each column of Fig. 4.10 indicates spatialisation across the glass plates (7 channels). For WINDSPIEL, the pairing of Fig. 4.10 means that each sequence is played “stereophonically” on two plates whose position is chosen from an urn.<sup>4</sup> There are many other instances of such DSP uses, and the creative re-appropriation of signal processing is again the subject of Sect. 4.4.1.

Another example of the reappearance of a technique is LICHT (“light”), which has been used in *Amplifikation*: After the setup of the sound installation, impulse responses from each glass plate are taken, choosing a reference location in the space. Inverting the spectral shape of these responses and using a minimum phase transformation to lessen temporal smearing, filters are obtained which “undo” the colourisation that each plate adds to the sound excitation by its own resonance. Using convolution, these filters are faded in and out, moving across the output signals of the plates. As a result, the physical presence of the sound emitted by a plate is diluted, and the sound appears much more translucent, floating independently in the space.

---

<sup>3</sup>For details on the ‘Daubechies’ wavelets employed by *Fscape*, see Ingrid Daubechies (1988), ‘Orthonormal Bases of Compactly Supported Wavelets’, *Communications on Pure and Applied Mathematics* 41(7), pp. 909–996

<sup>4</sup>An urn is a collection of items from which one draws without replacement. When the urn is empty, it is automatically refilled with the initial items.

### 4.2.3 Writing ‘Dissemination’<sup>5</sup>

*Dissemination* was written with a preliminary version of the *SoundProcesses* framework presented in Sect. 5.11. Neither was the persistent database in existence, nor was the representation of sound processes fully established: Thus, when looking at the programming, an arbitrary line must be drawn between the preparatory work and the composition in the narrow sense. Since both the framework and the composition have been managed using a versioning system, we are able to look at the development over time. The repository containing the actual composition was opened in August 2010. The lines of code written in the Scala programming language have been manually categorised as belonging either to the technical infrastructure of the system or carrying actual musical meaning.

The commit history is visualised in Fig. 4.12. The premiere taking place on 18 September and the beginning of the second exhibition on 20 October are marked by vertical lines. It can be seen that in the beginning more time is spent with the programming of the infrastructure than the actual composition. Since the framework was rather new, various adaptations were required to realise the ideas for the project. Infrastructure code generation decreases to nearly zero towards the end of the first composition cycle with another final spike due to preparing the work for autonomous operation during the exhibition.<sup>6</sup>

The other distinction made here is between newly written code and code derived from previous projects or from older stages of the same project. This follows from a recursion model of the composition process<sup>7</sup> which proposes that this process is driven by material injected from outside the system as well as (re-)transformation of material already inside the system. Encountering this pattern in manifest condensations—computer artefacts such as the code—could indicate that something similar is happening in the psychic system of the composer,<sup>8</sup> but it is important

---

<sup>5</sup>Parts of this section’s text and figures are taken from Rutz, ‘Limits of Control’.

<sup>6</sup>A bug was found that caused the system to become unstable after a couple of hours running, so several measures had to be taken to work around it.

<sup>7</sup>Hanns Holger Rutz, Eduardo Miranda and Gerhard Eckel (2011), ‘Reproducibility and Random Access in Sound Synthesis’, in: *Proceedings of the 37th International Computer Music Conference*, Huddersfield, pp. 515–522.

<sup>8</sup>See for example the study by D. Collins who observes a composer and identifies a combination of both linear and recursive motions in the development of the piece: David Collins (2005), ‘A synthesis process model of creative thinking in music composition’, *Psychology of Music* 33(2), pp. 193–216

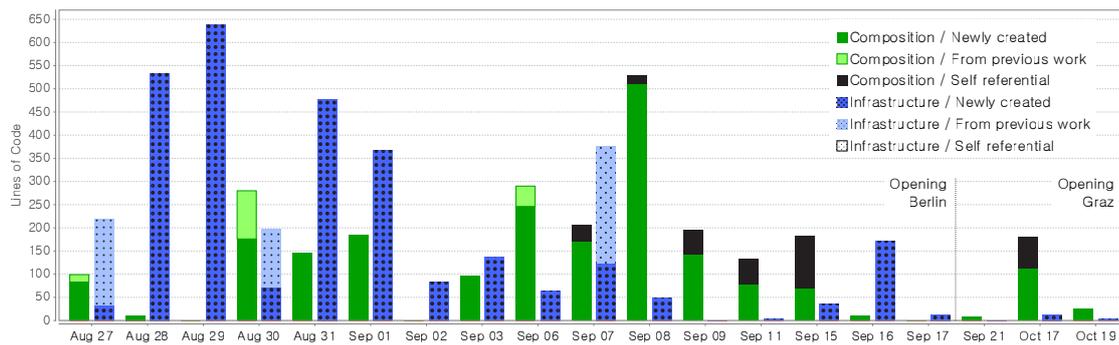


Figure 4.12: Code commits to the *git* repositories of the composition of *Dissemination* and frameworks it depends on. Line count includes lines created, lines edited and lines deleted. Multiple commits per day are integrated. Code carrying musical meaning in the narrow sense is shown in green/flat and contrasted with code used to build the infrastructure, used for debugging, and so forth, which is shown in blue/dotted. Bars are split to distinguish newly created parts from parts derived from previous work.

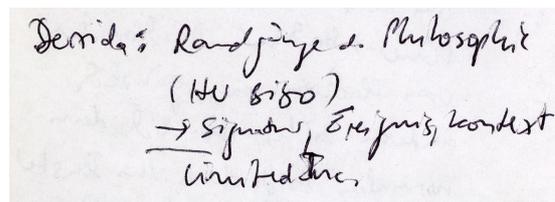


Figure 4.13: Sketchbook note (2003) about Derrida's 'Signature Event Context'

to remember that we do not wish to engage in a hermeneutic discourse: Derrida used the term 'dissemination' precisely to denote «disengaging from the concept of polysemics», disengaging from «a hermeneutic deciphering, the decoding of a meaning or truth».<sup>9</sup>

It is interesting that back in 2003, while preparing a lecture on "parasitism and music", I had made a note (Fig. 4.13) that I should be looking at Derrida's 'Signature Event Context'—which I never did. I had completely forgotten about it until now, and just after reading the text recovered that note.

Infrastructure code copied from previous projects indicates lack of modularisation, but can be mostly explained by the time pressure factor in the realisation of a work.<sup>10</sup> On the other hand, the adaptation of existing musical code mostly amounts to sound synthesis and transformation instruments which are reused. In the second half, self-referential composition code—code

<sup>9</sup>Jacques Derrida (1972/1988), 'Signature Event Context', in: *Limited Inc.*, ed. by Gerald Graff, trans. by Samuel Weber, Evanston, Illinois: Northwestern University Press, pp. 1–23.

<sup>10</sup>Copying code in the short term is a much faster measure than refactoring into reusable external modules which pays off in the long term.

which is derived from code within the same project—begins to increase and finally amounts to roughly half of the additionally produced code volume. Musically, this can be interpreted as variation: More sound processes are introduced which share structural similarities with previously created processes but are then differentiated, for example by using other sound files, other parametrisation, probabilities, spatialisation etc. Fig. 4.12 therefore seems to support—although not “prove”—the model of an increasingly recursive behaviour of the composition process.

#### 4.2.4 Permeable Boundaries

The lack of control over a compositional process driven by an incomplete software, the “deficiency” inherent in any software written specifically for computer based composition, may be the greatest strength. Creativity is often said to be fostered by constraints one has to deal with, developing solutions “around” these constraints.

Some limitations are of a technical nature. For example, the sound processes in Fig. 4.9 are understood as operating in “layers”, so that layers towards the bottom of the diagram may observe (record) or filter layers towards the top of the diagram. Since the mobile is a dynamic structure where actual instances of these processes appear and disappear, stable reference points must be created. The solution was, using the available definition of sound processes within the framework, to insert “collector” nodes in the signal flow which would be persistent throughout the piece. Fig. 4.14 is a visualisation of the signal flow; the centre corresponds to the master bus, from which seven spokes branch off, one for each glass plate. A spoke consists of two connector nodes, allowing for three connecting points—the horizontal layers in Fig. 4.9. The LICHT process is located north-northwest of the master node.

Two observations can be made: First of all, one’s knowledge of signal routing in the underlying DSP framework (*SuperCollider*) greatly influences the way the composition is thought out. Even if abstract concepts have informed the composition in the first place, such as the metaphor of dissemination, the technology initiates its own writing process which is inseparable from the composition. Secondly, the visualisation technology repeats this rewriting: The diagram shown in Fig. 4.14 is actually an *interface* to control sound processes and came into being as a

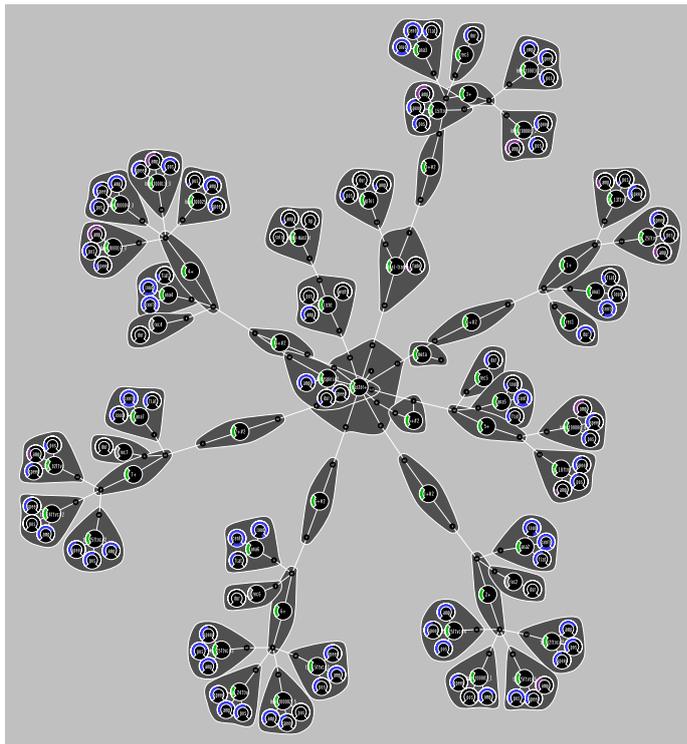


Figure 4.14: Momentary image of sound processes in *Dissemination*

live improvisation tool named *Wolkenpumpe* (“cloud-pump”, after a poem by Hans Arp). In a previous instantiation, shown in Fig. 4.15, it looked more like a traditional patch-chord system. *Wolkenpumpe* entered my sound installations through a side door: The *Amplifikation* project was embedded in a workshop with choreographers, and a performance was developed in which one could continuously shift from the structure of the sound installation to a choreographed piece live controlled by a graphic tablet.

As the visualisation turned out to be useful for monitoring sound installations, an automatic screen layout needed to be found, moreover the number of processes shown could easily go into the hundreds. As a result, *Wolkenpumpe* was rewritten based on the *Prefuse* graph visualisation toolkit.<sup>11</sup> *Prefuse* has a zoomable display and includes an animated force-directed layout algorithm which uses a pseudo-physical model to guide the automatic layout of vertices and edges. Along with the use of a convex hull based aggregation of visual elements that comprise a

<sup>11</sup>Jeffrey Heer, Stuart K. Card and James A. Landay (2005), ‘Prefuse: a toolkit for interactive information visualization’, in: *Proceedings of the SIGCHI conference on Human factors in computing systems*, ACM, pp. 421–430.



the falling velocity, the height of the seed release, the total tree height, and so forth, can be defined to simulate wind dispersal.<sup>12</sup> However, instead of using any direct biological formulas, the idea of an overall balance between production and consumption (death) was adapted.

At the boundary between first and second layer for each plate, loudness and spectral flatness are monitored. When the signal at a plate becomes too silent over a given duration, new sounds are chosen from a pool to be played back. Each sound has a lifetime; when it is exceeded the sound is either modified in speed and pitch or released. The “energy balance” (based on loudness and spectral flatness) between a plate and its spatial neighbours is monitored. When it becomes too low, the particular plate goes into “production mode”, where it records the signal currently played on its channel—formed from the sounds it spawned itself and sometimes the rhythmic patterns of the WINDSPIEL process which is occasionally inserted in the first layer (cf. Fig. 4.9). The recording is transformed into the Fourier domain and manipulated by randomly choosing one out of four transformations (granulation, removal of resonances, “frequency” warping and “frequency shifting”<sup>13</sup>). These manipulated recordings are then “injected” into the sound pool of a neighbouring plate, so a sort of temporal and spatial “echo” or “dispersal” is achieved. The sound pools are kept on the hard disk up to a certain size, and therefore sounds can be memorised over the entire duration of the exhibition. In the left-hand side of Fig. 4.10, the black elements are the sounds played from a plate’s pool, whereas white rectangles indicate the production (recording) process.

### 4.3 That Which Does Not Become Systemic

A careful look at the analysis of *Dissemination* in the previous section may reveal a fissure in the tracing of the compositional process: One moment, we were looking at a prescribed writing process (the algorithms used). The next moment, we unwittingly shifted towards looking at the process of prescribing. There is a re-entry of concept—tracing a process—but there seems to be no drawing of the mark which signifies this re-entry. Both sides seem unconnected.

<sup>12</sup>Ran Nathan, Uriel N. Safriel and Imanuel Noy-Meir (2001), ‘Field validation and sensitivity analysis of a mechanistic model for tree seed dispersal by wind’, *Ecology* **82**(2), pp. 374–388.

<sup>13</sup>Since these transformations use the signal’s *FFT transform* as input, frequency warping and shifting actually manipulates the temporal features of the signal, while granulation manipulates the spectral features.

The permeable boundary only seems to exist with respect to delimiting where the composition process begins and ends, and where its observation begins and ends (what I have called “limits of control”). In terms of the piece itself, we seem to be thrown back at «... the *derivative intentionality* of writing (Searle 2004: 20), where the code is predetermined by a human author who then yields moment to moment autonomy of execution to the machine.»<sup>14</sup> Our utopian imagination might be to create autopoietic pieces, when indeed we can only really write allopoietic systems—systems where «their primary goals are constructed in them by the designers ... their function is to produce something other (“allo”) than themselves.»<sup>15</sup>

But one can look at it from a different angle. Perhaps we should step back and question such a goal of “system design”. We *must design systems*, because that is essentially what software allows us to do, and not much else. But the aesthetic expression resulting from it has probably escaped the previous analysis, like the strange “guest” or “oikeion” that J.-F. Lyotard locates at the centre of artistic machinery:

«We have to imagine an apparatus inhabited by a sort of guest, not a ghost, but an ignored guest who produces some trouble, and people look to the outside in order to find out the external cause of the trouble. But probably the cause is not outside, that is my idea ... I have connected, and I will connect this topic of the oikeion with writing that is not a knowledge at all and that has, properly speaking, no function. Afterward, yes, when the work is written, you can put this work into an existing function, for example, a cultural function. Works are doomed to that, but while we are writing, we have no idea about the function, if we are serious.»<sup>16</sup>

So again, like every motion, if we want to make this productive, we need to keep the two processes/observations in oscillation. What matters is that looking at the process of prescribing in some form—even through detour and difference, and perhaps “failure”—is the background

---

<sup>14</sup>Nick Collins (2008), ‘The analysis of generative music programs’, *Organised Sound* 13(3), pp. 237–248; Collins refers to John Searle’s book *Mind: A Brief Introduction*.

<sup>15</sup>Francis Heylighen and Cliff Joslyn (2001), ‘Cybernetics and Second Order Cybernetics’, in: *Encyclopedia of Physical Science and Technology*, ed. by Robert A. Meyers, vol. 4, New York: Academic Press, pp. 155–170.

<sup>16</sup>Jean-François Lyotard (1988/1993), ‘Oikos’, in: *Political writings*, trans. by Bill Readings, Minneapolis: University of Minnesota Press, pp. 96–107.

against which I *did* prescribe the algorithms (the act is important, and what it actually produces, not the intention). And that these paragraphs which describe the process of prescribing will be the background of a future piece, etc.

#### 4.4 Sound Similarity as Transversal Reading/Writing across Pieces<sup>17</sup>

Such an attempt to look at the “guest” from the corner of one’s eye was made in a 2012 paper.<sup>18</sup> It reads across three different pieces: *Inter-Play/Re-Sound* is a live-electronic piece, *Writing Machine* is a sound installation, and *Leere Null* is a fixed electro-acoustic composition. The reading focuses on the way a specific signal processing technology—evaluating the similarity between sounds—is used as a composition strategy, and how this strategy is actually displaced from piece to piece, constituting its own motion or “Umwelt”.

##### 4.4.1 Signal Processing

Of the abundant possible acronyms, the term ‘digital signal processing’ (DSP) is chosen precisely because of its plain technical nature. It rests entirely on a material discourse, as opposed to the cognitive overloading introduced with ‘Music Information Retrieval’ (MIR)—assuming that we can establish the boundary between music and sound, and moreover that music contains “information” which can be unambiguously “retrieved”<sup>19</sup>—‘Machine Learning’ (ML) or ‘Artificial Intelligence’ (AI)—these assuming a correspondence between human understanding and experience and some placeholder in computation. Perhaps signal processing has the same dusty feel that made me look into databases (which were researched mainly in the 1980s).

It was mentioned that a tool I developed was *FScope*. It was started in 2000, but largely grew during the composition of a stereophonic CD album *Residual* (2001–2002). This was a major shift in my work, because while I had been “experimenting” with sounds and sound transformations before, the last album before *Residual*, titled *Stamina*, subscribed itself to the aesthetics of ambient music, using conventional structures such as rhythmic pulse and melodic

<sup>17</sup>Parts of this section’s text and figures are taken from Hannes Holger Rutz (2012c), ‘Sound Similarity as Interface between Human and Machine in Electroacoustic Composition’, in: *Proceedings of the 38th International Computer Music Conference*, Ljubljana, pp. 212–219

<sup>18</sup>Ibid.

<sup>19</sup>See the discourse on metaphors of communication in Sect. 3.2.1

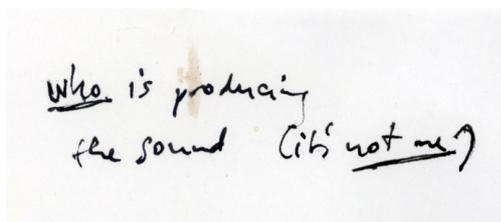


Figure 4.16: «Who is producing the sound?» Note from September 2002.

and harmonic patterns. I could already work with digital sound files on an Apple desktop computer, but relied on existing software and “plug-ins” to perform sound transformations.

By writing my own signal processing software, these transformation suddenly became an inseparable part of the music. I had some basic DSP training from my studies, but essentially I was self-taught. I remember trying to grasp algorithms published online, such as on the website of J. O. Smith III or from the DAFx conference, and probably my implementations were full of mistakes or misunderstandings. I remember my excitement at being able to actually touch the individual sample values of a digital sound and do whatever I imagined with them. I was especially fascinated by the possibility of working with arbitrarily long sound files—minutes or even dozens of minutes—and the idea of “rendering” sounds, i.e. using processes which could not run in real-time due to their heavy CPU usage or because they were operating in a random access manner and through multiple iterations on the sound.

There is a note dating from the completion of the *Residual* series that reads: «*Who* is producing the sound (it’s *not me*)» (Fig. 4.16). While it is impossible to be sure what I meant, it is interesting because *Residual* was as much “noise music” as “electro-acoustic music”, and the promise of tape music is indeed that the composer has full control over the realisation of sounds. I was barely aware of the acousmatic discourse, and certainly the sentence did not refer to the absence of the sound source in the Schaefferian sense. I believe it referred to the liberation of the decision-making process, where I could truly *experiment* through signal processing, not knowing the outcomes in advance.

One such experiment was the whole “proto-algorithmic” disposition of *Residual*: Given a set of seven distinct sound materials and a series of seven proportions—the Fibonacci numbers 1,2,3,5,8,13,21—so-called “canvases” were produced. Seven groups of seven permutations

of the seven numbers—taken as durations in seconds—were formed, as shown in Fig. 4.17. Where the same sound material appeared more than once in the same proportion, the exact same sound gesture reappears (indicated by arrows). It can also be seen that an eighth proportion 12 was added as silence, *STILLE*. It also appears that the materials labelled x1 and x2 are indeed silence, possibly to decrease the density in the subsequent transformation process. In this process, the seven sequences of each group were combined into pairs, by transforming them into the spectral domain using a single Fourier transformation, something I had implemented in *Fscape* (Fig. 4.18). I could not find specific notes of how the two complex spectra were combined, but the resulting sound, transformed back into the time domain, bears the features of both source sequences and was still well articulated and thus not a spectral multiplication (which is equivalent to a convolution).

A property of the FFT implementation was that it padded the files to a duration that was a power of two in terms of sample frames, so seven sequences of 65'' became six sequences (neighbouring combinations) of 95''.<sup>20</sup> The choice of proportion 12 might come from the fact that  $95'' \times 6 \times 7 \approx 66'$ , which is roughly the sequence sum 65, and indeed early bounces of the pieces indicate that there were again tracks with Fibonacci based durations of 1', 8', 2', 13' (others missing) and titles derived from the seven days of the week. These tracks initially were just cuts of these durations into the overall “canvas” of one hour, that is to say, track 1 had a seamless transition to track 2, etc.

This whole procedure had the function of a quasi-automatic “primer” of the time canvas, to produce a situation from which to develop the actual pieces. The notion of the canvas, relating to the visual arts, was actually inspired by my reading of the essays of Morton Feldman. He often used the idea of the “time canvas”, a given stretch of time onto which one could paint music, and he related it to the painting techniques of, for example, Mark Rothko, Philip Guston, and Piet Mondrian. I found another undated note, probably from the time *Residual* was composed, quoting from the German translation of ‘Some Elementary Questions’ (Fig. 4.19): «There is nothing in music, for example, to compare with certain drawings of Mondrian, where we still

<sup>20</sup>At a sample rate of 44.1kHz, 95 seconds equals  $2^{22} = 4194304$  frames.

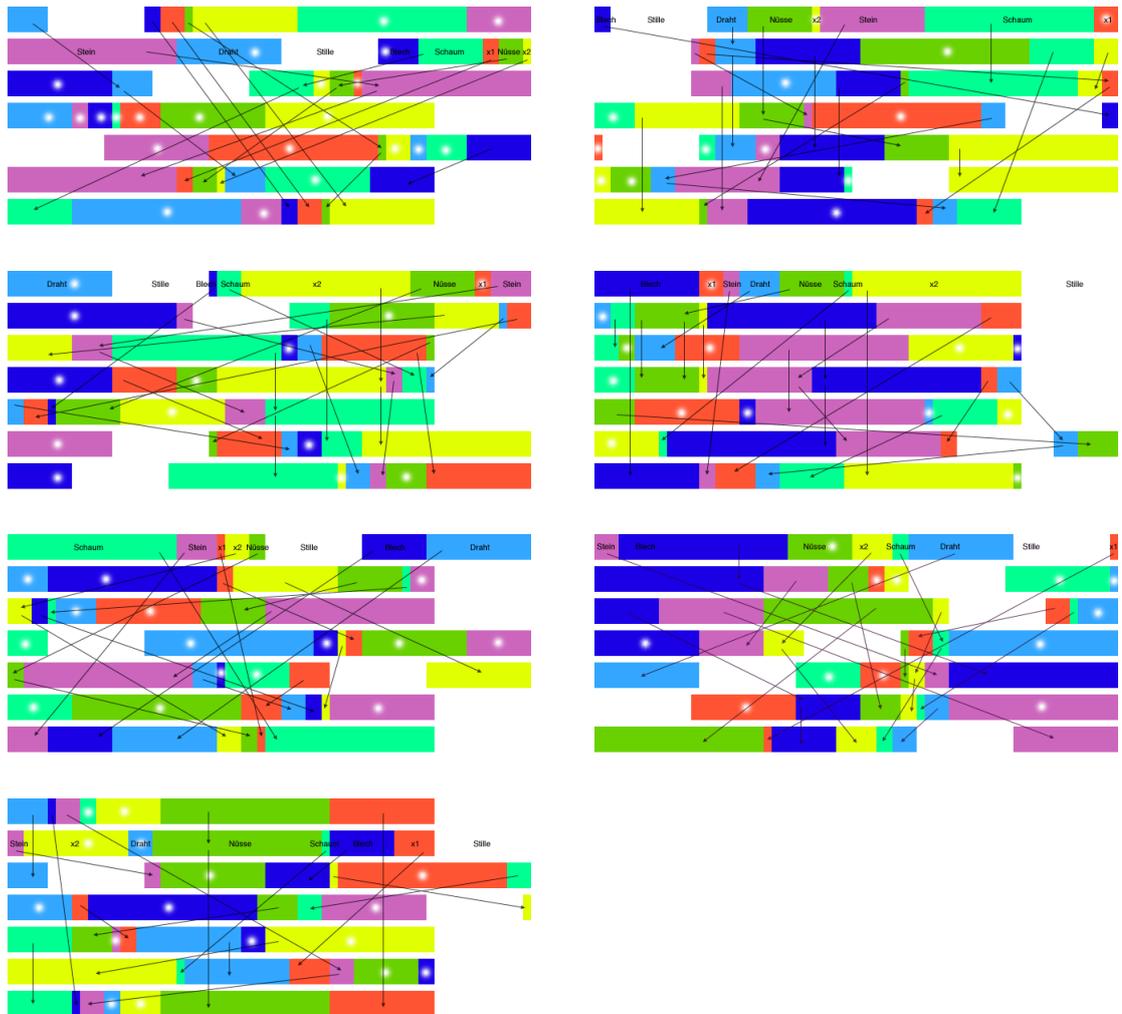


Figure 4.17: Construction diagram of the initial canvas of *Residual*. Colours correspond to the different source sound materials. Time elapses from left to right and top to bottom.

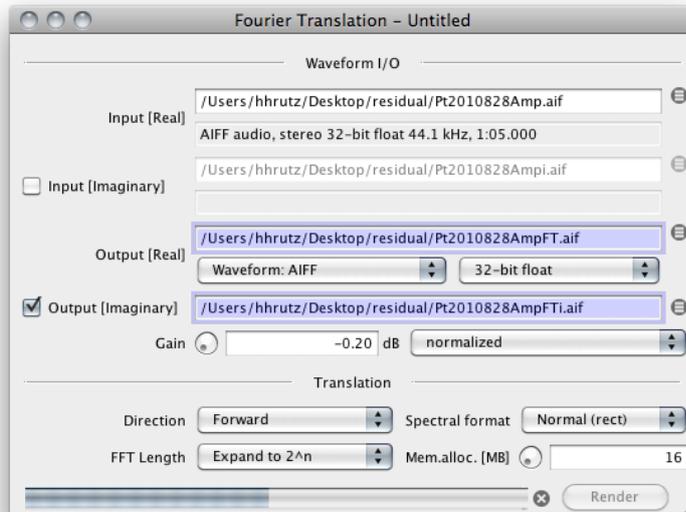


Figure 4.18: Translating a whole sound file to the Fourier domain in *FScale*

see the contours and rhythms that have been erased, while another alternative has been drawn on top of them.»<sup>21</sup>

Signal processing in *Residual* was the main methodology of creating the pieces. One of the elementary tape techniques, acceleration and deceleration without pitch correction—digital resampling—is extensively used. You can easily generate a variant of a sound that is  $8\times$  or  $1/8\times$  the original duration by resampling it three octaves up or down. While modern digital audio workstations refine their algorithms to transpose without time stretching or to stretch time without transposition, this very basic signal processing algorithm changes two perceptual qualities at once, and especially with noises which do not exhibit clear melodic elements the combination of the original sound with the resampled sound produces very interesting mixtures. Often I would “bounce” the whole current state of a piece and re-introduce it in a resampled variant, a highly effective recursive procedure.

Other algorithms used include brick wall frequency filters and filter banks, frequency shifter (Hilbert filter), frequency warp and functions applied to FFT spectra such as “zooming into” or “out of” a frequency range. There are almost no sketches dating back to December 2001

<sup>21</sup>Morton Feldman (1967/2004), ‘Some Elementary Questions’, in: *Give My Regards to Eighth Street: Collected Writings of Morton Feldman*, ed. by Bernard H. Friedman, Cambridge, MA: Exact Change, pp. 63–66.

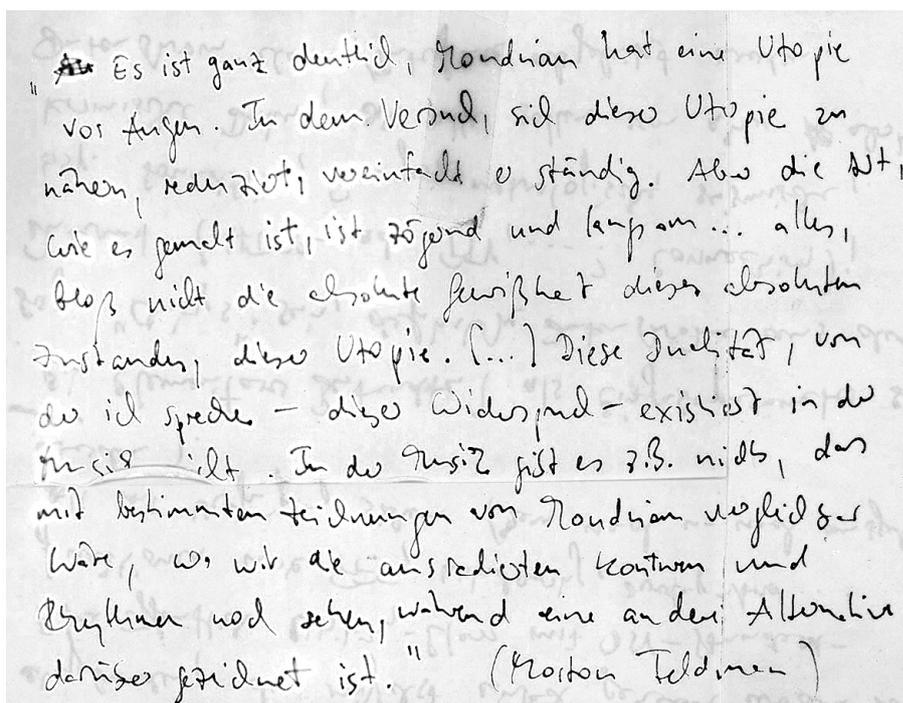


Figure 4.19: A quotation from Feldman I put down in a diary, dreaming of an overwriting technique in music akin to Mondrian

through September 2002, only some intermediary versions of the pieces, but it seems that the composition process was very intuitive and, instead of having a prescribed formal plan, based on observing where the pieces drifted through these recursive applications of signal processing. Also it seems that towards the end, certain structures were “coloured” or emphasised by liberally introducing other sound materials not directly evolved from the original canvases. The durations of the pieces started to shift, and at some point their number decreased to five pieces of 10'44", 16'22", 7'04", 17'50", and 9'14". A programme text was only formulated when the pieces were finished in October 2002. A note reads: «Turns out not to be a circle but an excentric spiral». I remember that with the seamless transition between the pieces, the original idea was that the last piece directly connects to the first.

The role that was designated to signal processing stems from the post-romantic traces that can be found in American abstract expressionism as well as in the music of Cage and Feldman.<sup>22</sup>

<sup>22</sup>One can debate these traces, of course, but it goes beyond the scope of this section. For example, see Edward M. Levine (1971), ‘Abstract Expressionism: The Mystical Experience’, *Art Journal* 31(1), pp. 22–25; or Leonard B. Meyer (1963), ‘The End of the Renaissance?’, *The Hudson Review* 16(2), pp. 169–186

The submission of long stretches of a continuous sound stream to transformations, the way the results are scattered across the canvas, strongly resonates with the idea of an all-over technique. The 2002 programme note mentions the intent of «not pushing the sounds around but trying to figure out how they would behave autonomously», again a quote from Feldman and his idea of a natural state of time that exists before human involvement. It also highlights the use of noise—for which the signal processing is mainly responsible—as a source of *polysemy*, not dictating meaning but being open to interpretation and producing «an imaginary landscape» (a reference to Cage). The more metaphorical and figurative parts of the programme note seem to use noise as a *hallucinogen*, an agent which permits the mind to construct these landscapes. Somewhere between this work and *Dissemination*, a motion has taken place from polysemy to dissemination, an abandonment of naturalism.

A testament to this motion is a different programme note from 2003, written in the context of the Bourges electro-acoustic competition, in which the last piece of the *Residual* series received an award. The erosion responsible for ‘residuals’ is less related to the material process of creation—e.g. rewriting in the sense of Mondrian or blurring the edges in the sense of Rothko—but to the double nature of time:

«The name [Residual] points to the process of temporal erosion in the mind of the composer as well as the listener. The strange conflict between elaborating an atmosphere and the ongoing composition process that covers a time span of several months and includes vast changes in mood, thought, social contacts etc. The strange activity of the listener in whose head a piece is kept in flux and changes meaning over time.»<sup>23</sup>

This of course is a timid motion: It is still talking about meaning and the psychic state of the composer, of which an all-over technique is just a medium, but it recognises the fissure in “meaning” and the *strangeness* in the relation between composing time and performing time.

<sup>23</sup>[http://sciss.de/texts/tap\\_residual.html](http://sciss.de/texts/tap_residual.html) (visited on 27/07/2013)

#### 4.4.2 Delineation

The idea of sonic erosion is also found in the 2005 tape composition *Zeichnung*. The German title has multiple meanings in English, including ‘drawing’, ‘outline’ and ‘delineation’. It connects in several ways to *Residual*. In the earliest notes I found, dating from January 2005, the Rothko idea of “bleeding edges” through iterative repainting is mentioned; to begin with an initial layer of sound which «slowly vanishes as it is repainted in an ‘aligned’ manner». Several questions were formulated: How to arrive at an initial structure? How do the iterations or layers relate to each other, are they opaque or transparent? What is the character of ‘delineation’, e.g. edged, soft, exact or coarse? What does repainting mean technically? Should (temporal) distortions be allowed? How is spatiality constructed?

The term which is used extensively is the ‘trace’. A trace signifies an action, a movement, a gesture, a subject. It is not just the frozen past but points into the future—if one encounters a trace, one can follow it in two directions. Traces decompose over time. The human memory is a trace: «The head is like a mobile [!] which slowly swings forth and back and may get into turbulences.» And finally, what is determined by the trace? Rhythm, dynamics, spatiality, brightness.

The actual disposition then establishes again a primer/canvas, based on some proportions and series, as shown in Fig. 4.20. Four fields *A*, *B*, *C*, *D* are defined, corresponding to four basic sound materials: plastic foil, stone, broken glass and foam chips. The four fields exist simultaneously with a total duration of twelve minutes. They are divided into a varying number of sections. The sounding sections of each field are separated by silent sections. The trace is embodied by a sequence of imitations, beginning in section  $B_1$ —called “Urszene”—which is the template for the imitation  $D_7$ , which in turn is imitated as  $D_5$ , and so forth following a random order of the 16 sections.

In order to match the target duration of each section, the imitation is either carried out on an accelerated or decelerated template, or the imitation drops parts of the template or creates additional material. Fig. 4.20 shows these operations on the left-hand side. The imitations are to be carried out by ear, carefully listening to the template and splicing target material so that

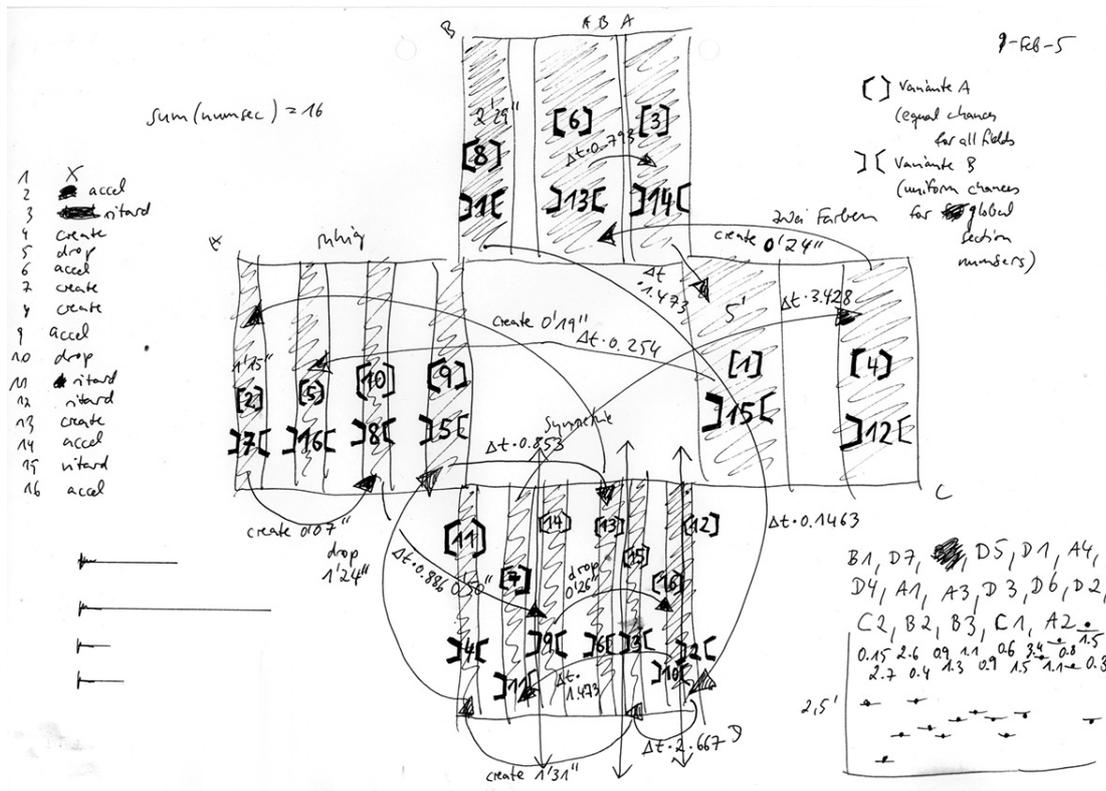


Figure 4.20: Plan for the construction of Zeichnung

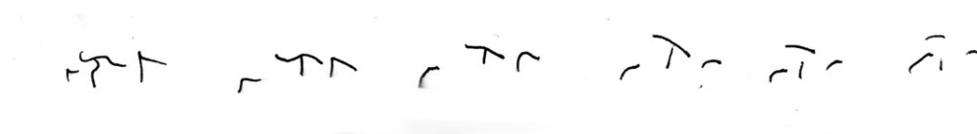


Figure 4.21: Sketch illustrating the gradual change of form due to imperfect imitations

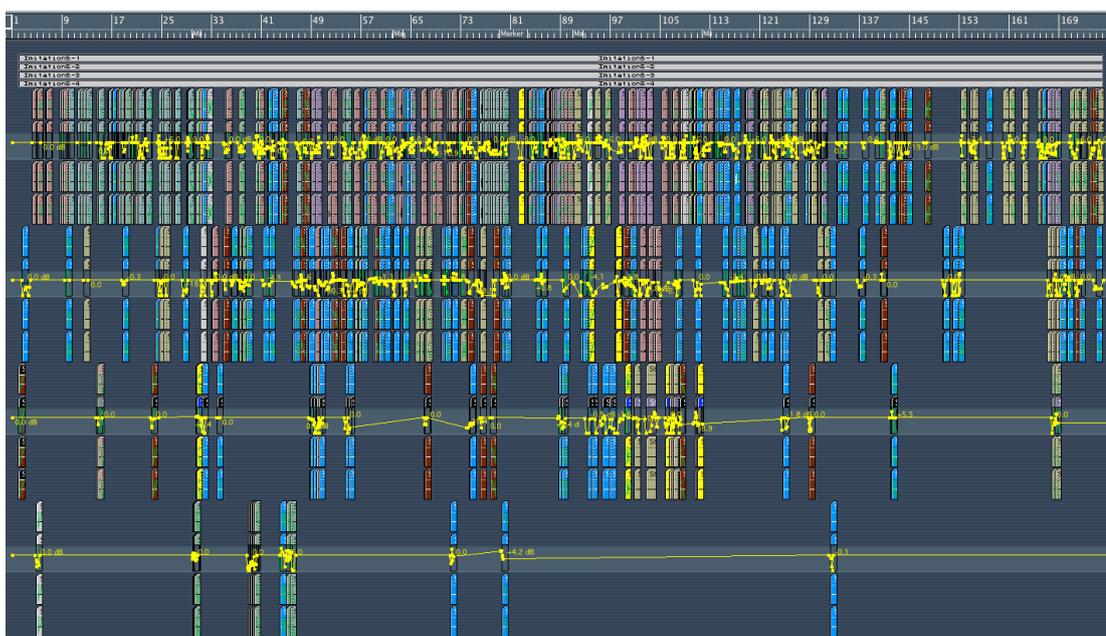


Figure 4.22: Timeline arrangement of imitation no. 6. The template is shown folded up on the top, four iterations carried out to construct the imitation below, each split into four channels.

the perceptual qualities such as gesture and micro rhythms are obtained. The imperfection of each imitation produces the trace, as indicated in a sketch in Fig. 4.21.

While this approach does not involve specific computer-based algorithms, it can be understood as a “human signal processing”. It involves a performative element, because each imitation is very laborious and requires a lot of time, and it plays with paradoxical elements such as the task to imitate the sound of stones with recordings taken from broken glass. The sections are around five minutes long and contain several hundreds of cuts, as well as individual volume break-point curves. Fig. 4.22 shows the timeline editor with four iteratively produced layers of the sixth imitation. All sound recordings were made quadrophonic in a sound booth, and were split into four channels due to a limitation of the multitrack editor used (*Logic*).

The result also contained some absurd aspects: In order to create the imitations, I was listening to template and target simultaneously but spatially separated, mostly via headphones. This led to a perceived melding of the separate channels as soon as there were *some* common elements between them. The ear is willing to accept two sounds as one connected gesture based on some features, even though the two sounds appear very different when played successively. Pitch and formants often play a minor role compared to the loudness contour.

Although produced in a sound booth specifically for the piece, the source sound materials were recorded in long improvisatory sessions independent of the work to which they were subject in the imitation process. This is also somehow absurd, because it gives them on the one hand the feel of “field recordings” which have not been “controlled” by the composer, while on the other hand the sound is totally controlled in the final work as the material is precisely cut and arranged. What can be witnessed here is a play with the conflicts of the compositional process itself. However, in retrospect it seems that this play is absurd in a different way: It is barely noticeable in a performance of the piece itself, but only visible through the meta-text and analysis.

In the final piece, which still lasts twelve minutes, although the sections had been stretched—resulting in a “zoomed in” version of the initial layout (Fig. 4.23)—there are dedicated moments which present template and imitation after another, hinting at the process. But the musical gesture and form are determined by other factors. The hatched portions in Fig. 4.23 mark transitions between different concurrent imitations. According to a simple typology (Fig. 4.24), each imitation belongs to one of four classes: It may either initiate a section, terminate one, occur isolated, or mediate between a preceding and a succeeding section. Each class is associated with a particular chain of DSP to “merge” or “morph” the overlapping sections.

Finally, there are multiple iterations in which the material is spatialised to four channels, mostly following the number of concurrent sections which are spread in space. Although it does not form part of this analysis, I would like to point out that I developed a custom software *Meloncillo* for this spatialisation, based on trajectories which are represented as ordinary sound files instead of break-point functions, allowing any signal processing algorithm to be applied to them.

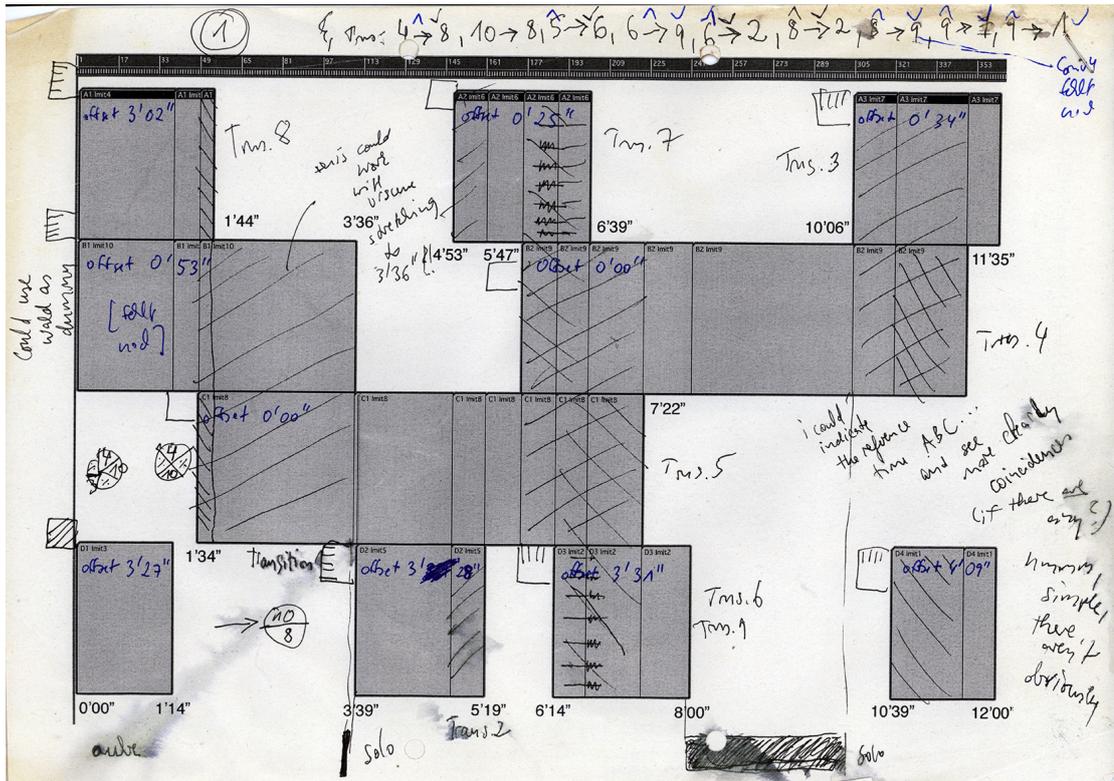


Figure 4.23: Form plan showing the timeline of the four materials. Handwriting details the transitions between the four fields to be carried out in the next step.

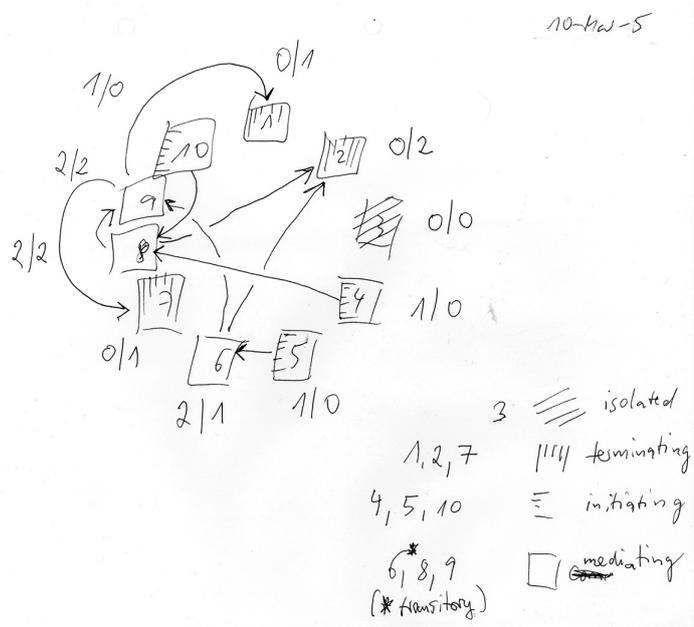


Figure 4.24: Typology of transitions derived from the co-occurrences and successions of imitations. The symbols used for the resulting four classes have been copied into Fig. 4.23.

To conclude, in *Zeichnung* the coordinates of signal processing received notable translations. The fact that the imitations were carried out manually seems to be less significant than the shift in the meaning of trace. It moves away from the archeological view of trace as something to be rediscovered, perhaps underlined by the facts that the original sound sequence in the end was not included in the piece and that the sound materials have been recorded in an “uninterested” way. Instead the productive moment of trace is put into the centre. Whereas the final shape of *Residual* was a product of many factors and interventions from my side, here rhythm, dynamics, spatiality and brightness indeed arise from the trace. It has been said that the musical form is strongly influenced by the co-presence of sections which led to the rendering of the transitions according to Fig. 4.24, but the quality of these transitions is a consequence of the surface of the imitations themselves. Despite—or because of?—the formalised and rigorous nature of the piece, the iterative process played out well as a source of “Anschlußfähigkeit” or connectivity (cf. Sect. 3.3.5).

#### **4.4.3 Inter-Play/ Re-Sound**

So far the self-imposed game between me and the computer, as shown in the composition process of *Zeichnung*, has not been directly included in the performance structure of the pieces. While *Inter-Play/Re-Sound* (2011) is also a pre-written piece, key aspects of it are established during the live performance.

Its components are an amplified derelict piano, a human performer and a computer. The performer is equipped with a miniature microphone and is asked to focus on highlighting the physical structure of the piano and to avoid producing “piano music” sounds. In particular, the hammer mechanics are removed, leaving “empty keys” and direct access to the strings, for which a preparation scheme was developed. Fig. 4.25 shows some of the object with which the strings were prepared.

The sounds get amplified and are projected back onto the corpus of the piano through a number of transducers (Fig. 4.26), making it a spatially extended and reverberating loudspeaker. The computer registers the sounds produced in the first improvisatory part of the piece, and extracts their temporal and spectral features. An interplay begins with the computer using a selection

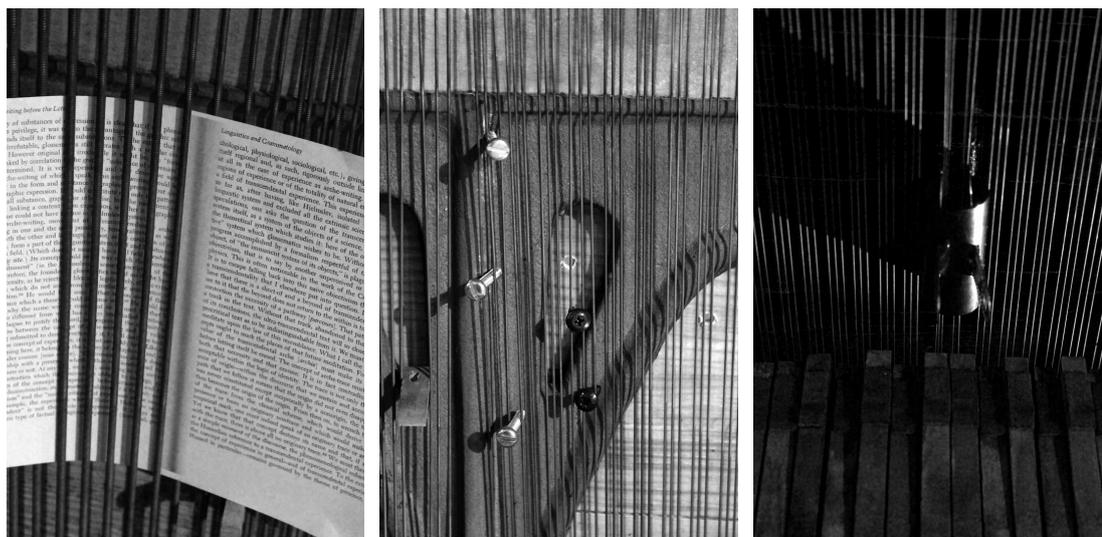


Figure 4.25: Different ways of preparing the strings of *Inter-Play/Re-Sound*

of algorithms which operate on the recorded and analysed material, sooner or later being fully in charge of the situation and dismissing the performer, who becomes another listener. At this point, the feedback circle is closed by having the computer turn its observation onto itself: running the analysis against its own output signal. Although the piece is now in a generative mode which could go on forever, parameter trajectories chosen within given randomised bounds allow the piece finally to decay. Fig. 4.27 shows an example of such a ‘tendency mask’.

During the first part, the live signal is continuously written to an audio buffer as well as a feature buffer holding the evolving Mel Frequency Cepstral Coefficients (MFCC) of the signal. Furthermore, markers from an onset detector are stored. The piece, like *Dissemination*, is realised with a provisional (ephemeral) version of *SoundProcesses* and the *Wolkenpumpe* interface for control purposes. Fig. 4.28 shows a screenshot. In the top part, the MFCC analysis of the microphone signal is shown, where time elapses from left to right. Beneath the coefficients, white vertical markers indicate the detected onsets.

The algorithm comprises a “body” of eight concurrent processes—each inspired by one sense, e.g. hearing, seeing, touching, and so forth. They iterate through three states, idle, “thinking” (analysing without producing sound) and “playing” (producing sound), based on bounded random durations or being triggered by observations. Durations and parameter ranges are mostly



Figure 4.26: The piano equipped with transducers during the development in the ICCMR lab

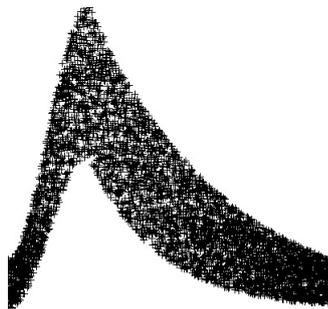


Figure 4.27: A tendency mask is a parameter boundary (ordinate) which dynamically changes over time (abscissa).

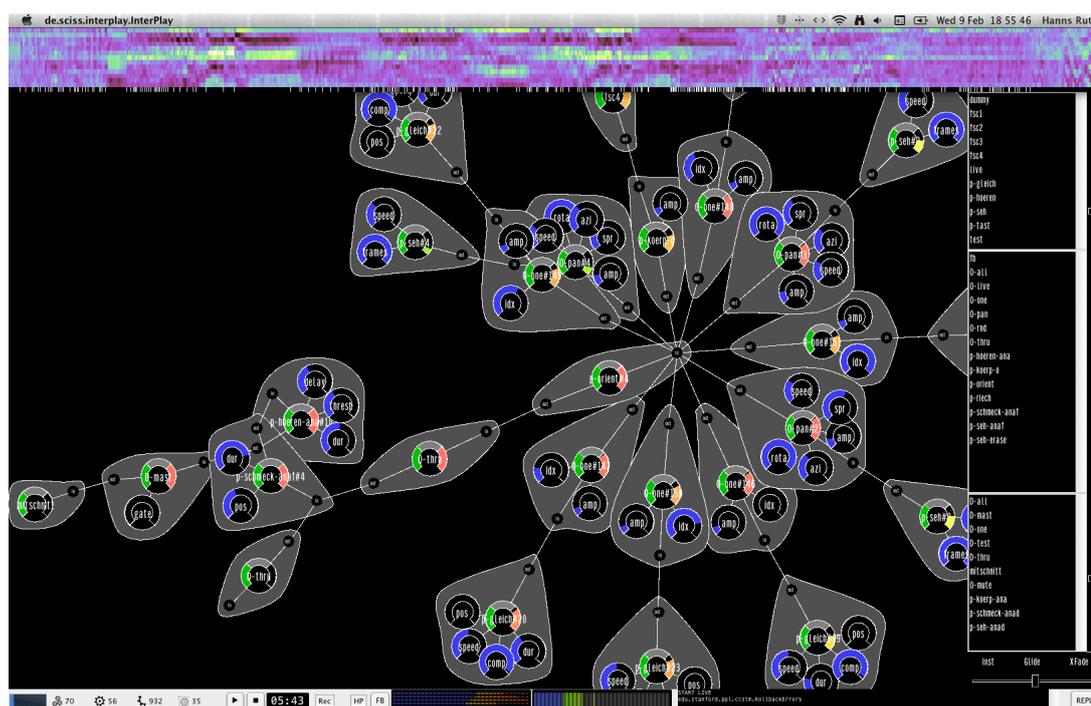


Figure 4.28: Screenshot from *Inter-Play/Re-Sound* which provides a visualisation of the sound processes in the centre and of the feature vector evolution in the top part

defined through tendency masks over a logical timeline spanning an assumed approximate overall duration.

Of these processes, four transform, filter and inject material without a dedicated analysis stage. The other four use features extracted from the material: Process HEARING measures averaged loudness and replays short transient sounds between onset markers when the loudness falls below a time varying threshold. Process EQUILIBRIOCEPTION searches for steady portions of flat (noisy) spectra in the developing live audio buffer, and plays them back as short concurrent loops. Resonant sounds from striking or stroking piano strings, hitting on the corpus etc., are thus ignored, while the background of the human actions, formed by the small sounds and reverberations behind the foreground, is amplified and intensified, essentially creating an additional space through varying loop durations across channels. Later in the piece, the playback speed is modified as well, resulting in timbre variations.

The two remaining processes TOUCH and SIGHT are based on sound similarity. TOUCH uses a set of given *sound profiles*, previously stored MFCC evolutions of short duration (0.5 ... 1.0''), taken

from prototypical sounds like flageolets on the strings, the empty keys and objects requested as preparation of the strings, such as a piece of paper clamped between adjacent strings (Fig. 4.25). The process then searches through the available live buffer for the best matches according to a similarity measure, cuts out small portions from the buffer at the matching positions, applies a spectral whitening filter to make the sound distinguishable from the live signal and appear more remote. Fig. 4.29 shows an example search result across a fully filled buffer.

In contrast, SIGHT looks at a sub-tree of the current DSP graph and picks up its sum signal, measuring its short term spectral evolution and using this as template. Instead of finding best matches, it generates a continuous similarity signal across the whole available live buffer. It then applies a threshold in order to obtain a condensed version of this buffer: Only those sounds showing a minimum similarity remain, and those sounds are moved together to form a new homogeneous gesture, with a duration ranging from a few to dozens of seconds.

The overarching thought is to create semi-autonomous structures, for example by amplifying the background and creating space. This space is the space between human and computer, not as an alienation effect, but to allow both to breathe and coexist without submitting to stimulus-response patterns often found in live electronic pieces. It aims at presenting the inexhaustible possibilities lying in the material left behind by the performer, while maintaining a strong sense of coherence and avoiding the feeling of the computer exerting an arbitrary power over the situation.

#### 4.4.4 Writing Machine

In *Writing Machine* (2011), the performer is gone and we are left with a sound installation which runs fully independent of human interaction. As in *Dissemination*, we encounter again the discrepancy between the prescription process and the process prescribed. This installation bases its notion of ‘writing’ on the broad meaning as formulated by Derrida: The grapheme is the manifestation of the process of writing-as-trace, an infinite chain of signification and an absence of “presence” or re-presence of an original signified.<sup>24</sup> Signification, if one wants to

---

<sup>24</sup>Jacques Derrida (1967/1997), *Of grammatology*, trans. by Gayatri Chakravorty Spivak, Baltimore: Johns Hopkins University Press, chap. 1 and 2.

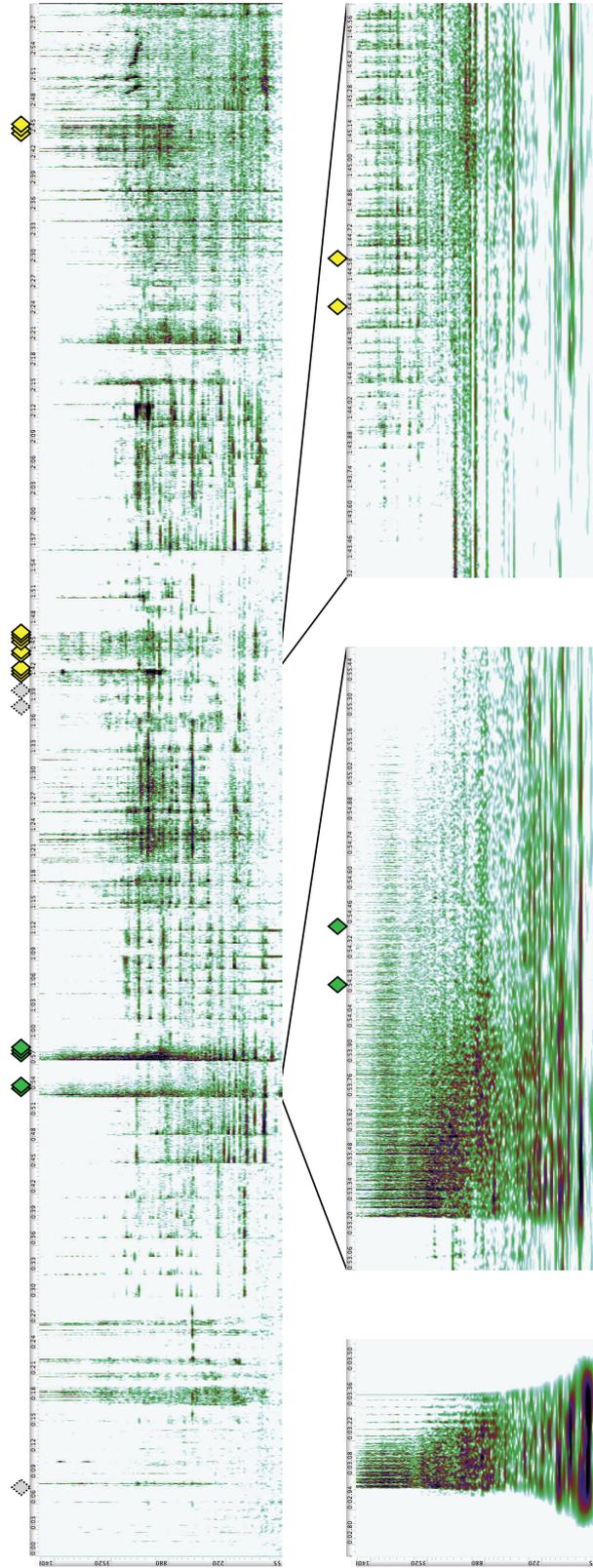


Figure 4.29: Example selection in process TOUCH: The upper sonogram shows a complete 3' live buffer. The target template is shown in the bottom left. The algorithm was instructed to find the twenty best matches, the results show as diamonds. Matches due to low volume are indicated in grey, matches coming from the same type of gesture (strings hitting clamped paper) are shown in green, and other gestures with similar spectral envelope in yellow. Centre and right bottom show magnifications of two matched regions at the same plotting scale as the target sound.

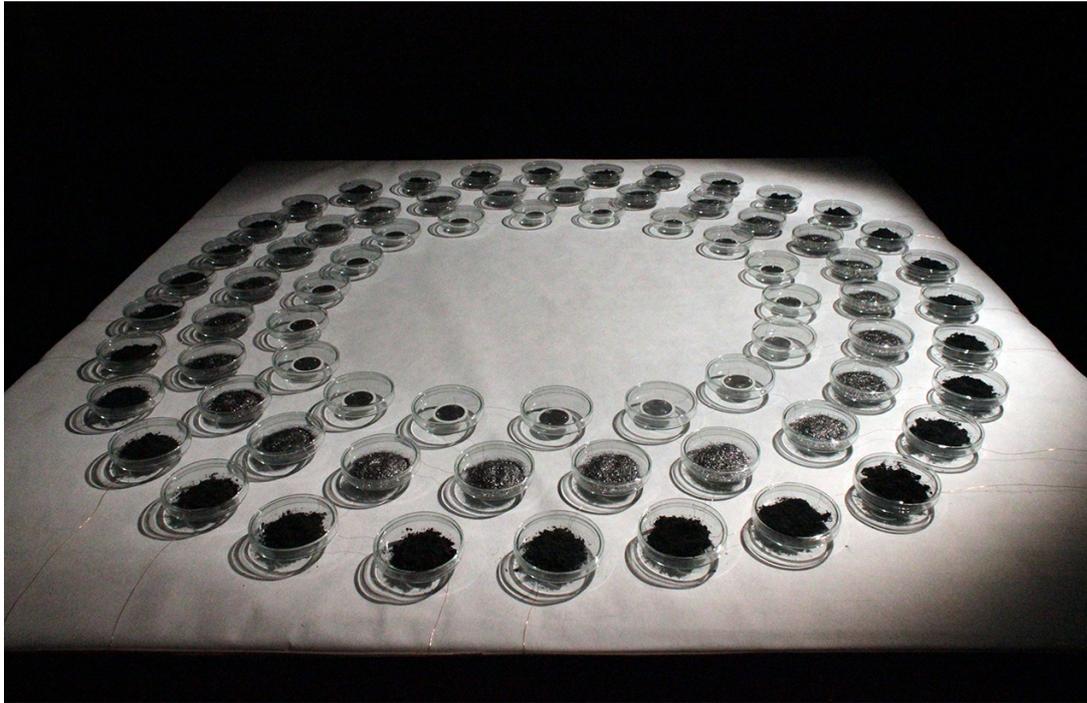


Figure 4.30: Photo of the sound installation, size approx.  $2 \times 2$  m.  
Festival SONICA, Ljubljana 10/2011.

keep the term, lies entirely in the transitions. *Writing Machine* takes these ideas quite literally, using an algorithm which re-writes a sound phrase over and over again, focusing the attention on the changes introduced in each iteration.

At first glance, the work resembles *Zeichnung*, although the feedback chain does not include the human composer. But because the chain runs with indefinite duration, the imitative aspect of each iteration with respect to its immediate predecessor becomes a subordinate one. The chain is reflected in the visual presentation (Fig. 4.30) by using the geometric shape of the circle. A poriferous circle, however, which acts as a system of consumption: The machine “eats” a live signal, preferably one with conventional signification, such as from a television channel. This signal is fed into a sort of database or pool from which all audible material is constructed.

The arrangement is that of a laboratory experiment—a tableau of petri dishes, put into vibration by piezo electric elements, and displaying heaps of graphite powder, the disintegrated mineral which borrows its name from graphein (to write). There are two iterations: The first describes the algorithmic cycle which departs from the previous situation or *phrase*, chooses parts of

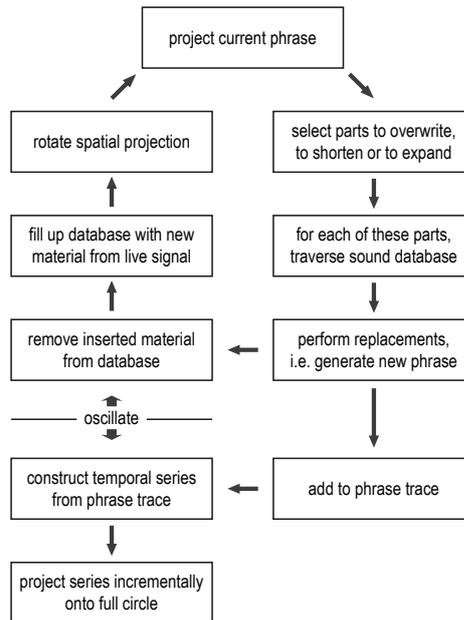


Figure 4.31: Diagram of the *Writing Machine* algorithm

the phrase to be overwritten, then looks for “suitable” materials in the database and overwrites those parts, in other words re-writes the phrase. These steps are shown in Fig. 4.31. Multiple piezo elements are grouped together to form channels, and the phrase, whenever it is restated, advances to an adjacent sector of the physical tableau, performing thus the second type of iteration, a spatial rotation over the course of a few minutes.

Everything is controlled by *motions* which are basically ultra-low frequency stepwise oscillators or random walks. There is a motion for the duration of the fragments overwritten, for the position of the fragments within the current phrase, for the cross-correlation length when looking for similar sounds in the database, and so forth. The procedure is quite similar to the approach taken in concatenative sound synthesis. As an interchangeable example, C. Frisson, C. Picard and D. Tardieu write: «This method involve[s] four steps: segment the target, extract feature[s] from the target segments and the grains, choose the grains that will replace each target segment

and finally concatenate the chosen grains to create the final sound»,<sup>25</sup> and part of the diagram is akin to the one outlined by D. Schwarz.<sup>26</sup>

The difference is that we do not seek a *final* sound, and the interesting moments originate exactly from the imperfection of the reconstruction. The imperfection is, so to say, the *contrast agent* for the spatial and temporal unfolding of the recursion. For example, in the area of automatic sound design and texture generation, repetition in the choice of matching sounds might be seen as a problem,<sup>27</sup> but we found that they are one of the most recognisable peculiarities of our system, as the repeatedly chosen sounds are layered slightly off the target sound's position, adding to the palimpsest's clarity. On the other hand, a notable characteristic of the *Writing Machine* is that the database is constantly drained, so each sound chosen has a certain chance of being removed from the database, making space for refilling—and with the live signal, the material is never exhausted.

With the joy in seeing the constantly new combinations emerge, comes another side which lies in the dark: A system was built which registers acts of writing, revealing the computer as a better prosthesis of the human memory—more calculable, more durable, but also external memory as more prone to manipulation: «Everything faded into mist. The past was erased, the erasure was forgotten, the lie became truth.» (George Orwell, 1984)<sup>28</sup>.

#### 4.4.5 *Leere Null*

The last example employing signal processing as a strategy is the fixed media composition *Leere Null*. It was composed in two parts. The first part, written in 2011, is a five-minute stereo tape, somewhat pedantic and focused on the process. The second part, composed in 2012, is a

<sup>25</sup>Christian Frisson, Cécile Picard and Damien Tardieu (June 2010), 'AudioGarden: towards a Usable Tool for Composite Audio Creation', in: *Quarterly Progress Scientific Reports of the numediart research program*, vol. 3, 2, pp. 33–36.

<sup>26</sup>Diemo Schwarz (2006), 'Concatenative sound synthesis: The early years', *Journal of New Music Research* 35(1), pp. 3–22.

<sup>27</sup>cf. Marc Cardle (2004), *Automated Sound Editing*, tech. rep., Cambridge, UK: Computer Laboratory, University of Cambridge, §4.1.3

<sup>28</sup>This is not as far-fetched as it may seem. Demaine, Iacono and Langerman indeed had the same association when developing their concept of retroactive data structures, cf. Erik D. Demaine, John Iacono and Stefan Langerman (2007), 'Retroactive Data Structures', *ACM Transactions on Algorithms (TALG)* 3(2), 13:1–13:20

four-channel tape of around eight minutes, a restatement of the first part, but using a different methodology and yielding a perceptually very distinguished result.

The point of departure arises from utopian ideas about composing. Some are very tenacious: The composer as the originator—faced with a blank sheet, He creates *ex nihilo*. This is one way to explain the title which is taken from the novel *Roadside Picnic* by the brothers Strugatsky. While the term “Empties” is used in the English translation, the German version is peculiar in that it uses a pleonasm: “Leere Null” means empty (or void) zero. In the book it is used to describe objects allegedly left behind by aliens: two copper plates spaced apart with “nothing” in between them, and no force can tear the two apart. The Empty fascinates due to a paradox: It is fully permeable for the senses—you can put your hand inside—yet its meaning is completely opaque. Like the blank sheet, the object could be seen as a container—the contour of the zero—which is susceptible to being filled with our imagination (rewriting the opaque meaning).<sup>29</sup>

The genre of “tape” music imposes another paradox or obstacle: During the writing of the piece, the composer can hear the sounds and gestures it is made of in arbitrary succession, indeed they can be moved around on the “time canvas”, looking down onto them with the comfortable vision of the bird’s eye (cf. Fig. 2.9). However, when the piece is presented, only one path in the decision space is left to be presented. This problem is not unique to electroacoustic music, but the composer’s capability of “double listening” greatly emphasises it.

The first part of the piece is a play with these two elements: To create *ex nihilo* (despite the impossibility), and to reverse the arrow of time. The solution I chose interrelates these two. The approach lies again in the use of sound similarity search and in the rewriting of phrases with the help of this search. For this it was necessary to create a multitrack timeline editor which could be programmatically controlled, and this tool was published as an open source project<sup>30</sup>. It allows the construction of timelines both manually via a graphical user interface, as well as

---

<sup>29</sup>Only just now, while I was finishing writing this text, I came across an interesting connection with J. Lacan, who presented similar thoughts on “creation *ex nihilo*”, using the example of the potter’s vase which is something signifying an emptiness and the potentiality of filling it: Jacques Lacan (1992), *The Ethics of Psychoanalysis, 1959–1960*, ed. by Marc E. Carvallo, trans. by Dennis Porter, vol. VII, *The Seminar of Jacques Lacan*, London: Routledge, ch. IX

<sup>30</sup><http://github.com/Sciss/Kontur> This is an ephemeral system which has been replaced by *Mellite* in newer works, allowing the traces of the compositional process to be recorded; see Sect. 4.6.

through programs written in the Scala language. Audio regions are shown as sonograms and can be manipulated in the typical manner, i.e. adjusting volume and fade curves, as well as cutting and splicing. The combination with easily accessible programming interface is quite unique, as most similar systems are either tailored towards symbolic manipulation instead of manipulation of electroacoustic material, or their graphical interface is more in the state of a visualisation and not a fully operable editor in its own right. This multitracker is complemented by the similarity algorithms which have been extracted into a separate library also published as an open source project<sup>31</sup>. It makes use of the feature extraction algorithms found in the SuperCollider system.

While the correlation of MFCC vectors alone proved useful in the live electronic piece, I was seeking better *temporal* synchronisation between target and match, as both are to be superimposed. For example, if there is a percussive sound, the algorithm should not only find a sound similar in terms of the spectral evolution, but also in terms of its temporal envelope. Furthermore, in part 2 which is constructed to great length by the computer alone, we need a way to adjust the volume of the matched sound particles. The program thus includes the loudness contour of the sounds, and for each search we may specify a weight which balances the cost between spectral and temporal features.

The use of a sliding MFCC matrix cross- or auto-correlation has been described in previous literature,<sup>32</sup> and I wish simply to add that I have achieved the most convincing results when using normalisation of the coefficients with respect to their occurrence in the corpus. I first created feature files from my database, comprising several gigabytes of sound recordings which had been made over the last years, and calculated the 1st and 99th percentile for each MFCC coefficient and the loudness contours. In the actual search, cross-correlation is performed using sliding windows, each of which is normalised according to their mean value and standard deviation. Matches are ranked by highest normalised cross-correlation and a Euclidean distance is used to balance spectral and temporal features.

---

<sup>31</sup><http://github.com/Sciss/Strugatzki>

<sup>32</sup>cf. Cardle, *Automated Sound Editing*, §3.5.1

## Part 1

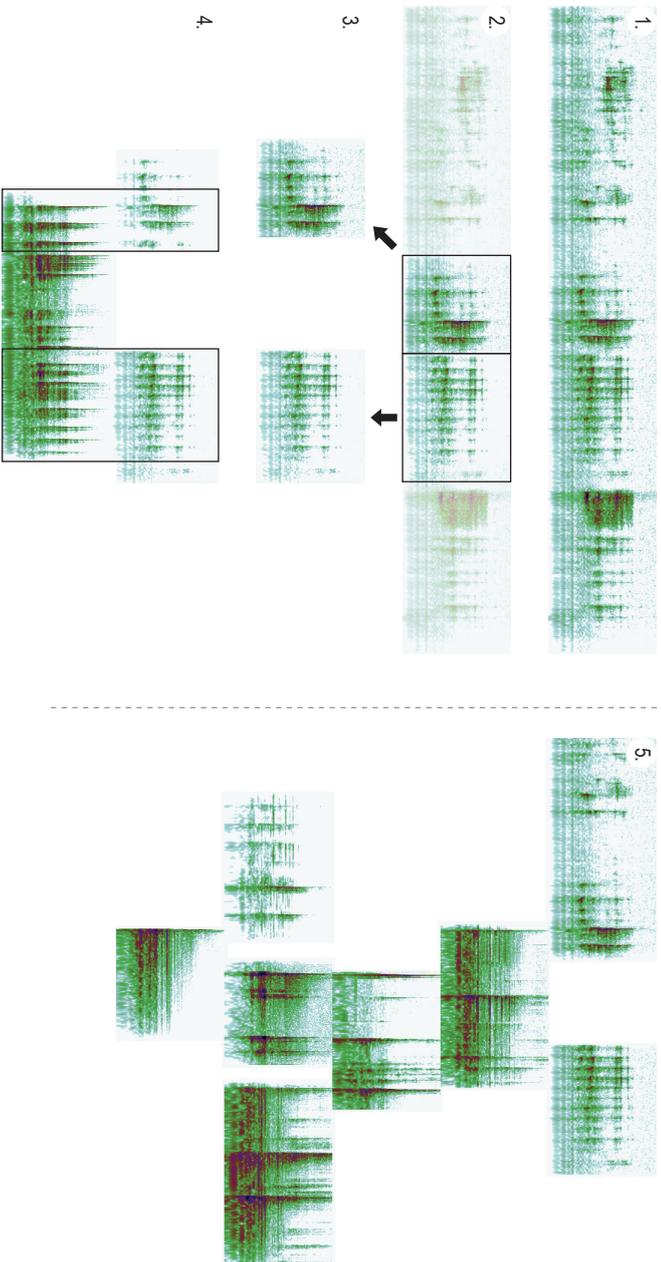
As the Empties are found objects, the first part begins with a fifteen-second quotation from Mikhail Belikov's *Raspad*, a film about Chernobyl which for me is, through Andrei Tarkovsky's interpretation of the *Roadside Picnic*, linked to the novel. The protagonist is crying for the loss of his friend. The first half of this part is constructed as shown in Fig. 4.32. The initial gesture is gradually extended and overwritten in a dialogue between composer and computer. A suitable sub-region to be overwritten is selected along with a cutting point. The algorithm is given further parameters for the similarity search, such as the balance between spectral and temporal features, a minimum and maximum duration to *insert between* the left and right part of the sub-region which will be split as part of the iteration, as well as constraints for the database such as limiting the amount of gain that can be applied to the found material. The interface for this is shown in Fig. 4.33. When the search is completed, the matches are ranked by correlation, and the composer can listen to the effect each choice of sound would have after insertion in the timeline. For this part of the process it was crucial not to stick to the "best" matches as seen by the algorithm, but often the musically surprising ideas were found further down the list.

This strategy combines coherence and change in an interesting way: while there are always good matches in the beginning and ending of the found sounds, due to the dilation they are completely unconstrained in their middle part (other than being subject to the composer's selection). The algorithm is further refined by gradually introducing *transformations* to the material injected. This is a creative method already noticed by Cardle in the field of texture synthesis.<sup>33</sup> I have used *FScape* again, which contains a series of transformations which are (quasi-)invertible. Resampling and frequency shifting are two of them which are applied here.<sup>34</sup> The invertible nature—a sound can be accelerated with resampling, and then slowed down again by using another instance of resampling with an appropriate factor—can be exploited so that the database does not need to be re-analysed, which is crucial as it contains several gigabytes of audio files. Instead, the inverse transformation is applied to the target sound which is only a few seconds

---

<sup>33</sup>Ibid., §2.1.1.

<sup>34</sup>If frequency shifting is used with anti-aliasing filters, shifting up and then down again leaves a frequency band at the upper end of the spectrum empty (for example 20 kHz to 22 kHz), therefore I call this quasi-invertible.



*Figure 4.32:* Iteration in constructing the first half of the first part in *Leere Null*: (1.) Initial sound gesture. (2.) Dissected into two smaller sub-regions. (3.) The previous phrase is extended by specifying a minimum and maximum dilation. (4.) The actual spacing is a result of the similarity search. It finds the matches which produce a best correlation *both* with the left region and the right region within the specified dilation bounds. (5.) Multiple sounds have been layered at time 0'48''.

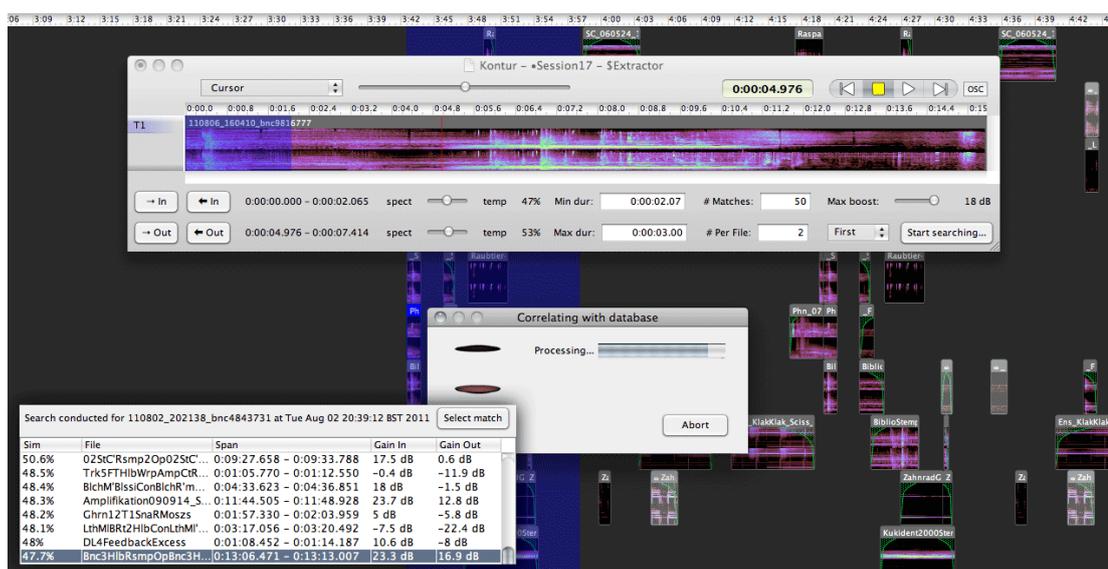


Figure 4.33: Screenshot with the user interface for *Leere Null*. The background shows the main timeline, the window in front shows the current selection to be subject to similarity search. A result table for an example iteration is superimposed on the bottom left.

long. For example, if we wish to inject material lowered by two semitones, corresponding to a resampling factor of 112.25%, we apply a resampling with factor 89.09% to the target sound, proceed with the search, and apply the desired resampling after a sound has been selected from the matches.

To address the time arrow paradox, the second half of part 1 is constructed from reverse: I begin with the result of the *last iteration* of the first half, and then proceed in a similar way, however selecting two sub-regions as search targets which are already spaced apart, resulting in a contraction of the phrase length in each iteration. Also the phrases are assembled from back to front. As a result, the second half, as it is perceived in the concert, starts with a very dense texture as it corresponds to the last stage in the layering process. This impression is intensified by ultimately having applied the process separately for the left and right channel. The texture then gradually becomes clearer and slows down again like the first part. Although one seemingly departed from very different situations, both sections end in the same gesture, thus questioning cause and effect.

## Part 2

For the second part of the piece, strategies were chosen which I hoped would yield a more organic and acousmatic unfolding of the materials. Re-writing should be used again, but on a different scale and in the form of what could be called “Hidden Strugatsky Chain”—the target sound is in fact the entire first part in itself, although it is never heard as such. This hidden target is approached by splitting it up into many small segments. At the same time, the algorithm should gain more autonomy and process entire sound layers on its own without further human intervention. As the second part was planned to have up to twice the duration of the first part, the first part (the hidden background layer) was incrementally stretched.

No bouncing had been involved in the construction of the first part, and it was thus possible to automatically part the overall time canvas into layers, reducing the “polyphony” or density in each layer to allow for more clarity in each search iteration. Six layers were constructed, three of which spread over the whole duration, while only material from the densest parts was left over for the other three layers. Each of the six layers was then processed by two different strategies. The layers were mixed to mono signals, while the searching process constructed *quadrophonic* outputs.

Both strategies operated by finding sounds similar to the target layer, but the channels were treated in opposite ways, achieving either congruency or complementarity between them (Fig. 4.34): The first strategy was called IMITATION. The cost function for matching not only considered similarity between database sound and target layer, but also in between the four channels it was asked to generate. The reuse of the same match for more than one channel was disallowed. Furthermore, the instruction was to keep coherence between its own successive choices. This is illustrated in Table 4.1. The overall cost function had therefore to consider fourteen components, where a tendency mask was given to balance between source/target similarity, diachronous channel similarity and synchronous inter-channel similarity. Brute force search kept several computers busy for a week and led to a simplification in the inter-channel measurement by only requiring adjacent channels to maximise cross-correlation.

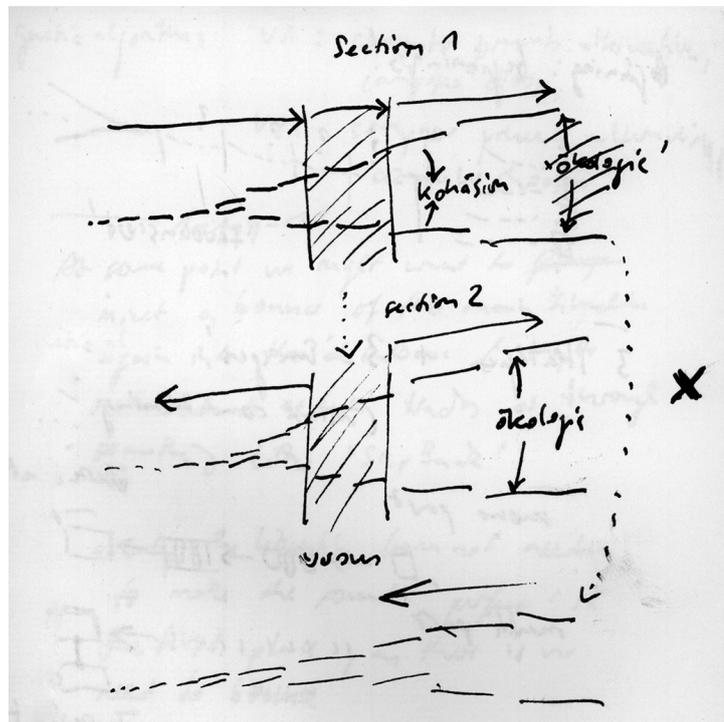


Figure 4.34: Original concept of the two strategies in *Leere Null*: Kohäsion (cohesion/imitation) and Ökologie (ecology). This assumed that each iteration follows along the horizontal arrows and uses the recursion based on the previous phrase. With the adaptation of using a pre-existing hidden target layer, a weighted similarity between the previous phrase and this target layer is used.

	Target Layer	Synchronous				Diachronous			
		Channel 1	Channel 2	Channel 3	Channel 4	Channel 1	Channel 2	Channel 3	Channel 4
Channel 1	+		*	*	*	+			
Channel 2	+	*		*	*		+		
Channel 3	+	*	*		*			+	
Channel 4	+	*	*	*					+

Table 4.1: Cross-correlations included in the matching cost function. + means larger correlations are better (similarity), - means smaller correlations are better (dissimilarity). Cells marked \* are treated as + in the IMITATION strategy, and - in the ECOLOGY strategy. Entries are commutative, i.e. the correlation between chan. 1 and 2 is the same as the correlation between chan. 2 and 1.

The choice of database was also significant. For the IMITATION, I recorded more than an hour of new sounds which were improvisations with metal objects in a sound booth while listening to the target layers on headphones, in the hope that I would generate gestural material rediscovered in the search. The aural profile of the target layers was also partly transformed through moving short time convolutions. The database was split between the six layers in order to generate more distinguished output layers. The outcome was highly satisfying: The quadrophonic experience was indeed as intended perceivable as a “cubistic” sculptural space in which the listener is immersed. The inter-channel correlation produced a feeling that uniform gestalts were projected into the space, while at the same time giving them dimensionality. Also the diachronous coherence parameter had a strong positive effect on the perceived musicality.

The second strategy was called ECOLOGY and based on the idea of stratifying the sounds between the channels: The cost function of the inter-channel correlation was inverted—the more dissimilar two sounds were, the higher was their rating, yielding contrasting spectra as well as contrasting temporal envelopes. The database was equipped in a way similar to the first part, using a very large selection of different recordings from my archive, although no sounds were present which had been used in the first part. I would like to note that “ecology” is thus meant merely as a measure of sounds being able to coexist without disturbing each other. It neither denotes ecology in the sense of soundscape composition, nor in the sense of engagement with the performance space (as in Di Scipio’s work). Instead, it resembles the model that A. Eigenfeldt and P. Pasquier used in their algorithmic realtime soundscape composition *Coming Together: Freesound*,<sup>35</sup> in which multiple agents select sounds based on disjunctive spectra. The results of this strategy are equally interesting as the imitative approach, yet very distinct in their airy transparency.

Another crucial aspect for the success of this process was the ability to automatically perform segmentation, since I am no longer required to do this by hand. Segmentation is also considered the most decisive part of concatenative sound synthesis,<sup>36</sup> however in the CSS case, the database

---

<sup>35</sup>Arne Eigenfeldt and Philippe Pasquier (2011), ‘Negotiated Content: Generative Soundscape Composition by Autonomous Musical Agents in *Coming Together: Freesound*’, in: *Proceedings of the 2nd International Conference on Computational Creativity*, Mexico City, pp. 27–32.

<sup>36</sup>Schwarz, ‘Concatenative sound synthesis: The early years’.

is already pre-segmented, whereas in our case segmentation is a dynamic process and regards “grains” with often several seconds of duration. Again I applied the idea of ecology, such that segmentation is based on maximising dissimilarity between the left and right half of a sliding matrix. A similar approach is described by J. Foote.<sup>37</sup>

In the final construction of part 2, although not using strict section boundaries as suggested by Fig. 4.34, the ECOLOGY material enters the time canvas later than the IMITATION material, creating sensitivity for their respective qualities. The finalisation was guided by constraining human intervention to an almost purely subtractive role, using two stages: First the density in each layer was reduced, and some annoying sounds removed. Then larger chunks of each bounced layer were cut out in the mix between the eighteen layers, taking care only to emphasise motions which were already formulated in the generated layers, and to give the piece an overall envelope.

#### 4.4.6 Pervasive Space

In the first part of the discussion, it was signal processing in the broad sense, in the second part it was the operationalisation of sound similarity which constitutes the space pervading the individual pieces. I have chosen the formulation of the “pervasive space” to allude to Spencer-Brown’s *Laws of Form* (cf. Sect. 3.3.1f). Like Lyotard’s “inhabiting” guest, the “pervading” space is some troublemaker that is difficult or impossible to observe. It seems to me more than a coincidence that both use a variant of “dwelling” to describe this oikeion or space. More than an actual, delimited space, the pervasive space of Spencer-Brown is a virtual one. Space is an axiom he uses to allow the operation of distinction to make a first severance—in the space—producing a form which indicates the inside of the severed space, and yet again space on its outside. Through the re-entry of the form in successive distinctions, one can speak of different levels of spaces which are numbered  $s_0, s_1, s_2$  etc. The virtuality lies in the fact that  $s_0$  is just the sense we have of where we stand:

---

<sup>37</sup>Jonathan Foote (2000), ‘Automatic Audio Segmentation Using A Measure Of Audio Novelty’, in: *Proceedings of the IEEE International Conference on Multimedia and Expo (ICME)*, vol. 1, New York, NY, pp. 452–455.

«In evaluating [an expression]  $e$  we *imagine ourselves* in [the shallowest space]  $s_0$  with  $e$  and thus surrounded by the unwritten cross which is the boundary to  $s_{-1}$ .»  
[Emphasis added]<sup>38</sup>

The imagination of the space is tied to the *unwritten* mark which would situate that space inside another containing space. In the second chapter, Spencer-Brown succinctly defines “knowledge” as those states that have been marked. What we can “know” about processes is only what we have distinguished as forms produced by the processes. We were looking at the particular selection of pieces (forms) in the previous sections in order to catch a glimpse of the underlying process or “guest” from the corner of our eye. Each new sketch or piece is the materialisation of the inexhaustible reservoir of unwritten crosses.

The space as form reservoir is called a ‘medium’ by Luhmann, who makes some important remarks on how to understand “it”. First, avoiding any kind of transport metaphors of communication,<sup>39</sup> the possibilities of the medium are not exhausted by producing forms, but on the contrary the possibilities are regenerated. The medium needs the forms, as accidental as they may seem, to persist over time through such regeneration. Using physical metaphors, Luhmann describes a medium as an ensemble of «“loosely coupled” elements», which offers no «resistance» to the forms—very similar indeed to the airy pervasiveness of Spencer-Brown’s space which does not inhibit the placing of crosses. Forms then are a «tighter» coupling of elements within a medium. A medium can only be observed through the forms it produces, again in accordance with Spencer-Brown’s definition of knowledge.<sup>40</sup>

While Luhmann’s distinction scheme of medium and form is useful here, the effort he undertakes to apply this to the “art system” and to “explain art” is maybe less so. However, it is worth mentioning the observation with regard to the stability and closure of the artwork. For a sociologist, of course, it is essential that multiple observers generate “sense”, the shortcut of which is to say that observations can be re-actualised. The reference or “identity” of an artwork

<sup>38</sup>George Spencer-Brown (1969/1979), *Laws of Form*, New York: E.P. Dutton, p. 42

<sup>39</sup>See the discussion of Krippendorff in Sect. 3.2.1f.

<sup>40</sup>Niklas Luhmann (1997), *Die Kunst der Gesellschaft*, Frankfurt a.M.: Suhrkamp Verlag, ch. 3; an English translation is available as Niklas Luhmann (2000), *Art as a Social System*, trans. by Eva M. Knock, Stanford: Stanford University Press

Jean-François Lyotard	oikos, writing	function
George Spencer-Brown	space	form
Niklas Luhmann	medium	form
Jacques Derrida	pure trace, différance	grapheme
Hans-Jörg Rheinberger	epistemic thing	technical object

Table 4.2: Conceptual pairs of “condensation”

that warrants such re-actualisation is given by its material manifestation.<sup>41</sup> This manifestation he says is «art-external . . . anchorage» which requires the separation of individual artworks, the breaking of the chain of signification in order to allow one to return to the same.<sup>42</sup>

From my perspective, I feel both reinforced and misled by this observation. Reinforced, because it confirms the idea that any such thing as “an artwork” or “a piece of music” is a partition which at best serves as a bookmark, but otherwise obscures the underlying motions which—like the *pervasive* space—are only indicated by the changing forms. The misleading part is a metaphysical contamination of the argumentation which differentiates between the material substrate and the artwork. And this is where I want to bring in H.-J. Rheinberger and my understanding of ‘interface’.

Rheinberger’s two opposing notions involved with processuality are ‘epistemic thing’ and ‘technical object’. By now it should become apparent that each of the authors introduced in this whole discourse is trying to tackle similar problems: How to observe the unobservable and how to talk about something like ‘process’ which cannot be or refuses to be represented. I have brought them together in Table 4.2. It goes without saying that no term replaces another term in the same column. There is a shift from row to row, but in the overall picture, we can understand process as this ensemble of terms, or more precisely by the ensemble of relations that invisibly connect left and right column.

<sup>41</sup>One can already see that Luhmann will have to go through hoops to address performance art which is time-specific and installation art which is site-specific, and consequently he restricts himself, whenever possible, to more traditional forms such as painting, music and poetry. . .

<sup>42</sup>Luhmann, *Die Kunst der Gesellschaft*, ch. 3.

#### 4.4.7 Playing out Intrinsic Capacities

Where the relation in Lyotard's row is unidirectional—uninterested writing becomes functional—the operations of Spencer-Brown and Luhmann are reciprocal but characterised by an asymmetry; the marked space can be further differentiated, the mark is always copied *into* the inner side; forms are ephemeral, yet “stronger” and “assertive”, the medium is transparent or penetrable, yet persistent. Rheinberger's two notions are also complementary to each other, but—at least in my reading—neither of them is “stronger” or includes the other. The motion by which epistemic things and technical objects are translated into each other appears to be a sort of oscillation between two partners at eye level, and this seems to have to do with the material substrate dismissed by Luhmann.

Rheinberger has formed his ideas through the study of the history of science and in particular the establishment of experimental systems in molecular biology, a field in which he himself had previously worked. Using the concept of ‘experimental systems’, he aims to understand the circumstances that permit scientific “break throughs” to occur, and how these are informed by the embedding of the researcher in his laboratory and research environment. An experiment of course, if taken seriously, is an endeavour of which the outcome is not foreseeable. Since the “object” under consideration is underdetermined and barely tangible, it is even impossible to formulate a clear a priori hypothesis. In that respect, Rheinberger clearly distances himself from the positivism of a Karl Popper. One could say, a true experiment generates questions instead of answers. I believe we can compare the scientific process of generating “new knowledge” and the artistic process of generating “new experiences”. Rheinberger himself hints at this possibility when referring to the art historian George Kubler who says that artists work “in the dark” in order to explore the unknown.<sup>43</sup>

Rheinberger builds his ideas on many things, referring for example to the work of Thomas Kuhn and Bruno Latour, including system theory and post-structuralism, in particular the work of Derrida, whose *Grammatology* he translated into German. Like Derrida, he views ‘writing’ as a fundamental process beyond the literal meaning of written language. The idea of the

<sup>43</sup>Hans-Jörg Rheinberger (5th May 2007), ‘Man weiss nicht genau, was man nicht weiss: Über die Kunst, das Unbekannte zu erforschen’, *Neue Züricher Zeitung*.

‘grapheme’ as the material condensation of writing is also found in Rheinberger’s work. The important aspect from system theory is recursion, which appears in the form of *differential reproduction*: This reproduction denotes the «material process of generating, transmitting, accumulating, and changing information»,<sup>44</sup> a process of repetition which aims at variation, rather than a process of replication which aims at identity, again opposing the classical concept of a reproducible “reliable” scientific experiment. This does not mean that there are no constants in an experiment. On the contrary, in each iteration of a scientific experiment a form of *cohesion* must be established that allows it to be compared to the previous experiment, but at the same time the system must be open to disturbances from the environment so that unforeseen things can happen. This dynamic reminds one of Luhmann’s medium/form distinction which essentially is a play of constancy and variation.

Rheinberger focuses on what he calls the material culture of science, the individual laboratory conditions and apparatus. It is often their constraints, such as dealing only with a very specialised and limited aspect of the research, which becomes an enabler for new knowledge. The novelties arise not primarily from the scientific reasoning (the scientist’s mind), but from situations directly related to the apparatus.<sup>45</sup> The interaction between apparatus and researcher is summarised in the following peculiarity:

«the more [the scientist] learns to handle his or her own experimental system, the more it plays out its own intrinsic capacities. In a certain sense, it becomes independent of the researcher’s wishes just because he or she has shaped it with all possible skill.»<sup>46</sup>

In the text from which this quotation is taken there follows a quote from Jacques Lacan who names this seeming contradiction “intimate exteriority”, which resonates with the description of Lyotard’s guest and Derrida’s imperative that deconstruction be performed *from inside* the

---

<sup>44</sup>Hans-Jörg Rheinberger (1997), *Toward a History of Epistemic Things: Synthesizing Proteins in the Test Tube*, Palo Alto: Stanford University Press, ch. 5; the chapter is titled “Reproduction and Difference”, and to elucidate its relation to Deleuze’s dissertation “Difference and Reproduction” would be a very interesting future project.

<sup>45</sup>Rheinberger, ‘Man weiss nicht genau, was man nicht weiss: Über die Kunst, das Unbekannte zu erforschen’.

<sup>46</sup>Rheinberger, *Toward a History of Epistemic Things: Synthesizing Proteins in the Test Tube*, p. 24.

system. The title of my 2012 paper ‘Sound Similarity as Interface between Human and Machine in Electroacoustic Composition’ was chosen to highlight this “extimacy”. We must use a machinery such as signal feature extraction and correlation repeatedly in an experimental setting, thereby getting hold of it, developing this skill to handle it. But we must be careful to transgress its prescribed purpose and control structures.<sup>47</sup> In the second part of *Leere Null*, the algorithms played out their intrinsic capacities much better—in my judgement—than in *Writing Machine*, because while my *concept* of the application of sound similarity was quite clear in both cases, only in the former case had my skills of preparing the experimental system been developed enough to bring out an unforeseen interplay of variables.

Instead of interface we could also say indirection. The intrinsic capacities of the machinery—i.e. not imposed by the programmer’s mind—present the composer with a situation which he could not have established without the indirection of involving the computer algorithm; without the de-coupling from pre-meditated intentions. Interface and indirection mean a crucial deferral or suspension. The interface is the thin line which Varela describes as the “dance” between autonomy as law-from-the-inside versus control as law-from-the-outside.<sup>48</sup> A completely controlled object is the technical object, defined by rigidity and specificity: a software, an algorithm must be defined, it must be correctly stated in the object language and well formed. On the other hand, an epistemic thing is characterised by an inherent blurriness. Rheinberger calls it the «whole commitment» («der ganze Einsatz»<sup>49</sup>) involved in a research project, it embodies that what cannot yet be denoted. The rigidity of the technical objects as «stable subroutines»<sup>50</sup> is responsible for keeping the vagueness of the epistemic things “hypocritical”. In other words, the algorithms provide guidance in the epistemic exploration and artistic research, they act as facilitators, they give coherence so that a previous experiment can be connected to the successive one.

<sup>47</sup> See the discussion of goal-directness in Sect. 3.1.2.

<sup>48</sup> Francisco J. Varela (1981), ‘Autonomy and Autopoiesis’, in: *Self-organizing Systems: An Interdisciplinary Approach*, ed. by Gerhard Roth and Helmut Schwegler, Frankfurt and New York: Campus Verlag, pp. 14–23.

<sup>49</sup> Hans-Jörg Rheinberger (2nd July 2008), ‘Epistemische Dinge—Technische Dinge’, *Bochumer Kolloquium Medienwissenschaft*, URL: <http://vimeo.com/2351486> (visited on 28/08/2012), 13’.

<sup>50</sup> Rheinberger, *Toward a History of Epistemic Things: Synthesizing Proteins in the Test Tube*, p. 80.

The experimental system is productive when in motion, meaning that technical objects and epistemic things are transformed into each other. A formerly experimental algorithm—which perhaps yielded unexpected results—can be used as a known function in a new experiment or piece. On the other hand, the focus might now be turned to one particular aspect of the algorithm which had previously been unimportant, promoting it to epistemic rank. The motion in between the two is the differential reproduction, and I have tried to show how the perspective has shifted in the creation of the presented pieces without implying a cause and effect, but rather exemplifying how each piece gained its contours through the act of differentiation from the others. Again, similarity is interesting not inasmuch as an imitation is evaluated with respect to identity (cf. Sect. 3.2.1), but in the way it diverges from the template.

## 4.5 Exploiting Graphemes

Taking the preceding thoughts into consideration, the main motivation behind the development of my own computer composition system becomes clear. Tracing the process of composing a piece is not primarily intended as an objectification of what the composer does with the aim of triangulating this data in a (musicological, cognitive, . . .) analysis, the “loop” in the upper half of Fig. 4.35. That is not to say that our system is not *useful* in that respect—and I hope to show with the plots in Sect. 4.6 that it is highly useful. But more than adding another layer to the toolbox for observing a composer by an external researcher, the idea is to enable the material trace of the process, the grapheme, to become something the composer can recursively work with, corresponding to the lower half of Fig. 4.35.

The recursive operations on the grapheme can take many different forms. In the remainder of this section I will give a few examples, as well as show how the analysis can benefit from the system’s database.

### 4.5.1 Dots

The first example involves multiple works which were shown together in an exhibition titled *Writing Machines*.<sup>51</sup> One of them is the room and sound installation *Voice Trap*, a collaboration

---

<sup>51</sup>Gallery ESC im Labor, Graz, October–November 2012.

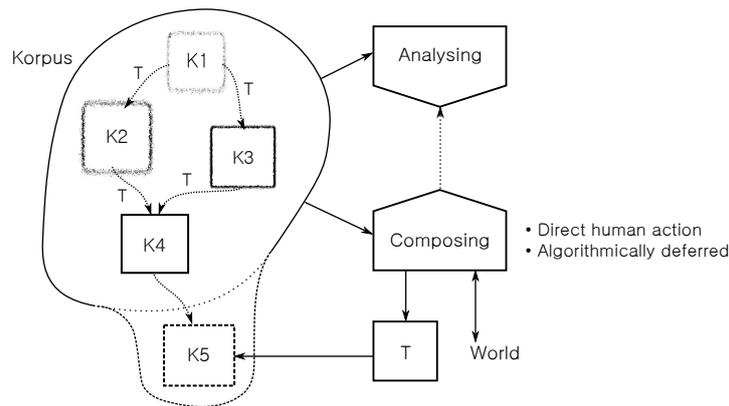


Figure 4.35: Two possibilities of exploring a grapheme. The corpus, shown as the ensemble of snapshots  $K_1, K_2, \dots$ , dynamically grows through the incorporation of elements from the “world” through transactions  $T$  carried out by the human composer or algorithm. This material trace both informs the composition process as well as an independent analysis which draws conclusions about the former.

between me and Nayarí Castillo. Nayarí has been working with story fragments which combine factual and fictional elements, typically shown as a combination of written text and collected objects. *Voice Trap*, shown in Fig. 4.36, is spun around the story of a girl who is haunted by voices. The story is written across four large mirrors on the floor of the room. Inspired by some strange bottles we saw in the Museum of Witchcraft in Boscastle, apparently used to trap ghosts, large jars filled with different materials are placed on the mirrors. The jars are tagged with the written description of a particular voice and their contents relate to the sound qualities of the imaginary voices. The sound installation is diffused from 96 piezo speakers grouped into twelve channels which are placed on a metal grid suspended below the ceiling. They are covered by wax paper discs to produce an amplification and a slight characteristic distortion of the sound. The sound composition runs in real-time from a computer, using for the first time the newly developed composition system which is detailed in Chap. 5.

The algorithm is a variation of the one used in *Writing Machine* (Sect. 4.4.4): A live signal, this time taken from a microphone picking up the noises from the street in front of the gallery, is fed into a database from which individual phrases are constructed. Instead of iterating over the channels, the twelve groups are using individual transactions. When searching for new sound fragments for a group in the database, a mixture between the currently heard phrase of the group and a hidden file is used to guide the similarity matches. These hidden files contain

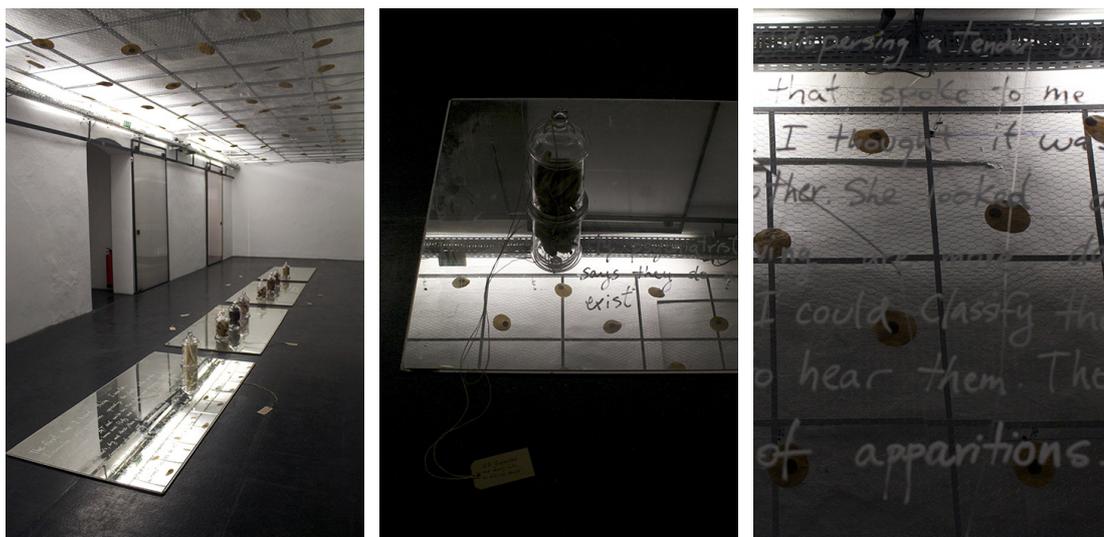


Figure 4.36: Wide shot and details of *Voice Trap*

different voice recordings with the hypothesis that from the outside sounds those fragments will be preferred which contain speech.

Since the evolution of each group/channel is captured by the persistent framework, the algorithm can make references to this evolution. This installation was more conceived as a general validation of the framework which was still at a young stage, so a rather rudimentary aspect was chosen—branching in the transactional time. For this, a concept of keeping track of the current movement in the version graph had to be found. A position would be a path from the root of the graph to the current version. It appeared paradoxical to define how these positions themselves would be traced, a requirement given by the fact that it should be possible to stop and relaunch the installation so that it picks up exactly where it left.

Prior to the exhibition I had been staying in Graz as an artist-in-residence at “Atelier RONDO”, working both on the pieces and the software, and one day came up with the idea of defining these “cursors” in terms of regular variables containing the current paths,  $S\#Var[S\#Acc]$ . In order to access and update these paths, a “master cursor” must be provided as an a priori of the system.<sup>52</sup> Fig. 4.37 shows an example version graph from the installation. The master cursor is not shown, just the twelve individual channel cursors. Each horizontal stretch is the succession

<sup>52</sup>In the later multitrack front-end *Mellite*, I simplified this idea by managing the cursors of a “document” using the ephemeral sub-system of the confluent document, i.e.  $D\#Var[S\#Acc]$  where  $D$  is the ephemeral system.

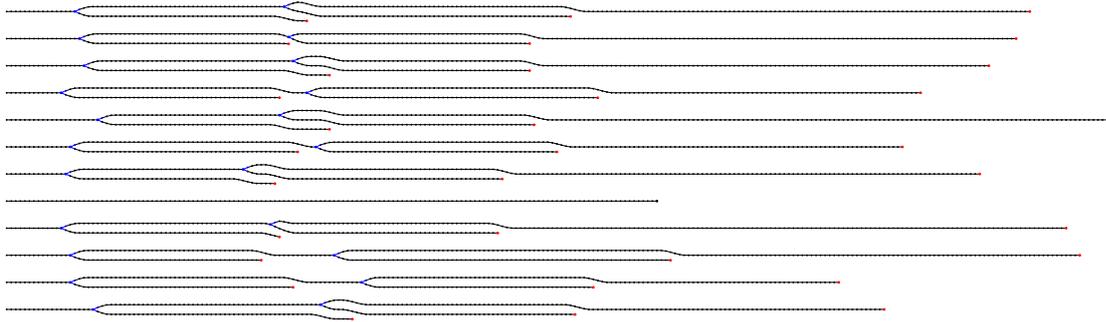


Figure 4.37: Excerpt from the version graph of *Voice Trap*. The transactions on the twelve individual cursors develop from left to right, amounting to c. 3700 individual steps and a time span of 40 minutes.

of transactions which produce a certain number of iterations over the sound phrase lengths, after which a jump into the “past” is performed, going halfway back between the current transaction and the last branching point.<sup>53</sup> So after a jump back in time, the sound phrase from that past moment in time is heard again, but the successive evolution (overwriting of fragments with new sounds) diverges from the previous path, because the sound database itself is ephemeral and not reverted to a previous state in the jump back.

Although the jumping back is highlighted by playing a particular sound when it occurs, it is difficult to perceive these jumps otherwise as repetitions, perhaps due to the locality of the jump—no synchronisation between the channels—or due to the fact that these specific environmental sounds are more difficult to distinguish than traditional musical gestures made from pitches. In that respect, although the experience of the sound space above one’s head is very enjoyable, the conceptual part of the sound installation remains intangible.

On the other hand, this piece was an important step for me. First of all, it demonstrated that the framework is functional and can handle a continuously growing database even after tens of thousands of transactions and several hundred megabytes file size. Secondly, the process of writing the framework and the sheer amount of work that went into this writing formed the basis for another work in the exhibition, *Dots*, shown in Fig. 4.38. This piece consists of fan-fold

<sup>53</sup>There are no branches visible in channel 8. This might have been a bug that caused that group to end unexpectedly, or it is a mistake in reading the version tree in the ex post analysis which took place a year after the database traces were recorded. Also note that although it seems as if no more branching occurs in the last third of the transactions, this is just a coincidence, and if the plot is extended to longer periods, branching continues at the same initial rate.

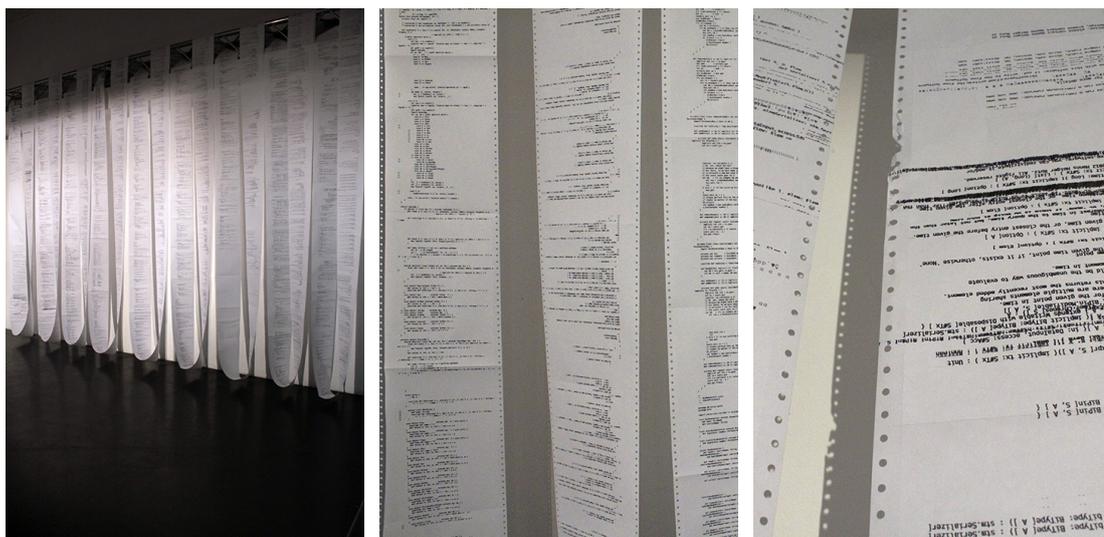


Figure 4.38: Paper installation *Dots*

paper with a print out of the framework’s source code. I found an old dot matrix printer which was used to produce the print.

I wanted to show the pro-gram, the writing before the writing, and this worked surprisingly well due to two distinct effects or visual qualities. The first quality is the appearance of varied rhythmic structures (middle photo in Fig. 4.38). Through the translation into a visual installation, this aspect which is neither apparent nor “functional” when writing the source code obtains an aesthetic quality of its own. It vibrates with a poetic quality which arises from the fact that the writing process which precedes the sound installation *Voice Trap* peels off from the latter. Somewhere in these rhythmic patterns lies the code that is sounding the next room, while at the same time these patterns also reflect the entirety of future sound pieces not yet written, their blueprint.

The second quality is the appearance of the particular material process of the *translation* itself. It was not possible to find the driver for the exact printer model, and I only managed to operate the printer from a very old PC which still had the necessary hardware interface. Especially in the beginning of the printing process, the printer exhibited all sorts of uncontrolled behaviour, such as at times printing random symbols, repeating lines, forgetting line feed so that multiple text lines were superimposed, or suddenly feeding an almost empty page. Some of these artefacts are

seen on the right hand side of Fig. 4.38. The process of using this particular printing technique overlays the rhythmic patterns of the code with its own rhythm of silence (gaps) and blackness (erroneous line feeds) caused by the glitches of the printer driver. It also adds another sound layer, the reminiscence of the needles hitting the paper. It was almost sad that the longer the printing process took (many hours), the less these artefacts appeared, as if the interface between PC and printer cleaned itself.

This independent aesthetic quality of the writing or translating process was echoed in another work of the exhibition, *Unvorhergesehen–Real–Farblos* (unforeseen, real, achromatic). It is a technically simple sound piece with three headphones mounted above a couch, each accompanied by a computer print of  $32 \times 32$ cm in rose and purple tones (Fig. 4.39<sup>54</sup>). On each headphone you can hear a different sound diary entry which was recorded during my stay at RONDO. The recordings were taken “live” so that I would improvise around a core idea—unforeseen, real, achromatic—connected to my compositional process. That is to say, I made an analogue translation between  $\mathcal{T}_K$  and  $\mathcal{T}_P$ . For the computer prints I wrote a program which produces a self-similarity matrix of the sound file of each recording. Time elapses from the bottom left to the top right, and each pixel found by horizontally or vertically moving away from the diagonal is coloured according to the similarity of those two moments in time. As in *Dots*, two types of rhythms interfere with each other: The rhythm between the soliloquies and the background noises, since the recordings were made in front of an open window to the city and the voice being suspended by many pauses, and the rhythm of the autocorrelation matrix which individuates the three prints.

These translations have always taken place. For example, in *Amplifikation* I had originally planned to print handwritten diaries onto the glass plates, but ended up translating them in the opposite direction: the texts were digitally scanned and interpreted as waveforms, producing a very characteristic rhythmical pattern. What is new, especially in *Dots* but also more subtly in *Unvorhergesehen. . .*, is that the compositional process itself is what is being translated.

---

<sup>54</sup>The chosen photos are from a successive viewing at “RONDO–Framed”, Graz 12/2012, because I prefer this piece to be shown in daylight.

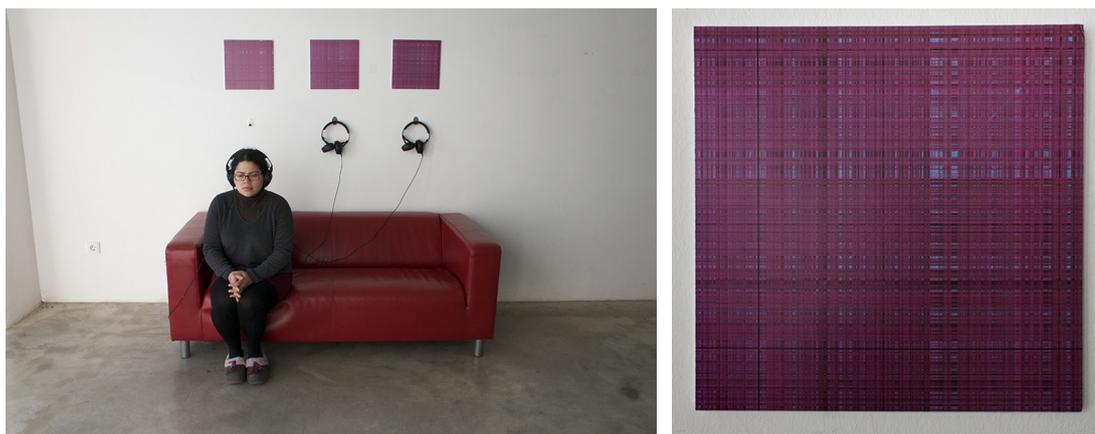


Figure 4.39: Sound piece *Unvorhergesehen–Real–Farblos*

#### 4.6 Indeterminus

Although I was able to run the framework in *Voice Trap*, there was not any specific development environment for it which allowed the composition of the algorithms to be employed in a traceable way. Speaking in terms of Fig. 3.7, I was still faced with the observation being limited to a specific domain inside the object language in which the installation was written (Scala). What was easily captured were the traces the algorithm produced inside the observed domain once it was written in the object language. Constructing a whole meta language was an effort undoubtedly beyond the scope of this thesis, so another path had to be taken to validate the approach in a more constrained setting.

While it was never my intention to specifically touch on the topic of graphical user interfaces in this thesis, a GUI is an excellent vehicle for such a constrained environment, as it is straightforward to encode the repertoire of actions within the observed domain. I had been offered a studio residency at the ZKM in Karlsruhe in May 2013 to work on an electroacoustic experiment, and so I defined this experiment in terms of another validation run for the framework. The working title *(Inde)terminus* refers to Gottfried Michael Koenig's tape piece *Terminus I* from 1961 which is based on a scheme for deriving sounds from previous sounds by applying a set of transformations. In his description of the piece's form I found a lot of connecting points to my work:

«... each derivation (such as filtering, modulation, chopping up or adding reverberation) was used as the source material for one of the successive derivations ... There are no prespecified relationships between sounds, but they emerge in the moment in which the derived materials are presented next to each other—or simultaneously. Thus, the problem of form appears in a very mediated way; the possible form parts ... are closely tied together based on the history of their production, yet devoid of a teleological relationship between them.»<sup>55</sup>

The idea to iteratively take the previous output and subject it to a defined process in the next step is also the foundation for the study titled *(Inde)terminus. Mellite*, a graphical front-end for *SoundProcesses*, is at first sight very similar to my previous tape composition software *Kontur*, however now using the persistent database to store the trace of actions performed. An example screenshot from *(Inde)terminus* is shown in Fig. 4.40. On the left side, a timeline view can be seen with several audio file regions placed on the canvas. The tool palette shows the possible operations: selecting, moving, resizing, adjusting gain, adjusting fade curves, and muting or un-muting a region. To add a region, a selection is made in an audio file view (shown behind other windows on the right hand side) and dragged and dropped onto the canvas. Cutting and deleting regions is possible through additional keyboard shortcuts.

At the front on the right-hand side, the “elements” window is shown. This is a generic tree structure in which objects can be created and organised. The opened popup menu shows the types of elements supported: folders, process groups (timelines), artefact stores (hard-disk locations), audio files, text strings, integer and decimal numbers, and code fragments.

The code fragment elements were an essential part of the experiment. It begins with an initial hand-constructed canvas of three minutes duration, sparsely placing sounds on an 8-channel layout. In the next step a bounce is carried out which is fed through a signal processing stage, to become blueprint for the next iteration. Here, a new canvas is built around this blueprint, possibly cutting it up, removing some parts of it and adding new sounds. Then again a bounce

<sup>55</sup>Gottfried Michael Koenig (1986/1993a), ‘Genesis der Form unter technischen Bedingungen’, in: *Ästhetische Praxis*, vol. 3, Texte zur Musik, Saarbrücken: PFAU Verlag, pp. 277–288; my translation

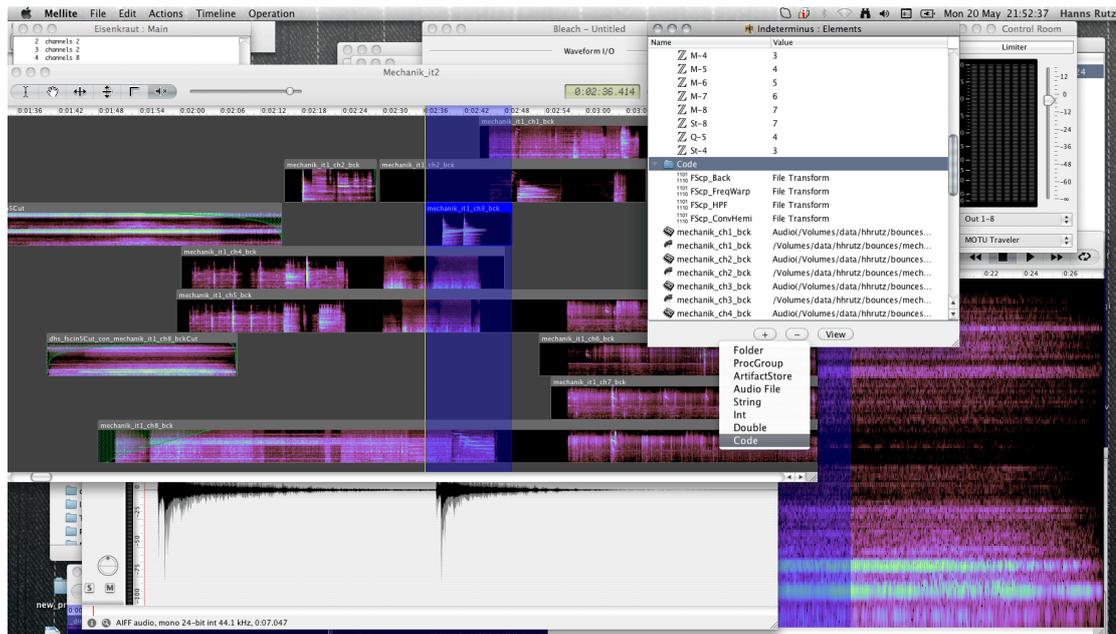


Figure 4.40: Screenshot of the *(Inde)terminus* session in *Mellite*

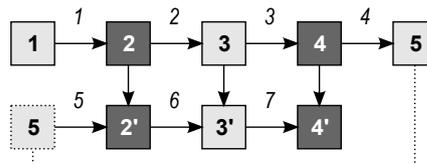


Figure 4.41: Iteration and recursion scheme of *(Inde)terminus*. The edge labels indicate the sequence in  $\mathcal{T}_K$ . Even and odd iterations share some similarity due to double reversals in  $\mathcal{T}_{(P)}$ . The fifth iteration replaces the first iteration in the first recursion—second row—which *rewrites* iterations 2 to 4.

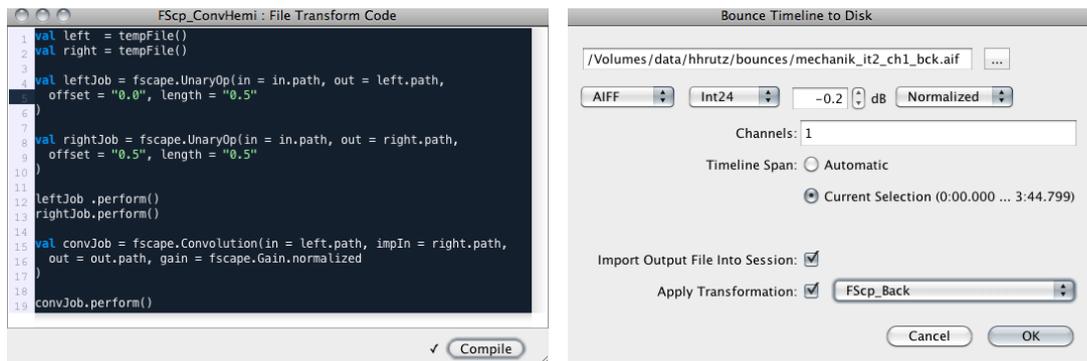
and a transformation is carried out, and so forth. The duration of each version grew slightly, leading to a 4 1/2-minute stretch in the fourth iteration. This is illustrated in the top part of Fig. 4.41.

The transformation is specified in terms of an *FScope* module which is invoked from *Mellite*. The module used for the inter-iteration step is *STEPBACK* which performs an automatic segmentation of a file and reverses the ordering of these segments. The code in the *Mellite* session is very short and looks as follows:

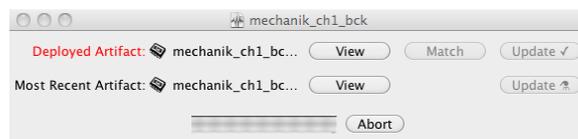
```

val job = fscope.StepBack(
  in = in.path, out = out.path, corrStep = 32, maxSpacing = "8.0s"
)
job.perform()

```

(a) Code fragment editor with *FScAPE* transformations

(b) Bounce dialog with transformation and re-import settings



(c) Recursive transformation dialog

Figure 4.42: Detail screenshots of *Mellite*

This is entered in an integrated code editor—shown in Fig. 4.42a—and compiled when needed using an integrated Scala compiler (“interpreter”). The code is wrapped in an auxiliary environment with bindings for the file input `in` and the transformed file output `out`. `job.perform()` launches an asynchronous rendering process which is reflected in the *Mellite* GUI using a progress bar dialog.

When bouncing, these code objects can be selected for application—shown in Fig. 4.42b—and when the output file is re-imported into the session, the *creation procedure* of this sound file is memorised as well. I bounced and transformed each channel separately, leading to different segmentations so that not just a diachronous reversal occurs, but also a synchronous scattering. The transformed bounces were placed on a new timeline, and cut again into chunks to remove the silent parts. I then reacted to the new temporal structure by adjusting it to my liking, possibly thinning out the material further or introducing new elements. Since the next iteration would again reverse the temporal succession, a specific similarity arises within the group of even-numbered iterations and within the group of odd-numbered iterations.

So again an interplay between me and the computer was established with a simple set of rules, this time keeping a trace of the actions I performed. The trace of the re-imported bounces allowed me to create a closed recursive setting, whereby after a certain number of iterations, the input to the initial bounce would be exchanged for the result of this last (fifth) iteration, *retroactively* re-triggering the bounce and transformation, so that I would have to re-work the iterations, theoretically ad infinitum, thus indeterminate. Practically, I carried out this re-working for the second (sixth), third (seventh) and fourth (eighth) iteration, as shown in the bottom row of Fig. 4.41.

The “flattening operation” of the bounce establishes the *crucial deferral or suspension* which I identified with the idea of interface in Sect. 4.4.7: I can manipulate a time canvas whose product was used in another canvas, and the propagation of the changes from the former to the latter is suspended. Furthermore, the flattening bounce provides the closure of the material which again makes it possible to subject it to general transformations such as the *Fscape* modules (cf. Sect. 3.3.3).

The suspension is operationalised in the recursion dialog, shown in Fig. 4.42c. The “deployed artefact” is the currently imported bounce. A match operator allows the recreation of the bounce and successive transformation using the original canvas selection, however viewing it from the current point in  $\mathcal{T}_k$ . If the updated artefact is different from the deployed artefact, one may choose to replace the latter with the former. This works by updating an expression variable holding the deployed artefact value.<sup>56</sup>

Before employing the recursion in my experimental setup for this piece, I did some isolated experiments in which short recursion cycles were created within the same canvas—a particular selection in the canvas was bounced, transformed, re-imported to overlap with the selection, thus changing the bounce input. By repeating the recursion steps a few times, very interesting “echoing structures” were produced. There is an infinite number of possibilities for introducing

---

<sup>56</sup>The expression system is described in Sect. 5.9. Basically the audio regions contain a reference similar to an `Expr[S, Artifact]` and the exchange of the bounce artefact propagates to all regions which used an expression of that artefact.

shifts from iteration to iteration, using the performance time, using the spatial positions of the sounds, using cutting and rearrangement, etc.

Another interesting aspect of using *Mellite* in this project was to compare it with my usage of the ephemeral predecessor *Kontur*. The latter existed for longer and was more mature, for example it has a standard undo/redo procedure which is missing in *Mellite*. In the beginning, this resulted in me working quite cautiously on the canvas, as an accidentally moved region could not be instantly reverted. As time passed, I both enjoyed this “destructive” aspect of the new interface, but also became more confident that a serious accident could be undone by going back to a previous point in  $\mathcal{T}_K$ . The cursor interface is still in its infancy and such a step is still a bit tedious, as the current view needs to be closed and reopened. This is a point which should receive more focus in future versions of the software.

#### 4.6.1 Ex-Post Analysis

In the last section of this chapter, I will change observer position and look at some of the traces which can be found in the data left behind by the *(Inde)terminus* project, as well as a newer tape composition *Machinae Coelestis*. This second piece was produced with a slightly newer version of *Mellite*, however not in the context of artistic experimentation, but with the aim of creating an electroacoustic or soundscape composition to be played in the planetarium of Judenburg (Austria) during my stay there with another residency programme in summer 2013. The planetarium (Fig. 4.43) is located in the town tower of Judenburg with a rather small auditorium of 65 seats and a Zeiss ZKP4 projector, a beautiful piece of analogue optics. The projector is programmed via a special authoring system which provides its own timeline. Commands which can be placed on the timeline include choosing an observer location on the Earth and an astronomical date, controlling intensities of different types of lights, adding or removing elements such as the projection of planets, moon and sun, the Milky Way etc.

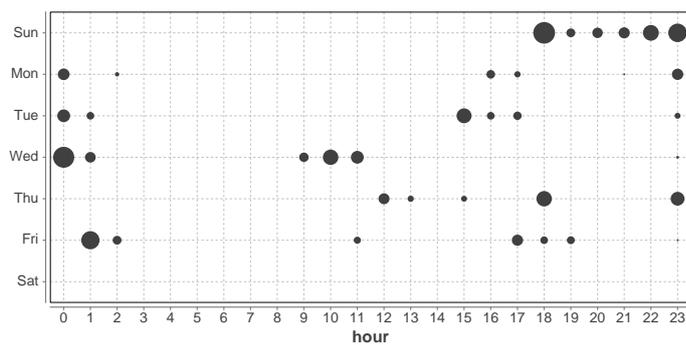
For this piece, lasting 16 minutes, I worked mainly with field recordings, but also processed and more abstract sounds and atmospheres. The field recordings connect to the place of Judenburg and various other sites I have visited. A subtle narrative is created by integrating two occurrences of spoken text. One could produce a timeline of sound registrations, as had been done for



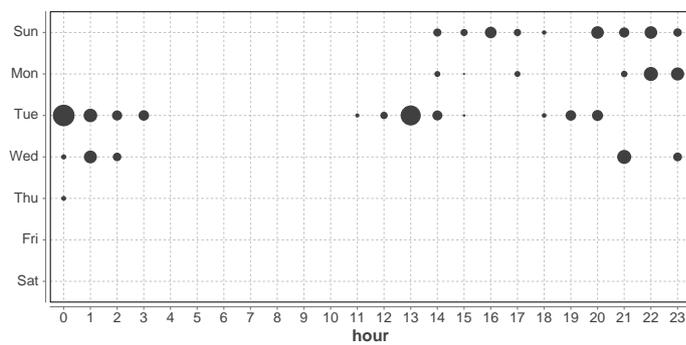
Figure 4.43: Planetarium Sternenturm Judenburg

*Dissemination* (Fig. 4.11), and one would find again that it is difficult to delimit the “beginning” of the composition. In fact, the time and place of some of the recordings are reflected by the visual composition, whereby the projector would move the sky to that time and place. I want to focus on the specific time window, however, in which the 16 minute arrangement has been done. A playful illustration of my *work shifts* is Fig. 4.44a, a “punch card” plot similar to the ones given by popular open source platform GitHub, showing at which times of the week someone has worked on a piece of software. Fig. 4.44b has the same plot for *(Inde)terminus*. While composing is hardly a regulated office job, plots like these, especially when more data is available, could reveal different profiles of composers or it could be used to highlight different types of activities.

Returning to the audio recordings used, Fig. 4.45 gives an overview when particular sound files have been added to the session. The first two sounds were “DreamSternenkarte. . .”—one of the two spoken text pieces, the recollection of a dream in which some sort of star chart appears—and “OrdingBuhne. . .”—a very long field recording from a walk across an icy wooden groyne by the North Sea. They appear towards the end and in the middle of the piece, respectively. A recording of sheep wearing bells, “Sheep. . .”, is added the following day. They make up the initial scene of the piece. The cryptically named “Drm13'Ich. . .” is already a heavily transformed sound file,



(a) *Machinae Coelestis*



(b) *(Inde)terminus*

Figure 4.44: “Punch cards” of weekly times at which work on the compositions was done

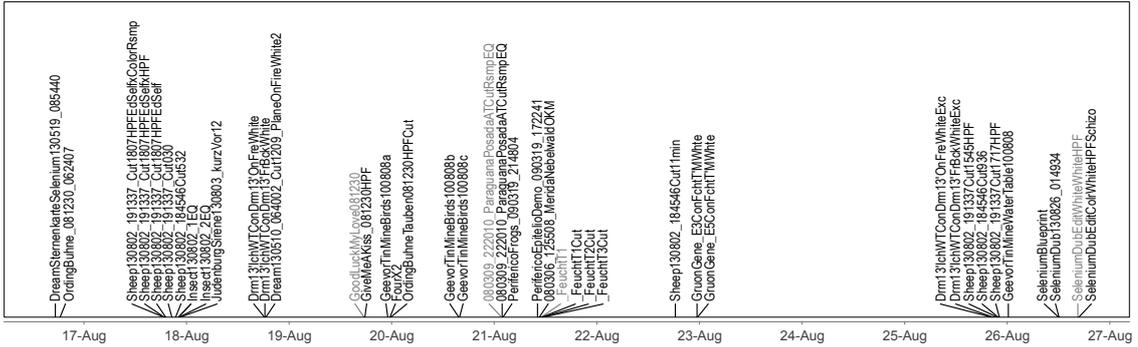


Figure 4.45: Sequence in  $\mathcal{T}_K$  showing when sound files were introduced to *Machinae Coelestis*

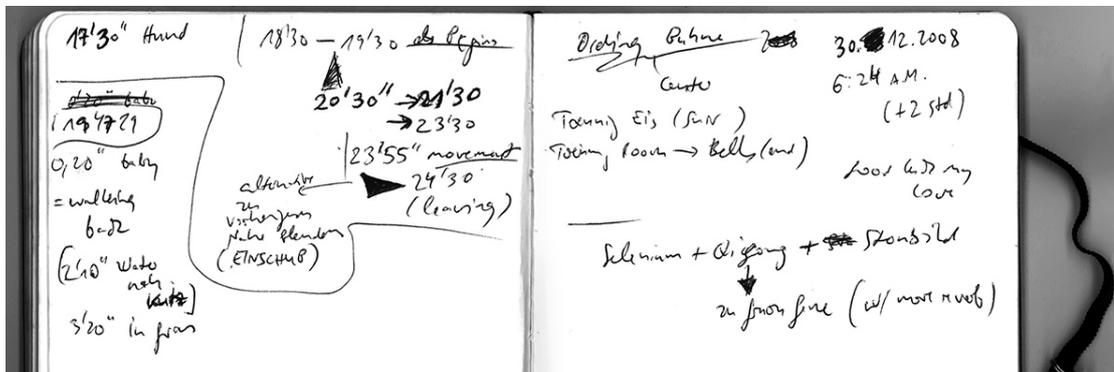


Figure 4.46: Notes regarding specific parts of the field recordings used. The times on the left side refer to the sheep recordings. On the right side, some sounds appear which were not used at all, such as “Toening Eis” which was captured nearby the “Ording” recording. 30.12.2008 6:24 am refers to the time of the Ording recording and was used as one date for the sky projection.

based on the speech fragment, used as an atmospheric drone in the second scene of the piece. In the evening of 19 August the file “GoodLuckMyLove...” was added, tested, and removed again (shown in grey) in favour of a similar file “GiveMeAKiss...”. And so the appearance of the material can be retraced.

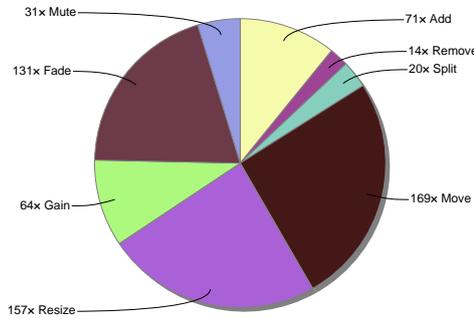
On the other hand, the major work of recording the sounds is unobserved; only the dates included in the file names give an indication. Also for the long files, there are additional sketches on paper which review the different elements contained in the recordings, an example of which is shown in Fig. 4.46.

## Tool Usage

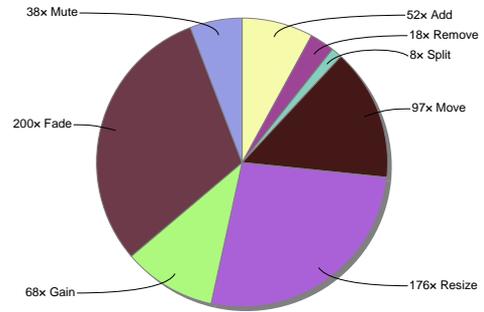
What happens after a sound has been imported to a session? The available tools have been enumerated in the previous section. One possibility is to look at their relative use. This was done in Fig. 4.47. Extracting the specific actions from the database was laborious, because the transactions were not specifically tagged by the software, so they needed to be reconstructed by analysing the structural differences between each two successive points in  $\mathcal{T}_k$ . Future versions should provide a more straightforward tracking mechanism, especially for collections, where for example finding out when elements were added or removed from the canvas requires iteration over the whole data structure for each possible version step.

In order to see if and how the proportions change over time, two charts were generated by splitting the set of transactions into two equal sized parts. In the first chart (Fig. 4.47a) which shows the earlier transactions, half of them deal with moving or resizing audio regions, while adding regions only accounts for 11%, removing regions for as little as 2%, and splitting regions into two for 3% of the transactions. When looking at the second chart (Fig. 4.47b) comprised of the later transactions, a few changes can be seen: The number of additions and splittings goes slightly down, while the number of removals slightly increases. The proportion between moving and resizing shifted from 1:1 to 1:2. A tentative and plausible explanation is that as the work on the piece progresses, less new material needs to be added, and more material has found “its spots” on the canvas, giving priority to adjusting existing material in place. Also the adjustment of the fading in and out curves becomes much more prominent, rising from 20% to 30% between the first and second half of the compositional work. Adjusting the volume of regions occupies 10% in both halves. A more complex composition with a larger number of transactions may allow for a finer grained resolution, perhaps looking at more than two time spans to see how the usage of a certain action is distributed.

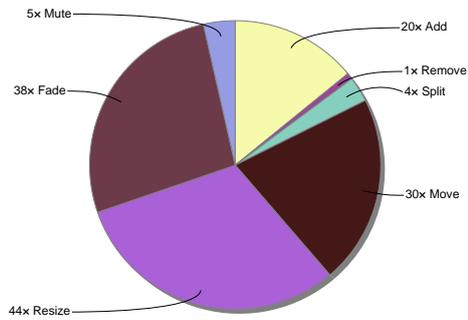
The remaining charts of Fig. 4.47 show the distribution among different iterations of (*Inde*)*terminus*. Fig. 4.47c covers only the first iteration or box 1 in Fig. 4.41. Fig. 4.47d covers the iterations 2 to 5 before entering the recursion, items 2 to 5 on the first row of Fig. 4.41. Finally, Fig. 4.47e covers the re-workings done after replacing the original input in the recursion, the



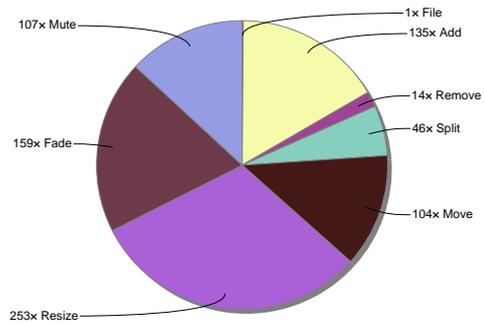
(a) *Machinae Coelestis* 1<sup>st</sup> half



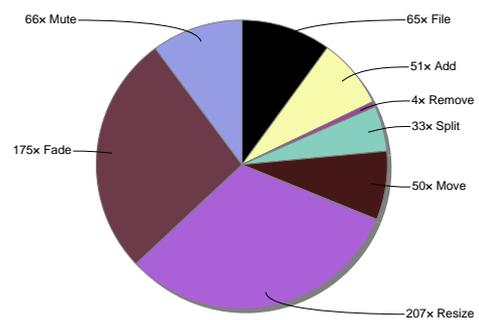
(b) *Machinae Coelestis* 2<sup>nd</sup> half



(c) *(Inde)terminus* 1<sup>st</sup> iteration



(d) *(Inde)terminus* 2.-5. iteration



(e) *(Inde)terminus* recursion of 2.-4. iteration

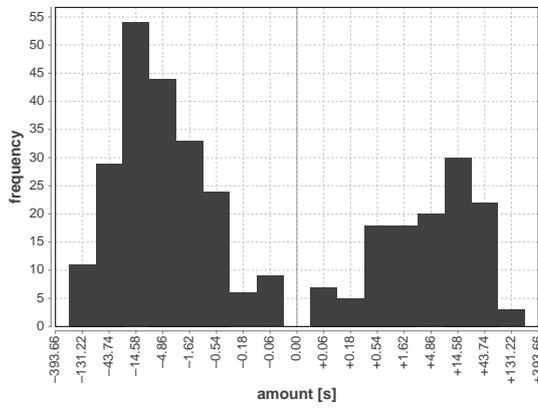
Figure 4.47: Frequencies of *Mellite* tool actions

second row of Fig. 4.41. The gain tool is not shown, because in the old version of *Mellite*, gain changes were differently encoded and we did not bother to write a special extraction algorithm to recover that information.

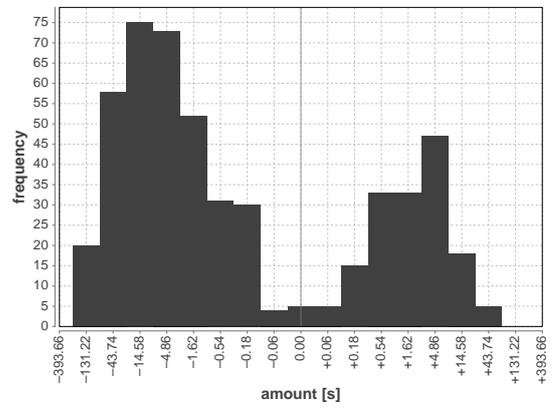
In general, the pattern is not unlike the one exhibited by *Machinae Coelestis*. The amount of additions is larger, as the study used a more liberal approach to introducing materials. Since the idea was to let the signal processing transformation create a temporally reversed structure by itself and then to accept that structure as a basis of each new iteration, naturally there are fewer movements of regions in  $\mathcal{T}_{(P)}$ —apart from the initial iteration of course. Instead, the number of splittings increases, since each iteration begins with the transformed bounces being cut up to remove silent parts. After the recursion begins, the black pie segment labelled “File” indicates the actions of replacing the previously deployed artefacts with the updated artefacts. In this last chart, we see again an increase in adjusting fade curves and a decrease in actually moving around regions. Notable in all but the first iteration is the relatively larger amount of mute and un-mute actions. This could reflect the better studio monitoring options and the larger number of independent channels compared to *Machinae Coelestis*. Some of the muting is also used as an alternative to removing regions, amounting to c. 25% of mute/unmute actions before the recursion and 50% after the recursion.

The next possibility is to look at the parametrisation within the groups of actions. Fig. 4.48 shows the distribution of varieties among the resize actions, and Fig. 4.49 looks at the region movements. The histogram bins use a logarithmic time scale and labels give the lower interval margin. Intervals of less than 60 milliseconds are grouped into the central bin. For the resize charts, the vertical line in the centre distinguishes contractions on the left and expansions on the right. For the motion charts, the centre line distinguishes movements backward in  $\mathcal{T}_{(P)}$  on the left and forward in  $\mathcal{T}_{(P)}$  on the right.

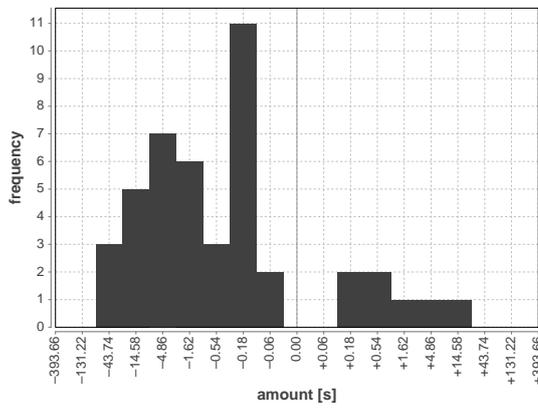
An interesting “left-leaning” tendency can be observed both in resizing and moving objects. Regions tend to be shortened rather than elongated, but also material seems to move backward in time more often than forward, perhaps due to an editing style which initially gives each region some isolated space before condensing the structure left-to-right. Also there seems to be an



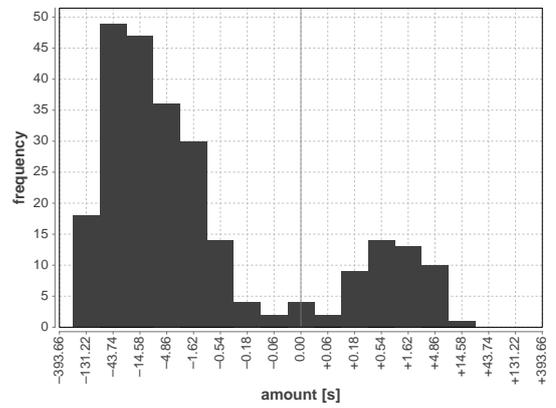
(a) *Machinae Coelestis*



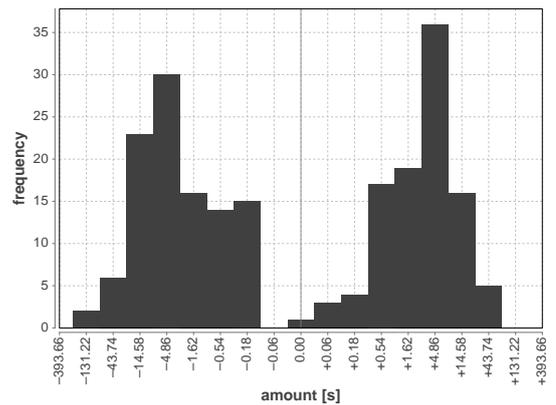
(b) *(Inde)terminus* (total)



(c) *(Inde)terminus* 1<sup>st</sup> iteration

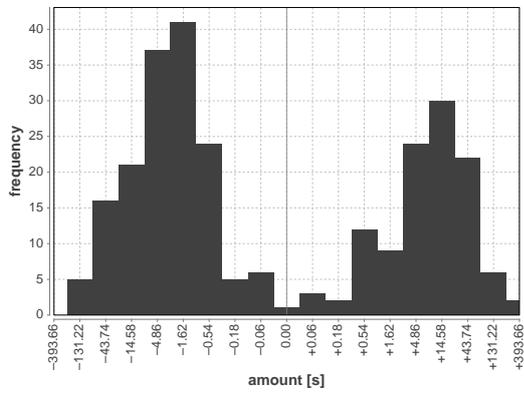


(d) *(Inde)terminus* 2.-5. iteration

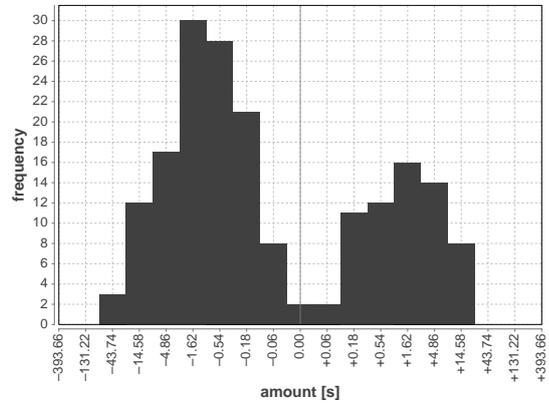


(e) *(Inde)terminus* recursion of 2.-4. iteration

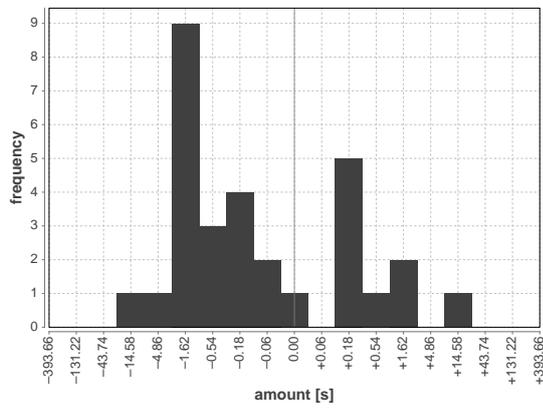
Figure 4.48: Distribution of the amount of contraction and expansion in resize actions



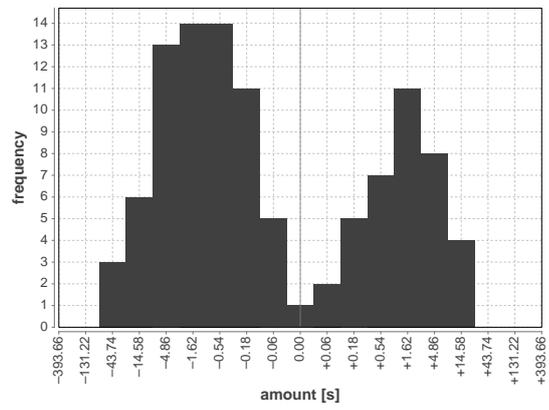
(a) *Machinae Coelestis*



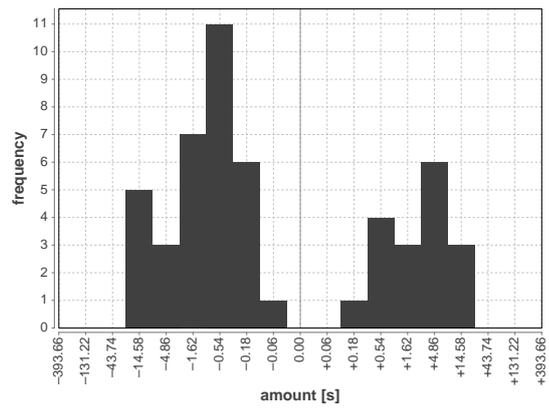
(b) *(Inde)terminus* (total)



(c) *(Inde)terminus* 1<sup>st</sup> iteration



(d) *(Inde)terminus* 2.-5. iteration



(e) *(Inde)terminus* recursion of 2.-4. iteration

Figure 4.49: Distribution of the relative time shift in move actions

overall bell shape in the distribution of both action types, which may be inherent to the type of sound material used or dominated by the typical zoom levels used in the graphical interface.

Looking at the total amount of transactions, the resize behaviour is similar between the two music pieces, whereas larger movements are carried out in *Machinae Coelestis*. This might be due to *(Inde)terminus* having a shorter total duration, but also due to the approach of overwriting existing structures. Other explanations would be a stronger perceived freedom of manipulation or more experience with the tool in the later piece.

Within the different stages of *(Inde)terminus* some changes can also be seen. For example, the initial iteration shows only subtle resizing, while in the subsequent iterations some strong shortening occurs and only few expansions. After the recursion took place, contractions and expansions are quite balanced, although having pronounced magnitudes. In contrast, the pattern of moving regions around is not significantly different before and after the recursion.

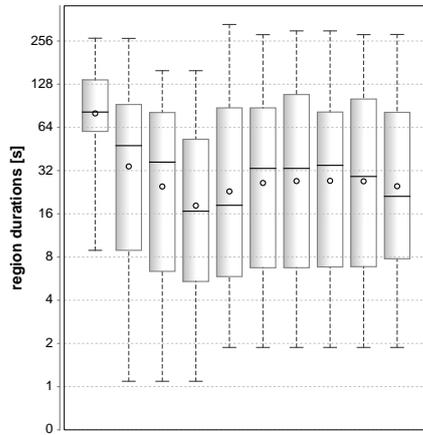
### **Evolution of the Temporal Form**

After looking at the action repertoire, we shall now return to the sound material itself, still in its characteristic manifestation as cut audio file regions, disregarding the “content” of the files. Fig. 4.50 shows the statistical moments of the regions’ durations. In each chart, we look at ten “snapshots” of the pieces, from the initial stages on the left towards the final stages on the right. For *(Inde)terminus*, a vertical line shows the transition to the recursive re-workings. The bars indicate the interquartile ranges of the durations with median shown as a line and mean shown as a circle. The whiskers indicate the 2nd percentile and the 98th percentile.

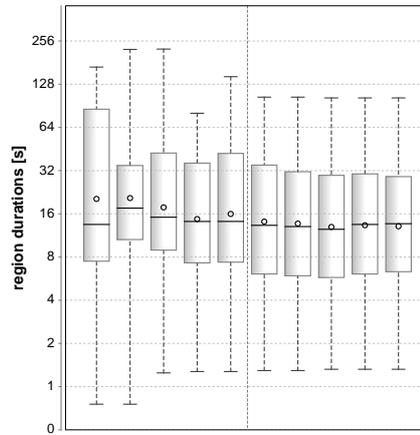
Fitting with the resize profiles, in *Machinae Coelestis* the durations seem to oscillate around a mean of 30 seconds, twice as much as in *(Inde)terminus*; however, towards the final stage, the median goes down to around 20 seconds. A value around 16 seconds is also approached in all individual iterations of *(Inde)terminus*.<sup>57</sup> Towards the end, very long regions of up to four minutes are still found in *Machinae Coelestis*, compared to less than two minutes for *(Inde)terminus*. In the former case, the overall variability is greater. In both cases, after around half of

---

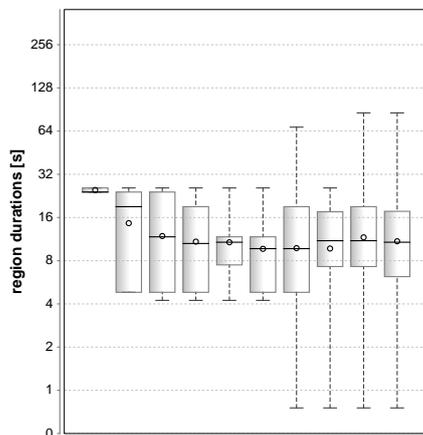
<sup>57</sup>The 4th iteration is not shown, as it looks like a mixture between the second and the third.



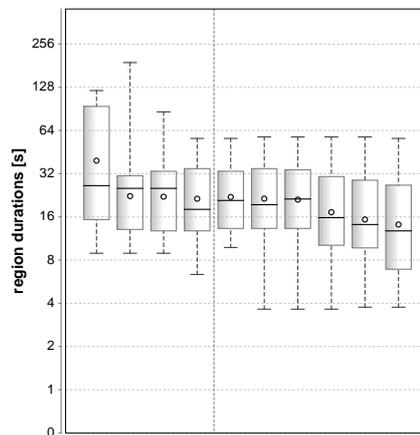
(a) *Machinae Coelestis*



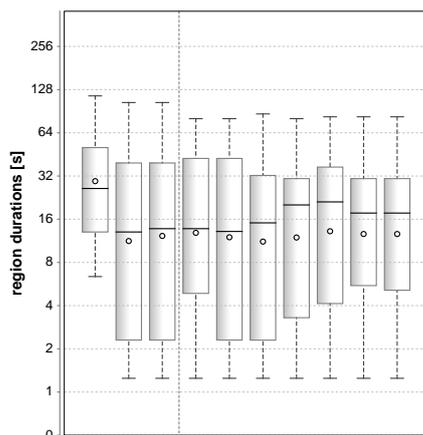
(b) *(Inde)terminus* (total)



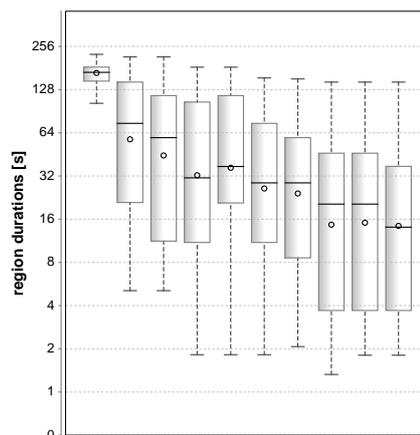
(c) *(Inde)terminus* 1<sup>st</sup> iteration



(d) *(Inde)terminus* 2<sup>nd</sup> iteration



(e) *(Inde)terminus* 3<sup>rd</sup> iteration



(f) *(Inde)terminus* 5<sup>th</sup> iteration

Figure 4.50: Evolution of the distribution of audio region durations

the transactions, the statistical values remain quite stable. For *(Inde)terminus*, this coincides with the post recursion re-working phase.

When looking at the different iterations of *(Inde)terminus*, the first iteration is characterised by a narrow variability in region durations in the beginning, which shrinks further as it evolves, and only in the last third do more extreme values arise. Naturally, by beginning with the long bounce files in the second iteration, the initial situation here is one of rather long regions. In the end, most regions last between 7 and 27 seconds. The third iteration is much more diverse, although the average duration remains around 16 seconds. Within the recursion re-work extreme values remain the same while the IQR decreases. The fifth iteration shows the longest regions, the extremes of which remain in place throughout the development, although the average in the end is again 16 seconds.

These are just some of the possible ways of extracting information from the database. One can compare composer with composer, piece with piece, sections within a piece, sections within the creational timeline; one might use such information to test or support hypotheses about the working process, the musical material or the human-computer interaction. The beauty of this approach lies in the fact that the situation is not a priori contaminated with “musical meaning” or “musical interrogation”, the short circuit of intention–product, but indeed accentuates motions which underlie the compositional process and which may otherwise remain tacit.

Finally, the suspension inherent in an ex post analysis (its temporal distance) may become again an interface for the production of new artworks, in a sense similar to the transcription processes found in *Amplifikation*—written text to scanned waveform—*Dots*—source code to spatial rhythm—or *Unvorhergesehen. . .*—visual interpretation of sound features. I can already discern a sensual quality in the box plots of Fig. 4.50, and it is not hard to imagine how they can be put to work by themselves, stripped from their empirical representational vestment. An even stronger instance of such transcription are the “motiongrams”, the final figure of this chapter, intentionally shown without axes or labels. They bring together  $\mathcal{T}_K$  and  $\mathcal{T}_{(P)}$ . Fig. 4.51 is drawn for *(Inde)terminus*. The blackening corresponds horizontally with the span in  $\mathcal{T}_{(P)}$  affected by an action at a given vertical point in  $\mathcal{T}_K$ .

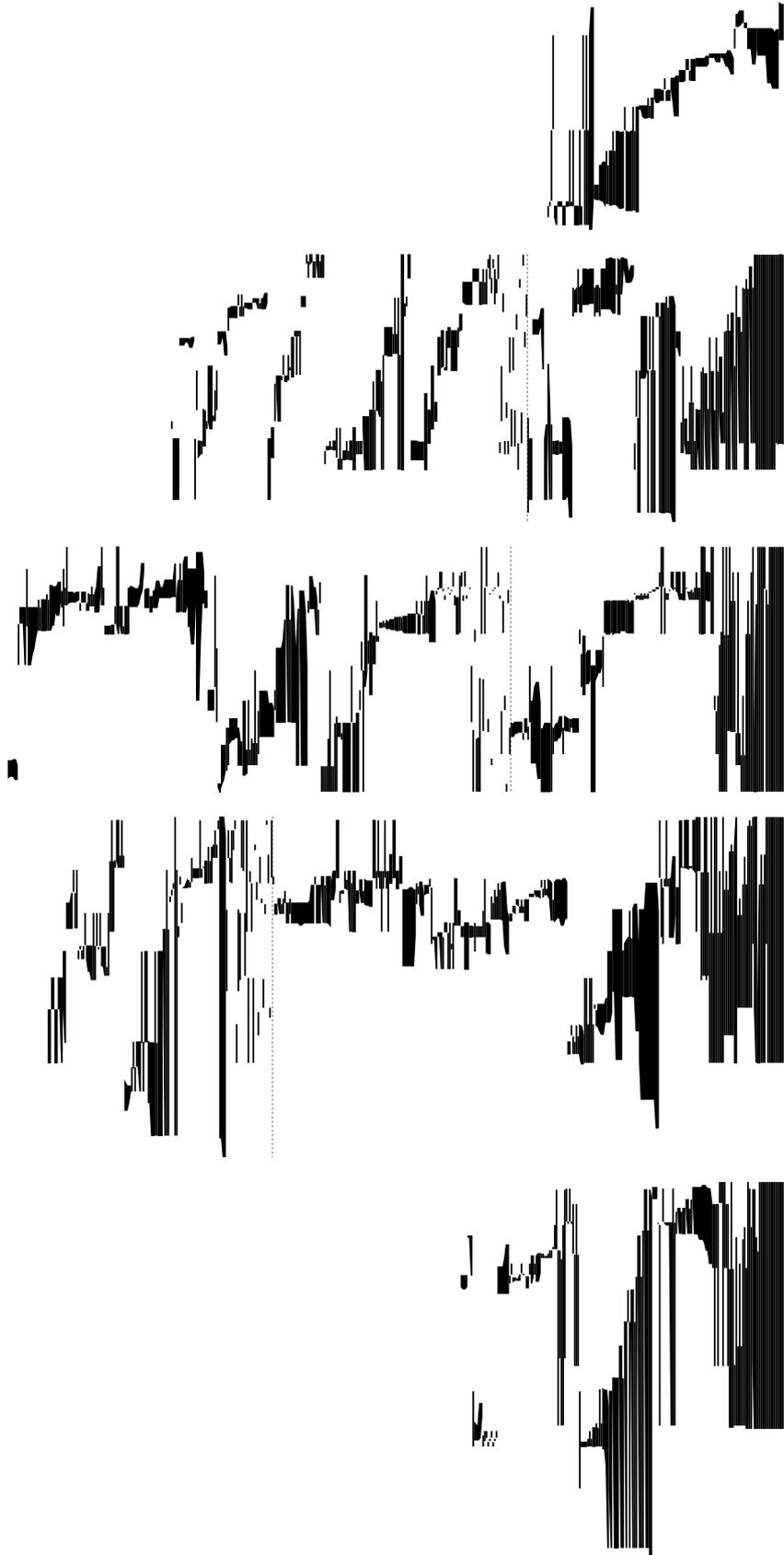


Figure 4.51: Motiongram for *(Inde)terminus*. The iterations are shown from left to right, transactions advancing from top to bottom. In each diagram, the horizontal extent covers the canvas duration of the particular iteration. Dotted lines indicate the beginning of the recursive re-workings. If an invisible grid is superimposed, the matrix of Fig. 4.41 can be seen.

One could interpret that diagram again. One would find the “carriage returns” in scanning through the timelines. One would discern the initial phase of each iteration from the subsequent refinement. One would see moments of obstinate distillation at a particular spot; see at which point in  $\mathcal{T}_K$  a certain part of the piece is more or less finished. . . . The interpretations in this section were deliberately kept vague, as I wanted to show the potential lying in the recorded data rather than focusing on a specific trait of these two pieces.

## 4.7 Summary

This chapter has presented three different ways to trace the compositional process within my work. There are pieces whose disposition is procedural, such as the *Residual* series or *Zeichnung*, but there is no explicit incorporation of the compositional time within the pieces themselves. This process can be retraced in a classical musicological analysis, for example by looking at the construction schemes as some sketches were preserved. But it can also be retraced on a different level, by taking specific concepts which are tangent to several pieces—the idea of a sound mobile, of foreground and background, etc.—and observing how their configuration changes.

The second set of pieces exhibits a stronger reflectivity on the creation time. For example, *Zelle 148* included a manifest anchor to this time by showing the rock and the scratch marks which produced the sound. Another strategy was the composition of writing processes which aimed at dissolving the diaphragm between composing and performing. For example, *Dissemination* maintained a pool of sound files evolving over the duration of the exhibition. *Leere Null* yielded autonomy to the machine as well, and although it was a composition fixed at the moment of its performance, it tried to play with this fact by exploiting the decoupling of  $\mathcal{T}_K$  and  $\mathcal{T}_P$ .

Finally, we were able to use a software framework to manifestly capture the decisions involved in a composition and inject these traces into the further development of the pieces themselves. Data was available from two pieces, *(Inde)terminus* and *Machinae Coelestis*, but only the former (which in fact was not a “piece” but a study) made use of a structural re-entry of that data by referring to the bounce of a timeline stretch and later recursively exchanging the referent.

It is important to remember that, as the first “stable” rendering of the framework stood nearly at the *end* of the thesis project, we have only just scratched the surface of its possibilities; the next chapter in turn will show how this framework came into being. It is equally important to understand that the machine-based tracing, while introducing a novel quality, does not replace the manual deconstructivist approach but complements it.

## Chapter 5

# Design and Implementation of a Tracing System

The purpose of this chapter is to discuss the development of a novel tracing framework for computer music. It is based on the idea of a common transactional system layer which encapsulates different possibilities of representing the creation time: Ephemeral in-memory, ephemeral durable, and confluent persistent and durable. We show how to understand a mutable entity and have a handle on its identity. We develop a set of data structures, such as deterministic skip lists and octrees, based on such mutable cells which can be persisted with a key-value store. They become the building blocks for the implementation of the confluent system itself. We exploit the version graph representation to add quasi-retroactive modifications, before moving on to a reactive event system and dataflow-like expression graphs to build scalable and interactive elements. With bi-temporal expressions we introduce the possibility of ascribing performance time to objects. We then define the basic unit of a sound process which is embodied by a synthesis graph function, an attribute map and a signal map to link different processes and maintain rendered artefacts. Finally, real-time sound synthesis is coupled via a model-view-controller separation.

### 5.1 The Programming Language

G. Loy and C. Abbott in 1985 distinguished three approaches to computer music programming. Either a program is written entirely in a general purpose language—as example Koenig’s *Project 1* was given—or it will be written in a general purpose language but with the help of specialised libraries for musical tasks—newer examples might be *Lisp* based systems *Common*

*Music* (symbolic composition)<sup>1</sup> or *Common Lisp Music* (sound synthesis)<sup>2</sup>—or a new domain specific language (DSL) is created as «embodiment of a musical paradigm».<sup>3</sup> Examples of the last category are *CSound*, *Pure Data*, and *SuperCollider*.

What is needed to embody musical paradigms that goes beyond the abstractions found in general purpose languages? James McCartney, in arguing for his language *SuperCollider*, said that it must be possible to express compositional and signal processing ideas as easily and directly as possible. These ideas can be very diverse according to the approach of the composer or the piece at hand, but what he imagined with *SuperCollider* was to «realize sound processes that were different every time they are played, to write pieces in a way that describes a range of possibilities rather than a fixed entity, and to facilitate live improvisation by a composer/performer.»<sup>4</sup> This is very similar to our own intention, namely to be able to capture the process character of music, to keep musical structures in flux, and to blur the boundary between composing and performing. It is thus no surprise that *SuperCollider* was used in preliminary sketches and remains the engine for real-time sound synthesis, albeit changing the object language in which structures are formulated for a general purpose language, *Scala*.

As we develop the data structures and conceptual abstractions of our computer music framework, the reader will encounter many small examples of *Scala* code. It is thus advisable to familiarise oneself with the basic elements of *Scala* and especially its sophisticated type system. The elements we consider important and which are used in this chapter are reviewed in Appendix B. *Scala* is a blend of object-oriented and functional programming concepts. Its name is a portmanteau of ‘scalable language’, which means that it aims to be a good choice both for small scale scripting purposes as well as building large modular code bases. It has an expressive syntax which makes it also good candidate for internal DSLs or language extensions. *Scala* runs on the *Java Virtual Machine* (JVM) which manages garbage collection, and the framework is

---

<sup>1</sup>Heinrich Taube (1991), ‘Common Music: A Music Composition Language in Common Lisp and CLOS’, *Computer Music Journal* **15**(2), pp. 21–32.

<sup>2</sup>Bill Schottstaedt (1994), ‘Machine Tongues XVII: CLM: Music V Meets Common Lisp’, *Computer Music Journal* **18**(2), pp. 30–37.

<sup>3</sup>Gareth Loy and Curtis Abbott (1985), ‘Programming Languages for Computer Music Synthesis, Performance, and Composition’, *ACM Computing Surveys (CSUR)* **17**(2), pp. 235–265.

<sup>4</sup>James McCartney (2002), ‘Rethinking the Computer Music Language: SuperCollider’, *Computer Music Journal* **26**(4), pp. 61–68.

platform independent and can use existing libraries and frameworks from the *Java* world. The language is statically typed and provides integration with major development environments such as *Eclipse* and *IntelliJ IDEA*.

The choice of *Scala* as the implementation language also allowed us to use it as a single language both for building the framework and for the code fragments a composer would eventually write with it. Other systems have used a split, for example *SuperCollider* itself is written in *C* and requires *C* for writing DSP unit generators and language primitives which require high performance, whereas user libraries are written in the dedicated *SuperCollider* language and runtime scripting is done in a subset of that language (among other things disallowing class definitions). Another example is the graphics composition software *Field*,<sup>5</sup> written in *Java* but with the user/composer writing code fragments in *Python*.

## 5.2 Framework Overview

This introductory section describes our computer music system in a compact form, functioning as a signpost into the subsequent sections which will then detail the development of the partial solutions and algorithms. The overview is again divided into a conceptual part, explaining the overall form of the system, and a part which breaks the form elements into technological pieces which constitute the framework's architecture.

### 5.2.1 Conceptual Summary

First, let us recollect from Sect. 2.4 that we are primarily interested in providing a trace of the creation time  $\mathcal{T}_K$ , whereas prospective performance time  $\mathcal{T}_{(P)}$  will be a *function* of the former. While not all materials produced by a composer need to be eventually projected in a heard musical time, the opposite holds—everything that happens in the process or the performance has been written in time.

Second, the number of musical paradigms is too large to be implemented in one monolithic system. Indeed, it is unbounded, and as B. Eaglestone et al. have observed, «a creativity support

---

<sup>5</sup>Marc Downie (2008), 'Field—a New Environment for Making Digital Art', *Computers in Entertainment (CIE)* 6(4), 54:1–54:34.

system should aggregate rather than integrate»,<sup>6</sup> meaning that it needs to be open in the sense of being accessible for unforeseen extension instead of trying to prescribe its future use, no matter how wide the boundaries are drawn. Naturally, a programming language provides this connectivity, where a finite set of provided abstractions can be used to create an unbounded amount of programs. For the system to scale—grow with the needs of the user—it must be modular. For example, the user may decide facultatively whether the observation of a particular musical structure should be memorising or oblivious, therefore the graphematicity of  $\mathcal{T}_K$  must be adaptable.

Third, being open to multiple musical paradigms does not mean that we have no responsibility to provide support for these. The symbolic nature of programming languages naturally produces a bias towards supporting symbolically represented structures, as can be observed by the amount of software based on computer-aided composition as a formal process abstracted from sound production and sound processing. To counteract this bias, we will explicitly devise abstractions for the handling of electro-acoustic material and real-time sound synthesis, even though those abstractions form leaves in our architecture dependency tree and may thus be replaced for other abstractions.

What a classification of programming paradigms such as declarative, procedural, functional, object-oriented etc.<sup>7</sup> omits is that they are just at the surface of a system which is fundamentally delimited by its choice of data structures. Before it is even decided which musical relations are represented, these structures preconfigure the *mechanisms* by which musical objects are instantiated and manipulated, in other words how the *process of composition itself* is enabled.

The design of the system then naturally evolves bottom-up, providing a common ground on which the temporal trajectories of the process can be preserved. It begins by defining an abstract model of transactions and how these may be preserved on a hard disk. It will then instantiate this model with ephemeral and persistent variants of  $\mathcal{T}_K$ 's trajectory, then proceed to define interactions of elements, and finally—only after all this—suggest a way to represent

<sup>6</sup>Barry Eaglestone et al. (2007), 'Information systems and creativity: an empirical study', *Journal of Documentation* **63**(4), pp. 443–464.

<sup>7</sup>cf. Henkjan Honing (1993), 'Issues on the representation of time and structure in music', *Contemporary Music Review* **9**(1), pp. 221–238

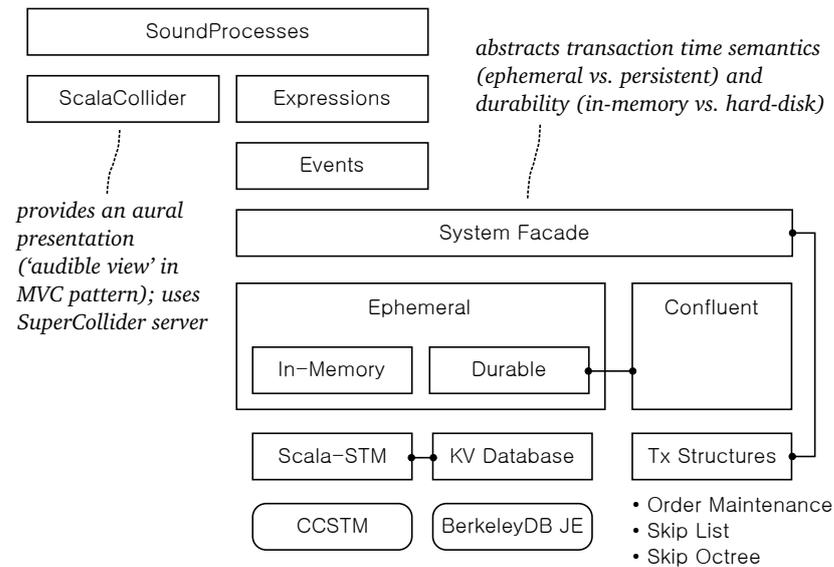


Figure 5.1: Architectural diagram of the framework

performative time and how to embody performative temporal evolution as a real-time sound synthesis process.

## 5.2.2 Architectural Summary

This stratification of the system design is reflected in Fig. 5.1.<sup>8</sup> Rectangles show the modules we developed for the framework. Rounded rectangles denote existing modules which were incorporated. Higher modules either depend on lower modules or encapsulate (abstract) them. For example, the event layer is built on top of the system façade, whereas the system façade is an abstraction of the ephemeral and the confluent layer, and the ephemeral layer in turn is implemented either by an in-memory or a durable layer. Lines indicate coupling: The system façade manages the transactional (Tx) structures, the confluent layer is coupled to a synchronised durable layer, the database is coupled to the software transactional memory.

At the level of the system façade, the framework introduces the concept of mutable entities. Any number of mutations, such as creating an object, adding an object to a collection of objects, or changing the parameter of an object, can be grouped together in one transaction. The transaction

<sup>8</sup>The figure is adapted from Hanns Holger Rutz (2012a), 'A Reactive, Confluently Persistent Framework for the Design of Computer Music Systems', in: *Proceedings of the 9th Sound and Music Computing Conference (SMC)*, Copenhagen, pp. 121–129; that paper also gives a condensed version of the implementation of the framework.

ensures that the user data structures remain coherent and any errors occurring within the transaction can be safely undone. When the selected system is durable, successful transactions result in the mutations being written to the secondary storage (hard disk). Furthermore, the confluent system associates an incremental version with each transaction which corresponds to the next point in  $\mathcal{T}_K$ .

The problem with a “plain” data structure is that it assumes *one* actor manipulating the structure. Naturally that one actors “knows” about the changes to the structure because it issues them. In a modular system there are independent components whose knowledge must be updated if any elements they depend on are mutated. A paradigm such as dataflow (briefly explained in Sect. B.5) provides such an active connectedness between elements. In our implementation, the event layer plays a similar role, although it is more general in allowing various kinds of messages to be propagated.

The next layer, expressions, narrows the event layer to dataflow variables. Other than the dataflow variables shown in Sect. B.5 which are initially unknown and will be assigned only one value eventually, expressions always have an initial value and may be updated multiple times. Thus they closer resemble the (non-audio) objects in a *PD* or *Max* patch.

The topmost layer, *SoundProcesses*, introduces a model of  $\mathcal{T}_{(P)}$  and connects to the real-time sound synthesis layer *ScalaCollider*. Expressions may now be bi-temporal by having an associated temporal expression for the performance time. Expressions and other bi-temporal objects can be organised in bi-temporal groups, which in the manner of “timelines” may be scanned in real-time by a transport mechanism. The core object here is a sound process *Proc* which is understood as signal processing function placed in  $\mathcal{T}_{(P)}$  and associated with a number of time-dependent parameters or other sound processes.

### 5.3 System Façade and Transactional Encapsulation

By a system *s* here we understand an object which manages transactions and provides a model for mutable data. In order to be able to use the same data structures with multiple systems—for example an ephemeral system which does not preserve the historic trace of the structure versus

a confluent system which does preserve this trace on hard disk—all systems must implement a common façade. This façade must provide a functionality that is both minimal but expressive enough to account for the different types of systems.

Our data model is based on mutable cells similar to the reference cells approach employed by some software transactional memories. To understand this approach, we first review STM.

### 5.3.1 Software Transactional Memory

Transactional memory,<sup>9</sup> while its historic origin is in database systems,<sup>10</sup> is nowadays mainly used as a mechanism to allow concurrent processes to safely use shared memory. Yet, our own use case has more to do with the versioning of the database updates and failure safety—something that is seen in literature as a «a side benefit».<sup>11</sup> But first the basic mechanism of transactional memory needs explanation.

There are two flavours, software and hardware transactional memory. The latter concerns special chip architectures, so only the former—purely implemented as software—is of interest here. STM is often contrasted with lower level concurrency control such as locking. There are two main advantages of STM. First, it provides a higher level abstraction of dealing with concurrency, while direct manipulation of locks requires to shift the focus away from an actual algorithm to implementation details and opens possibilities for introducing programming errors. Second, it is an “optimistic” approach, while locking is “pessimistic”.

To understand these, one must look at the problems occurring with shared memory concurrency. When a mutable datum is shared between two threads  $A$  and  $B$  which run in parallel, a situation can arise where a mutable cell  $M$  is first read by thread  $A$  and then overwritten by thread  $B$ . If  $B$  completes before  $A$  then  $A$  at a later point in time operates on an invalid knowledge of  $M$  (having read a value which can no longer be observed). In a pessimistic setting, one assumes that this kind of conflict is the norm and establishes strong constraints for example by using a mutually exclusive lock which forbids  $B$  to access  $M$  during the whole time that  $A$  signalled its usage.

---

<sup>9</sup>Tim Harris et al. (2007), ‘Transactional Memory: An Overview’, *IEEE Micro* **27**(3), pp. 8–29.

<sup>10</sup>See for example George Copeland and David Maier (1984), ‘Making Smalltalk a Database System’, *ACM SIGMOD Record* **14**(2), pp. 316–325

<sup>11</sup>Harris et al., ‘Transactional Memory: An Overview’.

STM on the other hand optimises the case where no conflict occurs, and multi-threaded code is not thwarted by locking mechanisms. Only in the less expected case of a contention between threads, the STM is penalised by spending more time resolving the conflict.

## Transactions

In our scenario, we are less interested in the properties of running concurrent code than in the semantics of transactional encapsulation. T. Harris et al. offer the following definition of a transaction:

«A transaction is a sequence of instructions, including reads and writes to memory, that either executes completely (commits) or has no effect (aborts). When a transaction commits, all its writes become visible, and other transactions can use those values. When a transaction aborts, the system discards all its speculative writes.»<sup>12</sup>

This quote describes two major properties of transactions, atomicity and isolation. Another property related to atomicity is consistency, the three forming the acronym ACI. J. Gray derives the transaction notion from contract making:<sup>13</sup> Atomicity then signifies that no element of the contract can be separated, the contract is valid in its entirety, and either all parties agree to it or it is not effective. Consistency means the transformations under the contract maintain the “correctness” of the object they operate on. Isolation makes transactions function in a concurrent setting, where a transaction made by thread *A* hides its mutations from another thread *B* up until the moment that the contract is “fulfilled” and all mutations become visible at once.

Therefore, consistency and isolation could be seen as technological requirements which guarantee the semantics of atomicity. From a user’s perspective, what matters is atomicity. If one decides in the middle of the transaction that an assumed condition does not hold, or if the system detects a conflict between threads—for example the composer is trying to modify an object while a reaction from an autonomous algorithm interferes—the transaction is aborted as if nothing had happened and perhaps tried anew. If a transaction is considered to be decision-making,

<sup>12</sup>Harris et al., ‘Transactional Memory: An Overview’.

<sup>13</sup>Jim Gray (1981), ‘The Transaction Concept: Virtues and Limitations’, in: *Proceedings of the 7th international conference on Very Large Databases*, IEEE, Cannes, pp. 144–154.

atomicity means that a series of operations implied in the decision must be thought as one compound operation, and logically there is no temporal gap between the constituting operations, temporality is only attached to the decision as one instant.

As a simple example, a sound process is started which is synchronised with another process. The operative steps of the transaction might involve connecting the process to a sound diffusion mechanism, and preparing the signal processing components of both synchronised processes to start playing at the same time. All must become audible at once and not successively, even though the program implementation needs to prepare these individual elements sequentially. Also, if an error occurs, we do not want either of the two sound processes to play alone, so all the operative steps must be undone. Finally, transactions also permit the user or an algorithm to be *speculative*. One could begin traversing a data structure, issuing transformations on the hypothesis of an eventual condition—for example that a solution to a constraint will be found—and once the condition is decidable and the hypothesis does not hold, the transaction may be actively aborted without side effects.

We believe there is a confusion with respect to the usefulness of transactional semantics in creative software, as demonstrated by the following statement of Eaglestone et al.:

«Within this context [supporting creative activity], conventional requirements for transaction correctness, i.e. the ACID (atomicity, consistency, independence and durability) test, may not be valid, since these are concerned with completeness and semantic isolation of transactions such that they do not interfere with each other, whereas, in seeking an artistically valid result the interaction and interference between activities may be of value. An extreme example is where failure can provide unexpected results that have artistic value.»<sup>14</sup>

There are two aspects of confusion. Firstly, a linguistic one. Transaction as a software concept does not need to align with transactions in terms of the artistic process, where in fact an artistic “transaction” might be decomposed into a number of software transactions, and only part of the

---

<sup>14</sup>Eaglestone et al., ‘Information systems and creativity: an empirical study’.

artistic process may be observed. Secondly, and more importantly, the confusion is between the technological and the epistemological layer in Rheinberger's taxonomy. We are dealing here with the foundation of the system and therefore seek a "stable subroutine". The bottom system must «prevent a breakdown of its reproductive coherence»,<sup>15</sup> which is the consistency of the transaction permitting connectivity (operational closure). *Because* the "goals" of the software designer and the composer/user will never correspond with each other, a "good" software design has never deterred a composer from "abusing" the system. The speculative feature of transactions indeed supports the upper epistemic layers by opening space for exploration.

### **Piggybacking an STM**

There are different approaches to implementing an STM. A survey by D. Goodman et al. looks at different systems that specifically support the *Scala* language.<sup>16</sup> They focus on the front-end interface—how the STM is accessed by the user—and less on the back-end engines which is also what we are interested in. They identify three types of interfaces:

- (1) Library calls: A transaction is typically wrapped in a call to a library function `atomic`, and mutable cells must be explicitly declared transactional by using a special reference cell type provided by the library. The advantage is the fine-grained control over which parts of a program use transactional semantics and which not. Disadvantages are the need to change to a different and more verbose syntax when using transactional variables, and that existing code cannot be retroactively instrumented.
- (2) Byte code rewriting: The STM transparently lifts the variable mutations to transactional equivalents at runtime. This can either happen globally, or be restricted to particular classes or methods which have been specially annotated. The advantage is a more concise code, and some systems will implicitly extend the transactional context to methods called from within annotated methods, making this approach suitable for retrofitting existing code. On the downside, a special runtime agent is required that detects and processes

---

<sup>15</sup>Hans-Jörg Rheinberger (1997), *Toward a History of Epistemic Things: Synthesizing Proteins in the Test Tube*, Palo Alto: Stanford University Press, p. 80.

<sup>16</sup>Daniel Goodman et al. (2013), 'Software transactional memories for Scala', *Journal of Parallel and Distributed Computing* **73**(2), pp. 150–163.

the annotations and does the byte code rewriting, possibly interfering with the virtual machine or other observing systems such as runtime debuggers.

- (3) Compiler modification: The most radical approach is to build STM support directly into the language, by introducing for example a new `atomic` keyword which will make the compiler treat the code following this keyword differently. The authors of the survey were not aware of any *Scala*-based system going this route.

The authors of the survey are ultimately biased towards the second approach, since they employed it in their own project *Manchester University Transactions for Scala (MUTS)*.<sup>17</sup> They notably distinguish library-based and byte code rewriting techniques by their ability to account for side effects in existing code. But under further examination, the only side effect they considered was mutating variables in memory. A far greater problem in STM usage is the management of *irreversible* side effects such as printing out text, sending out network messages or writing to files on disk, all which can be summarised as input/output (I/O). To the best of our knowledge, none of these STM systems automatically handles I/O in a transactionally safe manner, so it remains the responsibility of the programmer to correctly perform I/O.

On the other hand the explicitness of reference cells in the library approach makes the first interface type—despite the added verbosity—particular *advantageous* in our case. It allows us to piggyback on the existing STM library and enrich it with the necessary information of the confluent tracer. The library we build on is *Scala-STM* whose design is overseen by an expert group that aims to create a common application programming interface (API) for different back-ends. For example, the *Akka* framework mentioned earlier also implements this API for a back-end called *Multiverse*. It even offers transactions spanning multiple threads, combining actors and transactions into ‘transactors’, something which is beyond the scope of this thesis. Instead we rely on the lightweight *Scala-STM* reference implementation which evolved from N. G. Bronson’s *CCSTM*.<sup>18</sup>

---

<sup>17</sup>Daniel Goodman et al. (2011), ‘MUTS: Native Scala Constructs for Software Transactional Memory’, in: *Proceedings of the Second Scala Workshop*, Stanford.

<sup>18</sup>Nathan G. Bronson, Hassan Chafi and Kunle Olukotun (2010), ‘CCSTM: A library-based STM for Scala’, in: *Proceedings of the First Scala Workshop*, Lausanne.

The following example illustrates the basic interface used in *Scala-STM*. We first show the non-transactional plain version:

```
var a = 0          // mutable integer variable initially holding value 0
var b = 2          // mutable integer variable initially holding value 2
a = b * b          // update 'a' to hold the squared value of 'b'
println("Now 'a' is " + a)
```

Using STM, it becomes:

```
val a = Ref(0)    // integer reference cell initially holding value 0
val b = Ref(2)    // integer reference cell initially holding value 2
atomic { implicit tx =>
  a() = b() * b() // update 'a' to hold the squared value of 'b'
  Txn afterCommit { _ =>
    println("At the end of the txn, 'a' is " + a.single())
  }
}
```

The atomic block encapsulates all manipulations which form a transaction. It executes the function passed to it as argument with an instance of the transactional context `tx`. Reading a reference cell via `a()`, or updating it via `a() = ...` requires such a transactional context in a second argument list. It is therefore not possible to accidentally access reference cells outside of a transaction. The second argument list is marked `implicit`, freeing us from the need to explicitly pass the `tx` value to each and every call to reference cells:<sup>19</sup>

```
trait Ref[A] {
  def apply () (implicit tx: InTxn): A
  def update(v: A)(implicit tx: InTxn): Unit
}
```

A syntactic sugar allows to express the reading of the cell's value `a.apply()` as `a()`, and the writing of the cell's value `a.update(1)` as `a() = 1`.<sup>20</sup>

Printing the value of `a` to the console is an I/O operation and should be only performed when the transaction successfully completes. Often we do not care about the extra effort and place the `println` call directly inside the atomic block at the price of having it potentially executed

<sup>19</sup>There is a pitfall, though. When using nested methods, an implicit transaction context from an outer scope is visible in any inner method. If that method is called outside the outer transaction and the programmer forgets to ask for a fresh transactional context argument, a runtime error will occur, since one is trying to use an already completed transaction. This pitfall is very real and has been the source of many bugs in our code base. The best prevention is to avoid using nested methods with transactions wherever possible.

<sup>20</sup>Note the similarity to the examples of `Sink` and `Source` in section B.4.3.

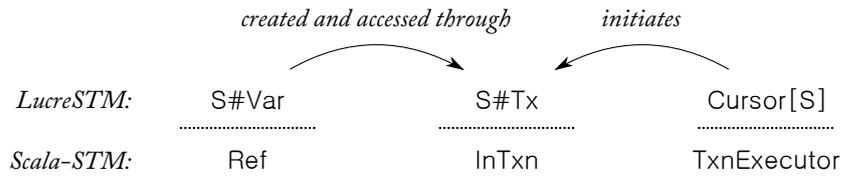


Figure 5.2: Encapsulating variables, transaction context and executor in our own API

multiple times if the transaction is aborted and retried. Obviously, the shortcut cannot be taken when writing to files, such as saving the mutable cell values in a database. The `Txn.afterCommit` call was used to demonstrate how functions are correctly scheduled to be executed only when the transaction succeeds. The value of a still must be transactionally read, but *Scala-STM* offers micro-transactions in the form of the `.single` method which can be thought of as an atomic block just to allow for the `apply()` call with minimal overhead.

### Our Payload

As long as the data structures are kept in memory, we can simply encapsulate the existing *Scala-STM* library in our own abstractions. These delegate calls, such as reading or writing cells, to the underlying peer in the *Scala-STM* perspective. The three main abstractions are shown in Fig. 5.2. The peer layer shows the reference cell and transactional context type from the previous examples. `TxnExecutor` is the instance which provides the `atomic` method. Our extension named *LucreSTM*<sup>21</sup> provides base types for cells and transactions as well, but they are mostly seen as type projections `Var` and `Tx` of a system `S`.

The `Cursor` type is to be understood as a moving position in  $\mathcal{T}_K$ , orthogonal to the traditional timeline cursor which describes a position in  $\mathcal{T}_{(P)}$ . Naturally, there can be multiple cursors in a system, representing different branches in the version graph. The transactional context `S#Tx`—which for brevity will simply be called ‘transaction’ in the following—can now carry further payload such as a reference to the cursor (knowing where in  $\mathcal{T}_K$  we are) and the system (database connection and event bus, which will be described later). It also extends the interface with methods for creating and serialising mutable data (serialisation is reading and writing data

<sup>21</sup>For the curious reader, I was looking for a short and unique name. In the durable variant, we use *BerkeleyDB JE* which was originally written by a company called Sleepycat Software. I also own a sleepy cat whose name is Lucrecia, or Lucre in short.

from and to disk). Systems can refine the type  $S\#T_x$  based on their needs. Mostly they add functions for the abstractions to interact with each other but which do not have to be exposed to the user.

### 5.3.2 Identifying Entities

A mutable cell needs an identity which allows us to talk about *that cell*, to establish a coherence between the different values it takes on over time, and to use the cell as a placeholder independent from its current value. In natural language, we typically *name* things to establish identity. For example, we may say a sound process is parametrised by a frequency  $f$ . A sound-producing function can then be defined by using the abstract cell  $f$  independent from the values it may take at a later point.

In a pure in-memory system, the host language *Scala* automatically gives us identities in the form of object references. The statement `var a = 0` creates a virtual object reference known by the symbol `a` which establishes permanence of an otherwise instable object (the value of `a` may change over time). Likewise, in the STM equivalent `val a = Ref(0)`, the object reference of the mutable cell is known by the symbol `a`.

It gets more complicated when mutable cells are written to and read from disk. If for example two entities  $X$  and  $Y$  both make reference to another entity  $Z$ , this reference must be expressible in a form which can be written to disk. If the application is quit and restarted, reading  $X$  and  $Y$  must again reconstruct the reference to  $Z$ , so that both references are again the same and so that subsequently the value of  $Z$  can be found on disk. The memory independent reference shall be called *identifier* of the object  $Z$ . In the creation of objects, each identifier to a new object must be different from all previously known identifiers, because otherwise a confusion between the new and an old object will happen.

In *LucreSTM*, identifiers are represented by the type projection  $S\#ID$  and their creation and serialisation is managed by the transaction context. For an ephemeral system the identifier can simply be an integer number which is incremented for each new identifier. In the confluent case where entities can re-enter a data structure, the identifier must be able to distinguish “original” and “copy”. This is done analogously to the fat node technique by A. Fiat and H. Kaplan which

Abstraction	Ephemeral systems	Confluent systems
S#ID	opaque Int	opaque tuple (Acc, Int)
S#Acc	Unit	opaque randomised compressed path

Table 5.1: System independent abstraction for object identifiers and access, and the concrete forms for ephemeral and confluent systems

uses a tuple  $(p, s(f))$  to identify versions of entities, with  $p$  being the access path from the seminal version of a fat node  $s(f)$ .<sup>22</sup> In our system, the seminal node would again be identified by a unique integer number. The access path is represented by the type projection S#Acc. To the user, this type is—like S#ID—opaque and only appears as a parameter used in deserialisation. Table 5.1 summarises the identification types for the two types of systems.

### 5.3.3 Concept of Associated Types

In their survey on generic programming abstractions, R. Garcia et al. introduce the term ‘concept’ as a formalisation of abstractions. According to them, a concept usually «... consists of *associated types*, *valid expressions*, semantic invariants, and complexity guarantees. The associated types of a concept are opaque types that are required by the concept (e.g., used in valid expressions) and must be defined by any model of the concept.»<sup>23</sup>

Although we only learned about this survey after implementation, our layout quite closely matches the idea of associated types as defined above. The core system declaration is shown in Listing 5.1. Opacity means that most of the details of these types are only known and meaningful to the models of the concept—the concrete implementing systems. For example, the path access Acc is completely abstract, and the identifier ID has a very weak requirement of being subtype of Identifier which only states that an identifier can be serialised and disposed of.

The example in Listing 5.2 demonstrates how the types interoperate in this mostly opaque manner. Only one type S is required to parametrise this hypothetical structure with any given system. When the disturb method is called inside a transaction within that system, S#Tx, it is

<sup>22</sup>Amos Fiat and Haim Kaplan (2003), ‘Making data structures confluently persistent’, *Journal of Algorithms* 48(1), pp. 16–58, §4.1. This is discussed in detail in Sect. 5.5.3.

<sup>23</sup>Ronald Garcia et al. (2007), ‘An Extended Comparative Study of Language Support for Generic Programming’, *Journal of Functional Programming* 17(2), pp. 145–205, §2 [Emphasis in original]

```

trait Sys[S <: Sys[S]] {
  type Tx    <: stm.Txn[S]
  type Var[A] <: stm.Var[S#Tx, A]
  type ID    <: stm.Identifier[S#Tx]
  type Acc
  ...
}

```

*Listing 5.1:* Type members of the system abstraction

```

trait Foo[S <: Sys[S]] {
  protected def freq: S#Var[Double]

  def disturb()(implicit tx: S#Tx): Unit = {
    val factor = math.random.linuxp(0, 1, 0.5, 2.0)
    freq()    = freq() * factor
  }
}

```

*Listing 5.2:* Parametrising an object with a system and interaction of its type members

capable of accessing and updating variables defined within the system, `S#Var` (the `apply` and `update` methods take the transaction as implicit argument which is thus not printed).

The type parameter `S` is peculiar in that it appears as part of its own constraint, the right-hand side upper bound in `S <: Sys[S]`. The form of type recursion is a powerful mechanism known as ‘F-bounded quantification’—a type is bound by a function  $F$  of itself.<sup>24</sup> The crucial point of recursion is in the declaration of `Sys` itself. It allows the associated types to be stitched together. We believe that our approach of defining associated types with a combination of F-bounded quantification and type projections is original. F-bounded types in *Scala* are a well known feature of its type system, but the use case we have seen before was merely in allowing type refinements to appear in super type interfaces without the interaction between a set of types.

For example, Listing 5.3 containing part of the declaration from `stm.Txn` shows how the system parameter is used to provide the matching identifiers and variables. A transaction is thereby guaranteed to create the specific identifier and variable types of the underlying system.

<sup>24</sup>P. Canning et al. (1989), ‘F-Bounded Polymorphism for Object-Oriented Programming’, in: *Proceedings of the fourth international conference on Functional programming languages and computer architecture*, ACM, pp. 273–280.

```
trait Txn[S <: Sys[S]] {  
  def newID(): S#ID  
  def newVar[A](id: S#ID, init: A)(implicit ser: Serializer[S#Tx, S#Acc, A]): S#Var[A]  
  ...  
}
```

*Listing 5.3:* The transaction context provides `newID` and `newVar` to instantiate `S#ID` and `S#Var`.

```
trait Sink[-Tx, -A] {  
  def update(v: A)(implicit tx: Tx): Unit  
}  
trait Source[-Tx, +A] {  
  def apply()(implicit tx: Tx): A  
}  
trait Var[-Tx, A] extends Source[Tx, A] with Sink[Tx, A]
```

*Listing 5.4:* Variable (mutable cell)

If the variable base type `stm.Var` is defined as in Listing 5.4, we can see how the types are properly aligned in the example of Listing 5.2. Frequency cell `freq` is of type `S#Var[Double]` which would have been produced by a call to `newID` and `newVar` from a transaction conforming to `S#Tx`. Its upper bound is `stm.Var[S#Tx, Double]` allowing the update to be performed with another instance of `S#Tx`.

The type recursion in the definition of `S` reminds us of the discussion of operational closure with regard to the works of Ashby and Spencer-Brown in Sect. 3.3.3. If we remove the recursion from Fig. 5.3a, the types become disassociated, as shown in Fig. 5.3b. The alternative approach of using path-dependent types is shown in Fig. 5.3c. It requires the presence of a value of the system to witness its type, and this value would have to be passed around to any mutable object, complicating the matter especially when serialisation is added.

## 5.4 Durability

Harris et al. see the relationship between I/O and transactions as a major research challenge, and outline three possible ways to combine them:<sup>25</sup>

---

<sup>25</sup>Harris et al., ‘Transactional Memory: An Overview’.

<pre> <b>trait</b> Sys[S &lt;: Sys[S]] {   <b>type</b> ID &lt;: Ident[S#Tx]   <b>type</b> Tx } </pre>	<pre> <b>trait</b> Sys {   <b>type</b> ID &lt;: Ident[Tx]   <b>type</b> Tx } </pre>	<pre> <b>trait</b> Sys {   <b>type</b> ID &lt;: Ident[Tx]   <b>type</b> Tx } </pre>
<pre> <b>trait</b> Muta[S &lt;: Sys[S]] {   <b>def</b> id: S#ID   // well defined:   <b>def</b> dispose(tx: S#Tx) =     id.dispose(tx) } </pre>	<pre> <b>trait</b> Muta[S &lt;: Sys] {   <b>def</b> id: S#ID   // incompatible type:   <b>def</b> dispose(tx: S#Tx) =     id.dispose(tx) } </pre>	<pre> <b>trait</b> Muta[S &lt;: Sys] {   // evidence required:   <b>val</b> s: S   <b>def</b> id: s.ID   // well defined:   <b>def</b> dispose(tx: s.Tx) =     id.dispose(tx) } </pre>
<pre> <b>trait</b> Ident[Tx] {   <b>def</b> dispose(tx: Tx): Unit } </pre>		
(a) Recursive	(b) Non-recursive (dysfunctional)	(c) Path-dependent

Figure 5.3: Different attempts to describe associated types

- (1) Execute I/O immediately within the transaction. The difficulty lies in undoing these actions when the transaction is rolled back (aborted). I/O might be categorised into undoable and non-undoable actions, and only undoable actions are allowed inside a transaction.
- (2) Delay the execution until the transaction commits. This may be problematic in terms of latency if some real-time guarantees are made for the I/O.
- (3) Restrict I/O to forms that «can themselves become transactional, such as access to a transactional database or file system».

For the durability layer, where the state of the mutable cells is stored on and retrieved from disk, we have chosen a combination of (2) and (3). In the ephemeral case, data can be immediately read and written to the database, given that the chosen database provides transactional safety by itself. In the confluent case, because we do not know until the transaction is closing if a new sub-tree in the version graph is entered or not, writes are buffered and flushed when the transaction is committed.

The number of database options is very large, but it is narrowed down by the requirements of the data model. A good overview is given by R. Cattell who compares traditional relational (table

based) databases which are typically accessed through the *Structured Query Language (SQL)* with newer approaches which are subsumed under a “NoSQL” label.<sup>26</sup> The latter are also called *data stores* to better distinguish them from the traditional database. The three main features of these new types of stores are scalability, availability and the abolition of fixed schemas (the layout of a data entry). Horizontal scalability means the system grows by being distributed across multiple computers in a network, whereas increased availability means that through replication in a network the store is made less prone to outages.

The feature that matters to us is “schemalessness”: Since we need to store arbitrary data structures, the table model of ordinary relational database management systems (RDBMS) is unsuitable. The simplest and most flexible access structure for us is the *Key-Value store*. It can be thought of as a durable version of a normal associative array (dictionary, map). Values are inserted at a given key. To retrieve a value, the key must be known. The data store typically allows any form of and size of keys and values, as long as they can be represented as a series of bytes. This is the origin of the term ‘serialisation’: Before a datum is inserted into the data store, its key and its value must be converted to an agreed upon series of bytes. When a datum is retrieved, the store returns the serialised form, and the client must reconstruct the original object by interpreting the series of bytes.

### 5.4.1 Serialisation

As a simple example, consider keys and values which are 32-bit integers. A straightforward serial representation of a 32-bit integer is a series of four bytes corresponding to bits 24–31, bits 16–23, bits 8–15, and bits 0–7. A boolean value could be represented by a single byte which is either 0 (for false) or 1 (for true). A string could be represented by the byte series of its UTF-8 character encoding, etc. The abstract interface we use for this kind of data store is shown in Listing 5.5. Instead of directly taking and returning arrays of bytes, it uses function arguments which operate on objects of type `DataInput` and `DataOutput`. These objects provide convenient methods to serialise standard types as the ones mentioned above. The `put` method—storage—

---

<sup>26</sup>Rick Cattell (2010), ‘Scalable SQL and NoSQL Data Stores’, *ACM SIGMOD Record* **39**(4), pp. 12–27.

```

trait DataStore {
  def put    (k: DataOutput ⇒ Unit)(v: DataOutput ⇒ Unit)(implicit tx: TxnLike): Unit
  def get[A] (k: DataOutput ⇒ Unit)(v: DataInput ⇒ A   )(implicit tx: TxnLike): Option[A]
  def contains(k: DataOutput ⇒ Unit)                (implicit tx: TxnLike): Boolean
  def remove (k: DataOutput ⇒ Unit)                  (implicit tx: TxnLike): Boolean
}

```

Listing 5.5: Abstract interface of a key-value store for serialised data

```

trait DurableIntVarImpl {
  protected def store: DataStore
  protected def id: Int // (S#ID)

  def apply()(implicit tx: Tx): Int = {
    val opt = store.get(_.writeInt(id))(_.readInt())
    opt.getOrElse sys.error("Key_not_found_" + id)
  }
  def update(v: Int)(implicit tx: Tx): Unit =
    store.put(_.writeInt(id))(_.writeInt(v))
}

```

Listing 5.6: Implementation of a durable integer cell

takes a *key-writing* function and a *value-writing* function, and the *get* method—retrieval—takes a *key-writing* and a *value-reading* function to handle requests.

The reference cells as defined in Listing 5.4 can now be simply connected to the `DataStore` interface, assuming the variable has an identifier `S#ID` (see Listing 5.1) which is used as the storage key. Listing 5.6 shows an implementation for integer variables.

A mutable structure by definition contains variable cells, so elements of type `S#Var[A]`. In order to serialise the mutable structure, all that needs to be done is write the `S#ID` for each constituent variable. Fig. 5.4a is such a structure, a singly linked list. Each element of the list with type `A` is wrapped in a cell which apart from the element `value` contains a variable pointer next to the next cell in the list. We use an `Option[Cell]` here, so the last element in the list will store the value `None` in the next variable. The list itself contains a reference `head` to the first element in the list (a `None` would indicate an empty list here).

To understand how this structure is stored and retrieved, consider the case where a list of three integers 3,5,8 is given and the task is to traverse the list and print these elements. The algorithm

```

trait LinkedList[S <: Sys[S], A] {
  def head: S#Var[Option[Cell]]
  trait Cell {
    def next: S#Var[Option[Cell]]
    def value: A
  }
}

def traverse[S <: Sys[S]](l: LinkedList[S, _])
  (implicit tx: S#Tx) = {
  def loop(opt: Option[l.Cell]) = opt match {
    case Some(cell) =>
      println(cell.value)
      loop(cell.next())
    case None =>
  }
  loop(l.head())
}

```

(a) (b)

Figure 5.4: Linked list data structure, and example of its traversal

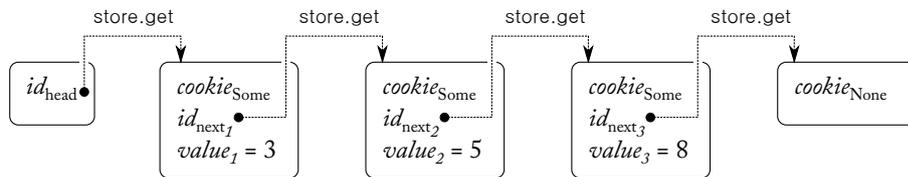


Figure 5.5: Subsequent key-value access in the data store for the traversal of list 3, 5, 8

is shown in Fig. 5.4b, whereas Fig. 5.5 depicts the corresponding entries in the data store. When the head element is accessed via `l.head()`, the value stored at the key `id_head` is looked up via `store.get` and deserialised to a `LinkedList[S, Int]#Cell`. To distinguish whether the optional content is `Some` or a `None`, a “cookie” byte is used. The cell itself would then follow as the identifier of its next variable, and the cell’s actual integer value.

This approach to serialisation could be called top-down because the data store reader is required to know how to deserialise the entries. This knowledge is formalised in the type `Serializer[Tx, Acc, A]` which provides the encoding and decoding functions for an object of type `A`. As a result, some degree of ‘transparency’ is lost, since such serializers must be passed around as soon as a structure should be capable of being made durable.<sup>27</sup> We try to keep this change as unobtrusive as possible on the user side by requiring serializer arguments only when creating a new reference cell via `tx.newVar` (cf. Listing 5.3), making them implicit and providing a set of standard serializers which include parametrised types such as `Option[A]`. For example, to create a new instance of a linked list of integers, the serializer for `Int` is automatically found.

<sup>27</sup>See the discussion of transparency and orthogonality (generality) in J. Eliot B. Moss and Antony L. Hosking (1996), ‘Approaches to Adding Persistence to Java’, in: *Proceedings of the First International Workshop on Persistence and Java*, Drymen, Scotland.

On the implementation side, the serialisation code for `Cell` and `LinkedList` must be written, however `Option[Cell]` is automatically composed with the cookie preamble shown in Fig. 5.5.

The balance between transparency and flexibility or granularity is similar to what has been discussed with respect to library calls versus byte code rewriting in STM implementations. Again the explicit library solution is required here in order to make the serialisation interlink with confluent persistence. While there are completely transparent approaches, such as the standard *Java* Serialization API,<sup>28</sup> this is not only inflexible, but can be fragile and pose space and time penalties:

- › An automatic encoding of a class and its state means that as soon as an implementation detail of the linked list is changed, an already stored data entry might become unreadable, because the new implicit schema of its serial representation has changed. A simple software update could mean that we cannot access a musical piece stored with a previous version any more. With an explicit schema, the serialised data is less tightly linked to the in-memory representation, and with careful design, multiple schema versions can be supported.
- › Storing the example list with three elements using *Java* Serialization requires around 1200 bytes of space, compared to 36 bytes with our scheme. Since the database can easily grow to hundreds of megabytes in a generative sound installation which continually creates objects, this improvement of factor 30 is crucial.
- › The slowdown is much worse than a factor 30 in disk I/O, because the bottom-up serialisation means that the JVM needs reflection to look up the serialised classes in its class loader and performs schema verifications.

There are of course more performative options, and we have evaluated some of them, such as *Hibernate*<sup>29</sup> and *DataNucleus*,<sup>30</sup> however they are also more complex than necessary—including

<sup>28</sup>Cf. Lukasz Opyrchal and Atul Prakash (1999), 'Efficient Object Serialization in Java', in: *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems*, pp. 96–101

<sup>29</sup><http://www.hibernate.org> (visited on 16/02/2013)

<sup>30</sup><http://www.datanucleus.org> (visited on 16/02/2013)

unused features such as querying—do not directly compose with our temporal semantics, and often deal poorly with the specific requirements of the *Scala* language. Overall, the benefits of the lightweight pure library solution we chose clearly outweigh the disadvantages, such as added verbosity and the increased risk of introducing errors in the manually written serialising functions. With the advent of a macro system, first introduced in *Scala* 2.10, we hope that future versions will be able to automatise the verbose and error prone parts of the serialisation.

### 5.4.2 Key-Value Store

The back-end chosen is *BerkeleyDB Java Edition (JE)*,<sup>31</sup> an open source key value store. It is written purely in *Java* and runs directly embedded in the client application process. Furthermore, it offers ACID guarantees based on transaction management.

JE is using a B<sup>+</sup>Tree<sup>32</sup> for storing the key index, where the leaf nodes refer to positions in a sequentially written log structure which stores the actual values. In the benchmarks published in the 2006 white paper, it has a read performance comparable to the older (and independently maintained) *C* version, and is particularly performative in write operations—something we rely on heavily.

Benchmarks are a difficult subject, because they depend on many factors, such as file system and operating system, virtual machine and cache tuning and the access pattern of the database. Published benchmarks can seldom be reproduced and are often fabricated by the database vendors to promote their products. The theoretical worst case cost of a B<sup>+</sup>Tree insertion and query is  $O(\log n)$ ,<sup>33</sup> but in practice the dominant factor might be in-memory caching. Running various unit tests of our framework against both an in-memory system and a durable system with JE yielded a slowdown factor of approximately 6 to 10, which appears to be almost independent of the data structure sizes. In a sound installation in which the system was tested (see Sect. 4.4.4), even when the database grew to several hundred megabytes after hours of running and constantly producing data, there was no significant impact on the fluidity of the sound production. However,

---

<sup>31</sup>*Berkeley DB Java Edition Architecture* (Sept. 2006), An Oracle White Paper, URL: <http://www.oracle.com/technetwork/products/berkeleydb/learnmore/bdb-je-architecture-whitepaper-366830.pdf> (visited on 09/02/2013).

<sup>32</sup>Cf. Douglas Comer (1979), 'The Ubiquitous B-Tree', *ACM Computing Surveys (CSUR)* 11(2), pp. 121–137

<sup>33</sup>See also Sect. 2.6

it would be advantageous to conduct more detailed profiling of the system's performance and perhaps compare the speed of JE with other database solutions.

### 5.4.3 Transactional Coupling

The integration of *JE* is straight forward. The `DataStore` interface translates well to its API, and the only question concerns the opening and closing of transactions. *Scala-STM* provides two elements which are useful. One is a `TxnLocal` type which is a fusion of a transactional reference cell and a thread local variable. In each new STM transaction, the value carried by a `TxnLocal` is undefined. As soon as it is accessed, it is initialised and then stays valid until the end of the transaction. Therefore, it can also be understood as a transaction local *lazy val*. Whenever a value is read from or written to the store, the database transaction is retrieved via a `TxnLocal`, so there is exactly one database transaction matching the STM transaction. The second mechanism is to register the database's commit function as an "external decider" with the STM. This hook will be invoked when the STM has reached its own commit phase and has verified that the transaction is guaranteed to succeed, given the permission of this "external decider". If an I/O error occurs and the database's transaction is aborted at this point, the STM transaction will still be able to roll back. On the other hand it is guaranteed that the "external decider" is never called repeatedly if the STM transaction is otherwise retried. This coupling is illustrated in Fig. 5.6. It should be noted that the durable system wraps an in-memory system, therefore it is quite possible that a cursor step is made which does not require database access. In that case the `TxnLocal` is never read, and consequently no database transaction opened.

## 5.5 Confluent Semantics

We will now present the main system which is based on the concept of *confluent persistence*. The opposition of 'ephemeral' and 'persistent' has been used several times before in this chapter, so the meaning of these terms should be explicated. They relate to the capability of data structures to preserve the history of their mutations. A concise definition which also includes different qualities of persistence is given by J. R. Driscoll, D. D. Sleator and R. E. Tarjan:

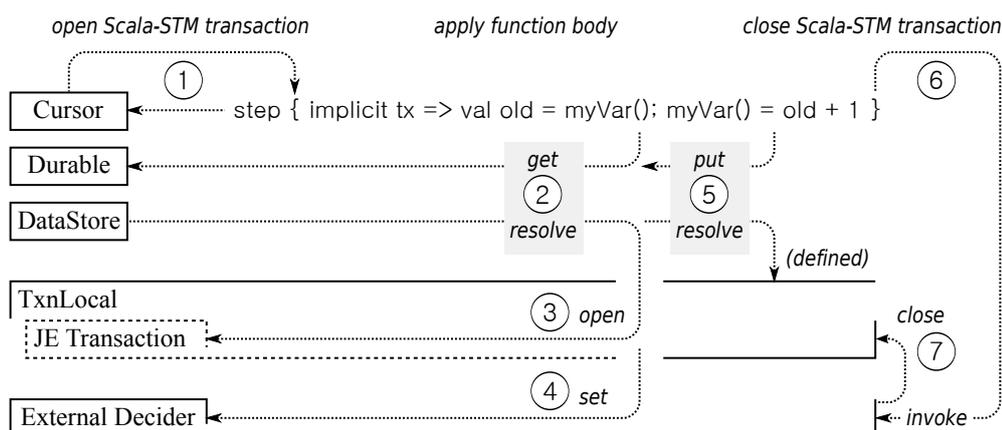


Figure 5.6: Coupling in-memory STM and database transaction

«A typical data structure allows two types of operations: queries and updates. A query merely retrieves information, whereas an update changes the information represented. Such a data structure is said to be *ephemeral* if queries and updates may only be done on the current version. With an ephemeral structure, the sequence of versions (one for each update done) leading to the current version is lost.

In the *partially persistent* form of an ephemeral data structure, queries may be performed on any version that ever existed, while updates may be performed only on the most recent version. Thus, the collection of versions forms a linear sequence, each one (except the current one) being the predecessor of exactly one version. In the *fully persistent* form of an ephemeral data structure both queries and updates are allowed on any version of the structure. Therefore, one version may be the predecessor of several versions (one for each update that was applied to it). The graph of relations between versions is a tree.

... A fully persistent version of an ephemeral data structure that supports an update in which two different versions are combined is said to be *confluently persistent*. The relationship between versions of a confluently persistent data structure is that of a directed acyclic graph.»<sup>34</sup>

<sup>34</sup>James R. Driscoll, Daniel D. Sleator and Robert E Tarjan (1994), 'Fully Persistent Lists with Catenation', *Journal of the ACM (JACM)* 41(5), pp. 943–959.

A major contribution of these authors along with N. Sarnak is the design of a general transformation by which any ephemeral linked data structure can be turned into their partially or fully persistent equivalent.<sup>35</sup> Because their paper is a common reference in the literature and serves as a basis for further developments and improvements, it is useful to review their main points and terminology.

### 5.5.1 Persistent Transformation

To allow the transformation to be universally applied, generalisations must be made on the *shape* of a data structure and the *operations* permissible. The operations have been mentioned in the quotation: A read only operation which is often called ‘query’ or ‘access’, and a mutating and re-writing operation called ‘update’ or ‘assignment’. The structure itself is decomposed into a number of nodes which in turn contain «information fields» and «pointer fields», the latter being links to other nodes. This structural model closely resembles for example the C programming language and corresponds with the pointer machine model of computation.<sup>36</sup> Even in a higher level programming language such as *Scala* which does not have explicit memory locations accessed via pointers, this view is still useful since nodes can be equated with objects and pointers can be equated with references to other objects. The linked list is a good example of a data structure which is decomposed into nodes linked through a *next* field (see Fig. 5.5).

In this sense, walking through the data structure is a query-only procedure, whereas creating a node, changing an information field or a pointer field constitute mutating updates. Logically each persistent modification can be seen as the creation of a new independent version of an ephemeral data structure which will be incrementally numbered, beginning at version 0 (for example an empty list). An update or version operation  $i$  can also be seen as a transaction, since repeated steps of modifying a field within one such logical operation bury the intermediate updates and only the latest change in association with version  $i$  will be remembered.

Driscoll et al. propose two different ways to implement persistent transformations, called the *fat node* and the *node copying*—for partial persistence—or *node splitting*—for full persistence—

---

<sup>35</sup>James R. Driscoll et al. (1989), ‘Making data structures persistent’, *Journal of Computer and System Sciences* **38**(1), pp. 86–124.

<sup>36</sup>Amir M. Ben-Amram (1995), ‘What is a “Pointer Machine”?’, *ACM SIGACT News* **26**(2), pp. 88–95.

method. In the fat node method, each node is augmented by a version stamp, and each (ephemeral) field is exchanged for an ordered list of tuples consisting of the field values and the stamps of the versions at which that value was assigned. The nodes are “fat” now because they contain their own history which can grow arbitrarily. With a balanced binary tree used to store the versioned tuples, it is easy to see that this augmentation introduces a slow down factor of  $O(\log m)$  for each access, where  $m$  is the number of assignments of a field.

The node copying and node splitting methods are more involved, but provide better performance. They require pointers (references) to be bi-directional, and nodes cannot grow infinitely fat but are limited to a fixed size above which the node contents gets copied to a new node or split between the old and the new node. A major implied limitation of the fixed size is that the in-degree of nodes—the number of nodes pointing to one particular node—is limited as well. This along with the burden of maintaining backward links, copy pointers and live versus dead tags on nodes makes the fat node method much more preferable for representing general data structures, despite its worse performance.

### 5.5.2 A Note about Purely Functional Data Structures

An alternative model to persistence has come to prominence in recent years and is embodied by “purely functional data structures”.<sup>37</sup> These are characterised by disallowing pointer updates, and good amortised performance is often achieved through the use of lazy computation, thus they are particularly suited for languages such as *Haskell*. The main update mechanism is through path copying or “structural sharing”, where a new root of the data structure is created along with new branches as necessary for including updated elements, up until the point where branches in the previous structure are encountered which are still valid and need not be further modified. A simple example is the linked list which in the purely functional variant does not allow updates of the next pointers other than their initial assignments. Prepending a new head element to the list means to just allocate a new head cell whose next pointer is initialised to the head cell of the old list. Removing the head element means using the old head cell’s next value as the new head cell. A structure which allows removal of elements at arbitrary points or from

---

<sup>37</sup>See Chris Okasaki (1998), *Purely Functional Data Structures*, Cambridge: Cambridge University Press

both ends must be typically organised in a search tree fashion so that on average only  $O(\log n)$  nodes must be recreated in an update.

Purely functional data structures have the attractive property of being automatically fully persistent—changing them does not invalidate any references to previous versions of the structure, and modifications can be carried out given any previous reference.<sup>38</sup> These structures are also beneficial in concurrent and multi threaded environments, because they are automatically thread safe as no mutable state is shared.

There are however three problems which make them a less ideal candidate for our endeavour. First, we wish to transactionally encapsulate modifications which requires an eager evaluation. Second, because we already use an STM, the thread safety argument becomes unimportant compared to the generally more easy and performative implementation allowing directly mutable fields. Third, the path copying involves more “garbage” to be collected which is fine for an in-memory structure, but produces additional overhead and fragmentation when carried out on disk. This does not prevent us from using purely functional structures when the granularity is small enough, for example we use immutable finger trees for the path representation (Sect. 5.6.3) which are serialised and deserialised at a stretch.

The observed trend away from mutable data structures implementing persistence towards purely functional data structures also has to do with the narrow perspective in which persistence is considered useful. Often persistence is just seen as a facilitator of efficient algorithms to erect structures, where the version trace is devoid of any temporal meaning in the sense that we are interested in. Driscoll, Sleator and Tarjan picture the scenario of «high-level languages» where persistence is merely the warrantor of immutability and isolation: A function can be called with a large data structure and is free to mutate that structure as the caller keeps a reference to its own “version snapshot”.<sup>39</sup> The scenario where the transactional temporal dimension allows navigation into the past and the tracing of the evolution of a structure is obliterated. Furthermore,

---

<sup>38</sup>Cf. Haim Kaplan and Robert E. Tarjan (1996), ‘Purely Functional Representations of Catenable Sorted Lists’, in: *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pp. 202–211

<sup>39</sup>Driscoll, Sleator and Tarjan, ‘Fully Persistent Lists with Catenation’, §5.

the main work required for such a navigation is the provision of version stamps, something that is independent of the computational paradigm.

### 5.5.3 The Confluent Case

While Driscoll, Sleator and Tarjan's 1994 paper presented a solution for a singly linked list with pop operation (a stack), it remained an open question whether a general transformation could be found for confluent persistence. In particular, the navigating algorithm of the node copying approach was not suitable for concurrently representing nodes which originate from different versions. An answer to this question was given by A. Fiat and H. Kaplan in 2003.<sup>40</sup>

They closely follow the model and terminology of Driscoll et al., namely the idea of nodes containing data fields (formerly called "information fields") and pointer fields linking to other nodes, as well as query and update operations on these nodes. This paper is not only interesting because it adds support for 'meld' operations—the efficient combination of elements originating from different source versions—but because the authors are aware of the difference between «... a set of derivations involving meld operations» where one only cares «... about some final result» and a situation where indeed navigating in the history is the main goal.

As has been noted, when melding is disallowed—the fully persistent case—the versions form a tree, whereas in a confluent persistent setting the versions can be organised as a directed acyclic graph (DAG). In order to make statements about the performance of the transformations, the notion of the *effective depth*  $e(w)$  of a node  $w$  is introduced. This can be best explained with the example that Fiat and Kaplan use, once more a linked list, the evolution of which is shown in Fig. 5.7.<sup>41</sup> When a node  $w$  (in this case a list cell) is created in version  $i$ , that node along with version  $i$  is called a 'seminal node'  $s(w)$ , because each future version of that node can be traced back to this initial version  $i$ . Such traces form paths in the version graph, which the authors in keeping with the genealogy metaphor call node 'pedigrees'.

In Fig. 5.7, three nodes are created,  $w_0, w_1, w_2$ , the first two originating from version  $v_0$ , the third being created in version  $v_2$ . The depth  $d(w)$  of a node is the longest among all paths

---

<sup>40</sup>Fiat and Kaplan, 'Making data structures confluent persistent'.

<sup>41</sup>The figure was adapted to have more consistent node names as well as better distinguishable element values.

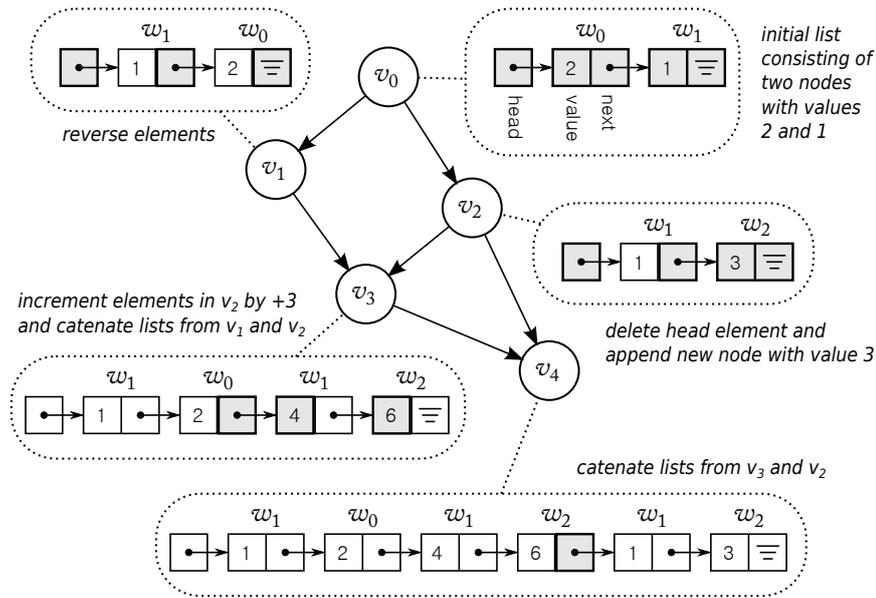


Figure 5.7: Directed acyclic version graph describing the evolution of a linked list with meld in  $v_3$  and  $v_4$ . Fields with bold outline and grey background have been updated in the respective version.

$R(w)$  from any of its occurrences in the graph to its seminal version. The authors show that if one can define an effective compressed representation of these paths, an optimal representation induces a performance and space cost proportional to the logarithm of the number of *different* paths coexisting within one version and not bound by  $d(w)$ . This overhead,  $\log(|R(w)|) + 1$  is the definition of effective depth. When taking the maximum effective depth across all nodes of the graph  $D$ , the result is called the effective depth of the graph  $e(D)$ . With the proper compressed path methods, discussed further down, the space requirement per assignment is bound by  $O(e(D))$ , and a query or update has a time cost of  $O(e(D) + \log \mathcal{U})$  where  $\mathcal{U}$  is the total number of updates.<sup>42</sup>

In the figure, the depth of  $w_0$  and  $w_1$  is 3, whereas the depth of  $w_2$  is 1. Looking at the last version  $v_4$ , there are three distinct paths of “copies” of node  $w_1$ , shown in Fig. 5.8:  $\langle v_0, v_1, v_3, v_4 \rangle$ , having an element value of 1 in version  $v_4$ ;  $\langle v_0, v_2, v_3, v_4 \rangle$ , having a value of 4; and  $\langle v_0, v_2, v_4 \rangle$ , having a value of 1.

<sup>42</sup>An overview of the costs with respect to different implementation methods as well as the “naive” ephemeral perspective—where each version is a full copy of the structure—and fully persistent methods is given in table 1 of Fiat and Kaplan, ‘Making data structures confluent persistent’. The costs can be further improved through randomisation, as discussed in section 5.5.6.

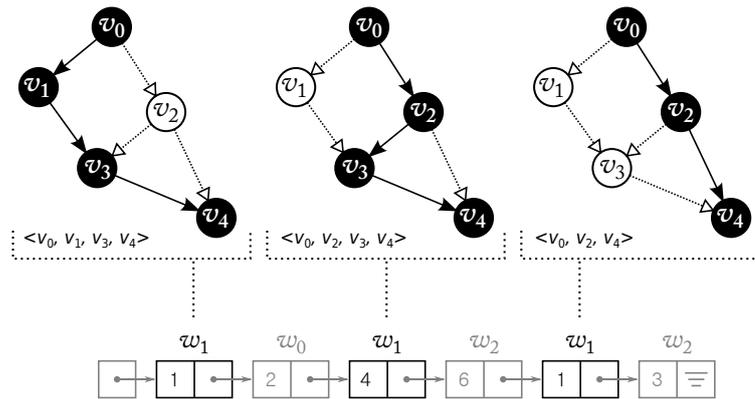


Figure 5.8: In  $v_4$ , there are three instances of  $w_1$  characterised by distinct paths (shown in black) from the seminal version  $v_0$  to  $v_4$ .

What has happened in the melded versions  $v_3$  and  $v_4$  could be described as a *re-entry* of nodes. The re-entered nodes initially share their full structure, but may then be individuated by applying mutations to selected copies of their seminal node. In Fig. 5.8, it can be seen that the versions of  $w_1$  which appear first and last in the final list still share the same element value, whereas the middle copy has evolved in that respect. Also, all three now have distinct next pointers.

If these nodes were to embody musical entities, the meld operation can thus be understood as the introduction of variations—copies of elements are introduced which can still be traced back to the same origin and which are initially identical, but will then be differentiated by their value and neighbourhood. Another possible interpretation is to consider these copies as alternatives between which we can switch back and forth. Filaments in the graph would then correspond for example to different realisations of a random number generator. This is further discussed in Sect. 5.8.2.

#### 5.5.4 Navigating the Graph

From Fig. 5.8 it is clear that an object is fully identified by a tuple  $(p, w_{id})$  consisting of the seminal node identifier  $w_{id}$  and an access path  $p$ . For example, the head element of the final list is  $(\langle v_0, v_1, v_3, v_4 \rangle, w_1)$ , the fourth element in the final list is  $(\langle v_2, v_3, v_4 \rangle, w_2)$ , and the last element is  $(\langle v_2, v_4 \rangle, w_2)$ . It remains to show how such an identifier is stored in a pointer node, such as the list's head field, how the entry of a field corresponding to a given access path is found and

restored. In order to establish this knowledge, we will first describe the abstract algorithm, and later an efficient implementation is presented.

Continuing with the fat node idea of Driscoll et al., all versions of the mutable fields of a node are represented by an augmented variant of the node where these fields store the history of their respective assignments. Let us assume we have already found the example head node and want to retrieve the list cell’s “current” value with respect to the *access path* ending in  $v_4$ . In the monotonically increasing path  $\langle v_0, v_1, v_3, v_4 \rangle$  we must find the last version in which the value field has been updated. Or more formally, with an access path  $q = \langle v_{i_0}, \dots, v_{i_k} = n \rangle$ , find in the augmented structure for the field value the *assignment path*  $p = \langle v_{i_0}, \dots, v_{i_j} = m \rangle$  which has the largest  $m$  which is less than or equal to  $n$ . If we ensure that each element in the set of assignment paths of field  $A$  of a fat node  $f$ —denoted  $P(A, f)$ —begins with the same version  $v_{i_0}$ , the node’s seminal version, and if we ensure that the access path  $q$ , too, begins with the node’s seminal version, then the first step in field retrieval equates to finding the *longest common prefix* of  $p$  and  $q$ . A crucial component of the implementation is thus an efficient algorithm for solving the so-called “pedigree maximum prefix problem”.

Where Fiat and Kaplan use the distinction between “data” and “pointer” field, we shall now use instead the notion of *immutable* and *mutable* objects. From an object-oriented polymorphic perspective we will see that this makes more sense, because the difference only becomes important in deserialisation and the objects remain opaque for the data structure that contains them. That is to say, a data structure does not need to “know” if it contains an immutable or mutable object, because otherwise we could not have generic data structures, but would need to devise a separate container for immutable and mutable objects.

Immutable objects can be deserialised without knowing the context of their access (their identity cannot be split by branching in the version graph), therefore accessing fields carrying immutable data is simple—once the maximum prefix is determined, one finds in the augmented field structure the datum stored at this prefix, and the query is complete. The following table lists the

entries in this augmented structure—perhaps it is implemented as a Trie where the keys are the assignments paths<sup>43</sup>—for node  $w_2$ :

Assignment path	Value
$\langle v_2 \rangle$	3
$\langle v_2, v_3 \rangle$	6

The two occurrences of  $w_2$  in the final list have access paths  $\langle v_2, v_3, v_4 \rangle$  and  $\langle v_2, v_4 \rangle$ . In the first case, the longest prefix of the access path with the two assignment paths of the preceding table is  $\langle v_2, v_3 \rangle$ , resolving to value 6, in the second case the common prefix is just  $\langle v_2 \rangle$  and value 3 is found.

The head field and each cell's next field store references to mutable objects (other list cells). They are also represented by fat nodes and require identification with an access path. Upon storage, their access path ends in the version in which the storage is performed. In the retrieval which generally happens in an access path  $q = \langle v_{i_0}, \dots, v_{i_k} = n \rangle$  extending the most recent storage (assignment) path  $p = \langle v_{i_0}, \dots, v_{i_j} = m \rangle$ , the stored object must be transformed into the current context. For example, let  $x$  be the node containing the mutable field which is queried, and  $y$  be the node which is found in that field at assignment path  $p$ , along with  $y$ 's own access path  $r$  at the time of storage. In the version of that assignment,  $m$ , by definition  $r$  must also end in  $m$ . In order to bring  $r$  up to date, it must be extended by the remaining path leading from the version following the assignment path,  $v_{i_{j+1}}$ , to the current version  $v_{i_k}$  which is also the last element of  $q$ . Or put the other way around, the prefix  $\langle v_{i_0}, \dots, v_{i_j} \rangle$  of  $q$  must be replaced by the stored path  $r$ , giving rise to the transformation's name "pedigree prefix substitution".

It will become clear how this works when traversing the list in version  $v_4$ . The starting point for the linked list is the head pointer. In general, there needs to be at least one entry point to the data structure. These entry points are called access pointers by Fiat and Kaplan. We drop however their requirement of updating the access pointers in each new version, as can be seen in Fig. 5.7: the head element is initialised in version  $v_0$  and updated in  $v_1$  and  $v_2$ , but not in  $v_3$  or  $v_4$ . This

<sup>43</sup>In the first iteration of our implementation which stored the full paths, we employed the Splay Tree which can be used as an elegant and self-balancing Trie structure. See Daniel D. Sleator and Robert E. Tarjan (1985), 'Self-Adjusting Binary Search Trees', *Journal of the ACM (JACM)* 32(3), pp. 652–686

<b>head</b>		<b>w<sub>0</sub> next</b>	
Assignment path	Value	Assignment path	Value
$\langle v_0 \rangle$	$(\langle v_0 \rangle, w_0)$	$\langle v_0 \rangle$	$(\langle v_0 \rangle, w_1)$
$\langle v_0, v_1 \rangle$	$(\langle v_0, v_1 \rangle, w_1)$	$\langle v_0, v_1 \rangle$	None
$\langle v_0, v_2 \rangle$	$(\langle v_0, v_2 \rangle, w_1)$	$\langle v_0, v_1, v_3 \rangle$	$(\langle v_0, v_2, v_3 \rangle, w_1)$

<b>w<sub>1</sub> next</b>		<b>w<sub>2</sub> next</b>	
Assignment path	Value	Assignment path	Value
$\langle v_0 \rangle$	None	$\langle v_2 \rangle$	None
$\langle v_0, v_1 \rangle$	$(\langle v_0, v_1 \rangle, w_0)$	$\langle v_2, v_3, v_4 \rangle$	$(\langle v_0, v_2, v_4 \rangle, w_1)$
$\langle v_0, v_2 \rangle$	$(\langle v_2 \rangle, w_2)$		

Table 5.2: Fat field entries for the head and head fields

gives access pointers the same interface as regular fields, although in the retrieval algorithm the latter are implicitly associated with the path through which they were accessed, whereas the former are associated with the path of the cursor which is used to navigate through the version graph. Another advantage of this approach—not requiring maintenance of an a priori known set of access pointers—is the possibility of generating auxiliary entry points to any part of the data structure, something needed for modular callbacks, as is shown in Sect. 5.8.2.

To follow the traversal, it is useful to look at the values stored in the augmented fields of head and next, shown in Table 5.2. The longest prefix of  $q = \langle v_0, v_1, v_3, v_4 \rangle$  and  $P(\text{head})$  is  $\langle v_0, v_1 \rangle$ , where  $(\langle v_0, v_1 \rangle, w_1)$  is stored, which is expanded to  $(\langle v_0, v_1, v_3, v_4 \rangle, w_1)$  in the prefix substitution.<sup>44</sup> Looking up next in this node yields longest prefix  $\langle v_0, v_1 \rangle$  where  $(\langle v_0, v_1 \rangle, w_0)$  is stored, which is expanded to  $(\langle v_0, v_1, v_3, v_4 \rangle, w_0)$ . Looking up next in this node yields longest prefix  $\langle v_0, v_1, v_3 \rangle$  where  $(\langle v_0, v_2, v_3 \rangle, w_1)$  is stored, which is expanded to  $(\langle v_0, v_2, v_3, v_4 \rangle, w_1)$ . Looking up next in this node yields longest prefix  $\langle v_0, v_2 \rangle$  where  $(\langle v_2 \rangle, w_2)$  is stored, which is expanded to  $(\langle v_2, v_3, v_4 \rangle, w_2)$ . And so forth, until the entire sequence of Fig. 5.8 is produced.

### 5.5.5 Compressing Paths

The direct implementation of the paths and operations presented in the previous section does not scale for obvious reasons: With each operation performed on the system, the access paths and any newly added assignment paths grow steadily. The system would slow down linearly with its

<sup>44</sup>The first two items of  $q$  are dropped here and replaced by the stored path. It so happens that the prefix  $\langle v_0, v_1 \rangle$  is replaced by that same prefix. The lookup of  $w_2$  is an example where indeed a different prefix is substituted.

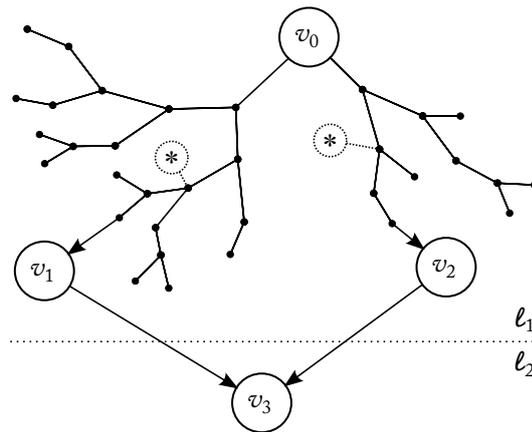


Figure 5.9: Illustration of path compression: The path between  $v_0$  and  $v_1$  and the path between  $v_0$  and  $v_2$  is unique, whereas there are two possibilities to get from  $v_0$  to  $v_3$ .

usage; even more than linearly if we consider the durability layer. Fiat and Kaplan address this by applying a compression mechanism to the paths which is particularly efficient when no meld operations are performed—the fully persistent case—and compressed paths have constant size, corresponding with the effective graph depth  $e(D) = 1$ .

The underlying idea is to reduce the path information to the minimum required to distinguish two copies of a seminal node. For example, if there is no melding, it suffices to remember the seminal version of a node and the most current version, as both form end points of a unique path in the version graph. Fig. 5.9 illustrates this: If one looks at node  $w_1$  of the previous example and imagines that  $v_1$  and  $v_2$  were not direct successors of  $v_0$  but separated from it by a path of intermittent versions, then  $v_0$  can be seen as the root of two sub-trees, one of which contains  $v_1$  as a leaf, and one of which contains  $v_2$  as a leaf. We assume that the element value of  $w_1$  has been modified along either path from  $v_0$ , which is marked by an asterisk in the figure. It is easy to see now that given the information that  $\langle v_0, \dots, v_1 \rangle$  is an access path and  $\langle v_0, \dots, v_2 \rangle$  another access path, leaving out the steps in between, this compressed path is sufficient to find the current element value of  $w_1$  in either of the versions. The retrieval reduces to the question: “Find the nearest marked ancestor of  $v_1$  and  $v_2$  respectively towards their parent  $v_0$ .”

This is the formulation of a general problem in graph theory. A survey about ‘Marked Ancestor Problems’ shows the previous research done in this area and the multitude of application areas.

The problem is known under various names, for example in the “tree colour problem”, the nodes of a tree may be associated with a set of colours, and the problem is, given a node  $v$ , to <find the first ancestor of  $v$  with colour  $c$ >. <sup>45</sup> In our case, the set of colours is the set of mutable fields in a node, and a “mark” means that a value has been assigned to a field.

In a meld situation, version  $v_3$  in Fig. 5.9, an ambiguity arises: The re-entry of  $w_1$  means that  $\langle v_0, \dots, v_3 \rangle$  is no longer a unique path in the version graph. A solution is to introduce the last parent version of  $v_3$  as additional information to the compressed path. As a result, the compressed path can be seen as the representation of the version graph decomposed into disjoint (sub-)trees. Any path in the version graph can enter each sub-tree at most once, this follows from the acyclic nature of the graph. The sufficient information per sub-tree  $T_i$  is the version vertex  $e_i$  at which the tree is entered, and the version vertex  $t_i$  at which the sub-path in  $T_i$  terminates, before moving on to another sub-tree.

The compressed path  $c(p)$  is thus formally a list of paired components  $\langle e_0, t_0, e_1, t_1, \dots, e_j, t_j \rangle$  which correspond to a subset of the versions of the full path  $p = \langle v_{i_0} = e_0, v_{i_1} \dots v_{i_k} = t_j \rangle$ , with  $2k \geq j$ . The compressed path always has an even number of elements, where the entering and terminating vertices are possibly identical, a case that can happen when melding in successive versions or when the path ends with a melded version. For example, traversing the list in  $v_4$  of Fig. 5.7 would find the following accessed nodes given in compressed form  $(c(p), w)$ :

$$\begin{aligned} &(\langle v_0, v_1, v_3, v_4 \rangle, w_1) \rightarrow (\langle v_0, v_1, v_3, v_4 \rangle, w_0) \rightarrow (\langle v_0, v_2, v_3, v_4 \rangle, w_1) \rightarrow \\ &(\langle v_2, v_2, v_3, v_4 \rangle, w_2) \rightarrow (\langle v_0, v_2, v_4, v_4 \rangle, w_1) \rightarrow (\langle v_2, v_2, v_4, v_4 \rangle, w_2) \end{aligned}$$

In particular it must be noted that not every meld operation introduces a new sub-tree. The partitioning of sub-trees is best understood with the level function  $\ell(T)$  which assigns an integer to each sub-tree. The root of the version graph is at level  $\ell_0$ . For every new version vertex, we look at the set of incoming version vertices and determine the highest level  $\ell_{\max}$  among the trees to which they belong. If there is more than one vertex leaving from that highest level, the new vertex lies on a tree with increased level  $\ell_{\max+1}$ . Therefore, the creation of  $v_3$  requires a new

<sup>45</sup>Stephen Alstrup, Thore Husfeldt and Theis Rauhe (1998), ‘Marked Ancestor Problems’, in: *Proceedings of the 39th Annual Symposium on Foundations of Computer Science*, IEEE, pp. 534–543.

level, but not the creation of  $v_4$  because the input versions  $v_3$  and  $v_2$  do not share the same level. The introduction of the new tree level in  $v_3$  is shown as a dotted line in Fig. 5.9.

The retrieval algorithm is similar to the full path scenario. First the longest prefix of the compressed access path's so-called *index*  $\tilde{c}(q) = \langle e_0, t_0, \dots, e_k = n \rangle$  with the set of compressed assignment path *indices*  $\tilde{C}(p)$  is determined. The result is an index  $\tilde{c}(p) = \langle e_0, t_0, \dots, e_j = m \rangle$  leading to the sub-tree associated with  $m$ . For this sub-tree, a search structure is maintained that can answer the nearest marked ancestor question with respect to the terminating version  $t_j$  which follows the prefix  $\tilde{c}(p)$  in the access path. If the retrieved value is a mutable entity, again a prefix substitution is performed upon deserialisation, this time taking care of the pairing of the compressed representation.<sup>46</sup>

To give at least the simple example here of retrieving an immutable value, let us reconsider the element value for node  $w_2$  as in section 5.5.4, but this time with the compressed path method.

The assignment table looks as follows:

Assignment path	Value
$\langle v_2, v_2 \rangle$	3
$\langle v_2, v_2, v_3, v_3 \rangle$	6

The two occurrences of  $w_2$  in the final list have compressed access paths  $\langle v_2, v_2, v_3, v_4 \rangle$  and  $\langle v_2, v_2, v_4, v_4 \rangle$ . In the first case, the longest prefix of the access path index with the two assignment path indices of the preceding table is  $\langle v_2, v_2, v_3 \rangle$ . In the search structure for the value field in the sub-tree containing  $v_3$ , the nearest marked ancestor is vertex  $v_3$ , resolving to value 6, in the second case the common prefix is just  $\langle v_2 \rangle$  and value 3 is found.

### 5.5.6 Adding Randomisation

The final suggestion regarding the original algorithm is a further compacting of paths through a randomisation of version identifiers. The idea is to associate each version with a random integer number such that with high probability the sum of all these numbers is unique for each individual (full or compressed) path. For example, looking back again at the three occurrences of node  $w_1$  in version  $v_4$ , if we use 32-bit signed integers (*Scala's* standard type `Int`) for the

<sup>46</sup>The details can be found in §5.1 and Fig. 8 of Fiat and Kaplan, 'Making data structures confluently persistent'

random numbers, the following table shows the equivalence between the normal compressed paths and the randomised reductions:<sup>47</sup>

Access path	Randomised	Sum
$\langle v_0, v_1, v_3, v_4 \rangle$	$\langle 0x18598416, 0x37297A3D, 0x2828C850, 0x60DEC3BC \rangle$	0xD88A8A5F
$\langle v_0, v_2, v_3, v_4 \rangle$	$\langle 0x18598416, 0x0AE59378, 0x2828C850, 0x60DEC3BC \rangle$	0xAC46A39A
$\langle v_0, v_2, v_4, v_4 \rangle$	$\langle 0x18598416, 0x0AE59378, 0x60DEC3BC, 0x60DEC3BC \rangle$	0xE4FC9F06

The sum of the randomised path components can be seen as a hashing function.<sup>48</sup> Given a well distributed pseudo random generator and a sufficient number of bits, the probability of hash collisions becomes extremely small. For example, if we allocate  $\log_2 R$  bits for the random numbers, and the number of (compressed!) access paths along with all of their prefixes occurring in a structure is  $|\mathcal{P}|$ , then the probability of a collision of at least two hash values is  $O(\frac{|\mathcal{P}|}{2R})$ . In other words, when using 32-bit random numbers, in a structure which contains a set of 10,000 unique access paths and prefixes, a hash collision has a probability of roughly 1 : 1,000,000.

This hashing is only useful inasmuch as it aids the pedigree prefix search. The methods assume an efficient representation of the sequence of randomised versions  $\pi$ . Such a representation can report the sum of the elements in  $O(1)$ , as well as split and concatenate sequences in  $O(\log|\pi|)$ . Fiat and Kaplan use a clever technique for the prefix search based on the binary representation of the sequence length  $|\pi|$ . A hash table  $H$  is maintained so that when a value associated with path  $\pi$  is added—the randomised assignment path—multiple keys point to that value: These keys are the sums of the set of prefixes of  $\pi$  defined as  $\hat{\pi} = \{p_{i_1}(\pi), p_{i_2}(\pi), \dots, p_{i_{m(\pi)}}(\pi) = \pi\}$ , where  $m(\pi)$  is the number of 1's in the binary representation of  $|\pi|$ . The prefix lengths are obtained by successively replacing 1's with 0's in  $m(\pi)$ . For example if the assignment path had length  $|\pi| = 298 = 100101010_2$ , there are  $m(\pi) = 4$  bits of 1 which will be successively removed, so there are prefixes of lengths  $i_4 = 298 = 100101010_2$  (the complete path),  $i_3 = 296 = 100101000_2$ ,  $i_2 = 288 = 100100000_2$ , and  $i_1 = 256 = 100000000_2$ . According to the assumption of sufficiently

<sup>47</sup>When calculating the sums, to avoid overflow we would extend to standard 64-bit integers; in the example table, w.l.o.g. we deliberately choose only positive random numbers below  $2^{31}$  to keep the sums small.

<sup>48</sup>For an overview of hashing, see Donald E. Knuth (1973/1998), *The Art of Computer Programming*, 2nd Edition, vol. 3, Reading, MA: Addison-Wesley, §6.4

small probability of hash collisions, the sums of these four prefixes, which are the keys into  $H$ , are unique.

The longest prefix search in  $H$  is then a very similar procedure: The access path is truncated stepwise as above, by taking prefixes of sizes obtained by removing bits from the binary representation of the access path length, until  $H$  contains an entry for the sum (hash value) of a prefix. As in the normal compressed path method, this way a search structure for a sub-tree of the version graph is found, and a successive nearest marked ancestor query yields the current value of the field for the given access path.

The randomisation of the compressed path method for in-memory data structures produces a speed increase of access time cost from  $O(e(D) + \log U)$  to  $O(\log^3 e(D) \cdot \frac{\log T}{\log U} + \log U)$ , where  $T$  is the total number of field retrievals. In our case, the path components forming a mutable object's identifier still must be stored on disk. As shown in Sect. 5.6.3, we limit the granularity of the data store accesses by conceiving the paths as immutable entities which are always stored completely with the mutable state, so reading or writing a path still involves  $O(e(D))$  words, apparently undoing the advantage of the exponential speedup promised by the randomisation method. However, using the prefix sums as *keys* to the data store significantly simplifies our interface, as now the keys have constant size and are not broken up as in a Trie structure. And so the data store remains an interchangeable black box: the advantage of the key value store is indeed that it is indistinguishable from a hash table.

## 5.6 Building a Confluent System

As others have noted,<sup>49</sup> there is often a gap between theoretically conceived algorithms and data structures and their actual implementation and practical usability. Often the constraints required by the theoretical research do not align with the constraints of the actual systems on which these algorithms are to be implemented. For the theoretical researcher it is sufficient to proof the theorems which guarantee the correctness and performance bounds of the algorithms. A software developer, on the other hand, has only limited resources for engaging with the

---

<sup>49</sup>Frédéric Pluquet (2012), 'Efficient Object Versioning for Object-Oriented Languages from Model to Language Integration', PhD thesis, Brussels: Université Libre de Bruxelles, pp. 1–4.

theoretical discourse and often resorts to sub optimal or even brute force implementations in order to solve a problem.

From the previous discourse of this thesis it has become clear that algorithms need a material embodiment if they are to be made productive for creative use. No implementation is straightforward, and there is no unique mapping between algorithm and implementation. The implementation is itself a true process which produces traces and differences which shift the discourse into directions previously unforeseen, and may trigger new interesting questions.

Even before we engage with our own extensions of the confluently persistent approach, the incorporation of quasi-retroactive elements, the integration with a reactive even system, the augmentation with bi-temporal semantics or the coupling with realtime sound synthesis, we wish to emphasise that the core implementation of the confluent persistence, backed by a durable data store, is already a major contribution of this thesis.

### 5.6.1 Related Work

To the best of our knowledge, there existed no general implementation prior to our own. Of course, software versioning systems are ubiquitous now, and they all—*Subversion*, *Git* etc.—provide some mechanisms of merging different branches in the code base. In that respect, they can be called confluent systems. On the other hand, they operate on the file system level with the assumption of text files which are observed, so the granularity of these systems is on a completely different scale than what we are doing in our system: The augmentation of any program *data structure* with a tracing mechanism, an automatic online algorithm with selectable temporal semantics which allows effective querying and combination of the history of each mutable state.

The closest approach we found is a library called *HistOOry* which provides persistence for the *Smalltalk* language.<sup>50</sup> One of the use case scenarios of this system is the observation of legacy systems in order to understand their workings. With *HistOOry*, objects and fields in the object language can be selectively augmented to record a trace of their modification. The

---

<sup>50</sup>Frédéric Pluquet, Stefan Langerman and Roel Wuyts (2009), ‘Executing code in the past: efficient in-memory object graph versioning’, in: *ACM SIGPLAN Notices*, vol. 44, 10, pp. 391–408.

system restricts itself to partial persistence and generally seems to aim at system analysis, such as debugging a program and reviewing the past states of a mutable entity, although cases are also presented which use persistence purely for the design of particular algorithms such as the geometric problem of planar point location.<sup>51</sup> In the more recent thesis of the author, it seems that “branched versioning” (full persistence) has been added to the library, although it is unclear in what sense branching is actually used in the presented case studies.<sup>52</sup> More than the missing melding capabilities, the restriction to in-memory recording only makes this approach unsuitable for a composition framework which must operate across long time spans where the data is stored on hard disk and structure may grow beyond the available RAM.

### 5.6.2 Transaction Cycle

The transaction cycle is slightly more involved than in the ephemeral durable case which was shown in Fig. 5.6, because we want transactions to compose nicely and transparently. It implies that we do not know at the beginning of a transaction whether it will contain updates or version melds, and with the compressed path method it cannot be determined until the commit phase whether a new sub-tree in the version graph must be created or not. The solution is to buffer the write operations for an ongoing transaction in an in-memory cache. Furthermore, the ongoing transaction can only operate based on its *input access path*, and mutable objects retrieved in a transaction are actualised (using the pedigree prefix substitution) with this input access path—precisely because it is not known whether the transaction will constitute a new version node or not. When the buffered write operations are flushed at the end of the transaction, the transaction’s *write version* is known and will be part of the data store’s *key*, but not the opaque *values*. Special care must therefore be taken to correctly append the missing information in deserialisation.

This is best illustrated by going through a brief example. Using the linked list structure previously introduced (Fig. 5.4a), let us assume that there is a given factory to create a new empty

---

<sup>51</sup>Cf. Neil Sarnak and Robert E. Tarjan (1986), ‘Planar Point Location Using Persistent Search Trees’, *Communications of the ACM* **29**(7), pp. 669–679

<sup>52</sup>Pluquet, ‘Efficient Object Versioning for Object-Oriented Languages from Model to Language Integration’; although the author says the implementation is “inspired” by Driscoll et al., it seems that no automatic persistent evolution is used, but instead an explicit “snapshot” taking performed when a new branch in time is created, cf. §3.6.3 and §4.3.11.

```

object LinkedList {
  def apply[S <: Sys[S], A]()(implicit tx: S#Tx): LinkedList[S, A] = ...
}
trait LinkedList[S <: Sys[S], A] {
  def cell(init: A)(implicit tx: S#Tx): Cell
  ...
}

```

Listing 5.7: Factory methods for instantiating a list and its cells

```

val store      = BerkeleyDB.tmp()
val s          = Confluent(store)
val (access, cursor) = s.cursorRoot { implicit tx =>
  val list      = LinkedList[Confluent, Int]()
  val w0       = list.cell(init = 2)
  val w1       = list.cell(init = 1)
  list.head() = Some(w0)           // (1)
  w0.next()   = Some(w1)           // (2)
  list
} { implicit tx => _ => tx.newCursor() }

```

Listing 5.8: Creating a confluently persistent system initialised with a list of two cells

linked list, and the list itself has an extra method for creating new cells, as shown in Listing 5.7.<sup>53</sup> Creating the the first version of Fig. 5.7 looks like Listing 5.8. A store is opened in a temporary directory, the confluent system is selected and the data structure initialised. The cursorRoot method takes two functions: the first creates the initial access point to the data structure and is only executed if the database is empty (otherwise, the previous state is read). The second function is used to initialise a cursor to navigate the version graph.

<sup>53</sup>This is slightly simplified by leaving out the implicit serialiser for the list element type A

<b>Main Cache</b> : Map[Int, Map[Long, (S#Acc, Any)]]		
Seminal	Object Cache	
$id_{\text{head}}$	Path Hash	Value
	$\sum(\pi_{\text{head}} = \langle \rangle)$	$(\pi_{\text{head}}, \text{Some}(\langle \rangle, w_0))$
$id_{w_0 \text{ next}}$	Path Hash	Value
	$\sum(\pi_{w_0 \text{ next}} = \langle \rangle)$	$(\pi_{w_0 \text{ next}}, \text{Some}(\langle \rangle, w_1))$

Table 5.3: Transaction local cache for ongoing write operations

When the new linked list instance `list` is created, its access path is empty  $\langle \rangle$ —this follows from the above explanation—and so is the the access path of both cells  $w_0$  and  $w_1$ . The assignments of the list head and of the `next` field of node  $w_0$ <sup>54</sup> are write operations which are buffered, so that the paths can be completed upon transaction commit. The two lines marked (1) and (2) produce the cache entries of Table 5.3.

The cache is a nested map. The outer map, the main cache, uses the seminal identifiers of mutable entities (variables) as keys, and the values are the inner cache maps which use the hash code (sum) of the assignment paths as keys and the values correspond to the assigned values. In a field retrieval, we can simply check the main cache for the field’s identifier and if it exists in the cache, the sum of the field’s access path is directly used as key into the inner map. No prefix search is necessary, because the cache per definition only contains values updated in the current transaction. If the key is not found, the regular retrieval with longest prefix search is performed on the durable store.

In the example of Table 5.3 the keys of the inner maps are zero because they are sums of empty access paths. When the transaction is closed, the flushing algorithm traverses the cache and appends the write version to these keys. In this case, they would become  $\langle v_0, v_0 \rangle$ . In general the paths are not empty, and then a write version which does not begin a new version graph sub-tree *replaces* the last terminating version. For example, considering the evolution of Fig. 5.7, in version  $v_1$  which reverses the list, the *input access path* is  $\langle v_0, v_0 \rangle$ , so this path will be found in the cache for the updates to the list head pointer and the next pointers of both nodes. Upon commit, as we stay on the same sub-tree, “appending” the new write version  $v_1$  yields an updated access path of  $\langle v_0, v_1 \rangle$ . In the case of entering a new sub-tree, the new write version is appended twice (as entering and terminating vertex). For example, in  $v_3$  the first cell has input access  $\langle v_0, v_1 \rangle$ . The flush algorithm will determine that a new sub-tree is needed as result of the meld, and this path will be updated to  $\langle v_0, v_1, v_3, v_3 \rangle$ .

As said, no object in value position is updated in the flush phase. So the fat node for  $w_1$  will contain in its `next` field an entry pointing from  $\langle v_0, v_1, v_3, v_3 \rangle$  to  $(\langle v_0, v_1 \rangle, w_0)$ . When retrieving

---

<sup>54</sup>As well as the implied initial assignments such as the cell’s element values which are not taken into account here

```

trait Measure[-A, V] {
  def zero: V
  def |+|(a:V,b:V): V
  def apply(e: A): V
}
      (a)

object Index
  extends Measure[Any, Int] {
  def zero = 0
  def |+|(a:Int,b:Int) = a+b
  def apply(e: Any) = 1
}
      (b)

object Sum
  extends Measure[Int, Int] {
  def zero = 0
  def |+|(a:Int,b:Int) = a+b
  def apply(e: Int) = e
}
      (c)

```

Figure 5.10: Measure type class for Finger Trees, and two example implementations

this value, the access path of  $w_0$  must be updated first to match the suffix of the write path (key), before applying the pedigree prefix substitution.

### 5.6.3 Path Encoding

The data structure used to represent paths must be efficient with respect to splitting, concatenation, and summing any prefix of the path. A purely functional data structure which performs well in all these cases is the Finger Tree.<sup>55</sup> Recalling Sect. 5.5.2, this means that update operations leave the input structure unmodified, and instead of directly mutating the cells, a new object is produced which shares part of the former structure in order to minimise the cost of such operations. The Finger Tree provides amortised constant costs for prepending and appending elements, while splitting and concatenation take  $O(\log n)$ ,  $n$  being the smaller of the two paths concatenated or the two paths resulting from a split.

What makes the Finger Tree interesting is that it equips a sequence with an *annotation* which “measures” the elements of the sequence. The measurement of type  $V$  is taken by a monoid and a function which gives a single measure when applied to one element of type  $A$  in the sequence of the tree, as shown in Fig. 5.10a.

The monoid’s identity value  $\emptyset$  (in code written as `zero`) provides a meaningful measurement of an empty sequence, whereas the `||·||` function (in code written as `apply`) is the starting point for a non-empty sequence. Any other measurement is then derived by successively combining measurements with the monoid’s binary operation ( $\oplus$ ) (in code `|+|`). In order to efficiently combine measurements, this binary operation must be associative (but need not be commutative).

<sup>55</sup>Ralf Hinze and Ross Paterson (2006), ‘Finger trees: a simple general-purpose data structure’, *Journal of Functional Programming* **16**(2), pp. 197–217.

R. Hinze and R. Paterson show the versatility of this approach by providing measurements that produce a random access sequence—measuring the indices of elements—a priority queue—measuring the maximum element value—an ordered sequence—tracing the right most element in a subsequence—and even a spatial data structure: an interval tree which allows spatial intersection queries. Another strong point of the Finger Tree is the possibility of maintaining multiple independent measurements.

The solution of having both random access and efficient computation of partial sums is based on this possibility of combining measurements. We use the indexing scheme from Hinze and Paterson, the implementation of which is shown in Fig. 5.10b, and add a summation scheme, shown in Fig. 5.10c. The first counts the number of elements, where an empty sequence obviously has a measurement of 0, and the measurement of one element gives size 1, independent of the element type. The second operates on the elements themselves, by summing them up. Again an empty sequence has sum 0, whereas the sum of a single element  $e$  is equal to the element itself.

To avoid overflow, we would use a 64-bit Long for the summation. Then, to combine indexing and summation, the measurement type  $V$  is a tuple  $(Int, Long)$ , and  $\emptyset$ ,  $(\oplus)$  and  $\|\cdot\|$  simply treat the two tuple elements independently. In order to have a convenient user representation of paths and also in order to quickly determine the temporal order of versions, the elements of the path are themselves tuples  $(v_i, \pi_i)$  combining the non-randomised and the randomised version indices.

Examples for the compressed path  $\langle v_0, v_1, v_3, v_4 \rangle$  (cf. Fig. 5.8) and its hypothetical meld into extended compressed path  $\langle v_0, v_1, v_3, v_4, v_5, v_5 \rangle$  are shown in Fig. 5.11. A Finger Tree is an instance of either `Empty`, `Single`, or `Deep`. The deep tree consists of three parts: a prefix, a pointer to a sub-tree in the middle, and a suffix. Prefix and suffix are represented by the `Digit` type which is a container for up to four elements—these containers are called `One`, `Two`, `Three`, and `Four`. When a fourth element is to be added to a full prefix or suffix, the digit is split and part moved into the middle sub-tree. In the example expansion, the first addition of  $(v_5, \pi_5)$  changes the suffix `Three` (Fig. 5.11a) to a `Four`, the second addition causes the split into the new suffix

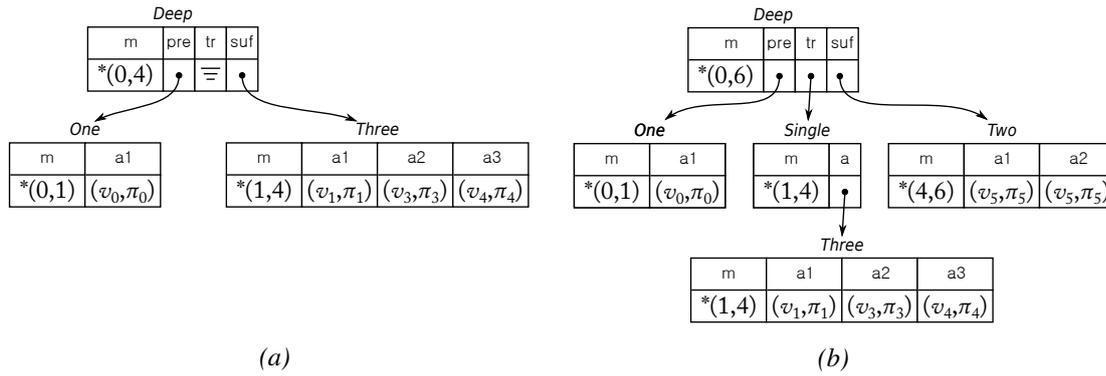


Figure 5.11: Finger Trees representing compressed paths

Two and a preceding Three which is stored in the root tree's middle field, which is promoted from Empty to Single (Fig. 5.11b).

The measurements are cached in each digit and sub-tree which makes the look up fast. They are notated in the figure using a function  $*$  defined as follows:

$$*(m, n) = \left( \underbrace{n-m}_{\text{index}}, \underbrace{\sum_{k=m}^{n-1} \pi_{i_k}}_{\text{sum}} \right)$$

To determine the hash code (sum) of the path's prefix of length 3,  $\langle v_0, v_1, v_3 \rangle$ , in Fig. 5.11b one would look for the digit in which the predicate  $(\text{init} \oplus m) \dots 1 > 3$  becomes true, and combine the partial measurements  $\text{init}$  on the way. The root's prefix  $m$  has index count 1 and sum  $\pi_0$ , so we set  $\text{init} := (1, \pi_0)$  and continue to examine the middle tree. It has an index count of 3 which gives an accumulated index measurement of 4, satisfying the predicate. We examine the *Single*'s only digit by incrementally calculating the measurement to find the exact position at which the predicate becomes true. The result is the partial sum  $(\text{init} \oplus \|(v_1, \pi_1)\| \oplus \|(v_3, \pi_3)\|) \dots 2$ .

The Finger Tree is a recursively defined structure: While the prefix and suffix of the root tree have element type  $A$ , the sub-tree hanging off the root *Deep* tree has element type  $\text{Digit}[A]$ . If this sub-tree is deep, its own middle sub-tree entry points to a tree of type  $\text{Digit}[\text{Digit}[A]]$ , and so forth, and a tree with  $n$  elements will have  $\log n$  levels. Then a partial sum calculation is performed in  $O(\log n)$ , too. Using the bit erasure method to calculate the partial prefixes from

the hash codes, it thus takes  $O(\log^2 |\pi|)$  hash table lookups to find the longest prefix of a given access path at which a value was stored in a fat field.

#### 5.6.4 Nearest Marked Ancestor

Having introduced the path representation based on compression and randomisation, where sequences of unique sums are stored as Finger Trees, it remains to find a data structure to efficiently query the nearest marked ancestor in the sub-trees into which the version graph is partitioned. Remember that with the hashing technique from the last sections, we find the longest prefix  $\tilde{c}(p)$  of the access path *index*  $\tilde{c}(q)$  pointing to the tree in which a value has been assigned to a field. In the data store at the key given by the field node's seminal identifier along with the hash code of  $\tilde{c}(p)$ , a yet to be defined search structure is found which in some form must contain the vertices of the sub-tree in which the queried field has been updated. The sought vertex is the nearest marked ancestor of the terminating vertex which follows in  $c(q)$  after prefix  $\tilde{c}(p)$ .

For example, if the search path  $c(q)$  is  $\langle e_0, t_0, e_1, t_1, \dots, e_k, t_k \rangle$  and the hash-based search using  $\tilde{c}(q)$  yields longest prefix  $\tilde{c}(p) = \langle e_0, t_0, e_1, t_1, \dots, e_j \rangle$  where  $j \leq k$ , we need to answer the question: In the sub-tree corresponding to entering vertex  $e_j$ , which is the nearest ancestor  $x$  of  $t_j$  in which the queried field has been modified?<sup>56</sup>

#### Two Observations

(1) Driscoll et al. describe the ancestor problem in the context of their full persistence approach.<sup>57</sup> Since the version graph in the full persistence case is a tree, this problem can be directly applied to our problem of searching in the index sub-tree. The idea is to turn the partial ordering of the versions in the tree into a total ordering based on some criterion. The idea goes back to P. F. Dietz who designed a data structure for maintaining a linear order.<sup>58</sup> The ancestor problem appears as one application example: If the tree is decomposed into its pre- and post-order traversals, then to determine whether a vertex  $x$  is an ancestor of a vertex  $y$ , one merely has

---

<sup>56</sup>If this is unclear, go back to Sect. 5.5.5

<sup>57</sup>Driscoll et al., 'Making data structures persistent'.

<sup>58</sup>Paul F. Dietz (1982), 'Maintaining order in a linked list', in: *Proceedings of the fourteenth annual ACM symposium on Theory of computing*, pp. 122–127.



(2) A second observation is seemingly trivial: The relative positions of the marked vertices remain the same if all unmarked vertices are removed. If we also keep the query vertex  $u$  in the filtered list at its relative position, the traversals become:

$$\begin{array}{l} \text{pre : } b \ m \ g \ n \ u \\ \quad \quad * \ * \ * \ * \ \leftarrow \\ \text{post : } m \ g \ u \ n \ b \\ \quad \quad * \ * \ \rightarrow \ * \ * \end{array}$$

To find the nearest marked ancestor, again one would look for the nearest vertex of  $u$  to its left in the filtered pre-order and to its right in the filtered post-order list.

### Keeping Order in a List

As Dietz has shown, an order maintenance structure strikes a compromise between the speed at which comparisons between two elements can be made, and the costs of inserting new elements into the list. The basic idea to allow fast comparisons is to assign number labels to the elements which represent their relative order. For example, in the lists above, if we tagged each element with their position in the list, e.g.  $a_{\text{pre}} \rightarrow 0$ ,  $a_{\text{post}} \rightarrow 23$ ,  $n_{\text{pre}} \rightarrow 15$ ,  $n_{\text{post}} \rightarrow 20$ ,  $b_{\text{pre}} \rightarrow 1$ ,  $b_{\text{post}} \rightarrow 22$ ,  $u_{\text{pre}} \rightarrow 22$ ,  $u_{\text{post}} \rightarrow 16$ , then we can immediately compare those numbers to determine the relative position of two vertices in either the pre- or post-order traversal.

However, if a new leaf is inserted, for example  $y$  as child of  $e$ , there is a problem: All the elements after  $y$  in either of the two lists change their indices and would need relabelling. A common idea is thus to try to minimise relabelling by allowing sufficient space between the labels. If a label was a 32-bit integer number, to represent the initial 24 elements, the labels might be spaced by  $(2^{32} - 1)/24$ . Since the evolution of the tree is not known in advance, a simple strategy is, when an entry  $y$  is inserted between  $x$  and  $z$  in the total order, to attach a label  $\text{lb}(y) := (\text{lb}(x) + \text{lb}(z))/2$ . In the case of  $\text{lb}(z) - \text{lb}(x) = 1$ , only a number of neighbouring entries will be relabelled, until the label density has become smaller again.

Several people have proposed algorithms for maintaining an order structure at improved costs. For example, P. F. Dietz and D. D. Sleator proposed a structure which allows insertions and

comparisons at the optimum of  $O(1)$  worst case.<sup>59</sup> However the structure is quite involved—it requires several types of nodes and different algorithms must be decomposed into piecewise operation to achieve the bounds—and in reply M. A. Bender et al. developed a simpler structure.<sup>60</sup> While it only guarantees  $O(\log n)$  insertion costs, unlike Dietz’s original proposal, it does not require any explicit tree structure, but operates entirely on a plain linked list. The advantage for us is that if this linked list is made durable similarly to what has been shown in Fig. 5.5—but with bi-directional links—the space needed per version vertex is rather small: its own cell identifiers and the identifiers of the successor, predecessor and tag variables, once for the pre-order and once for the post-order entry.

The dynamic insertion in the order structure follows the proposal by Driscoll et al.: For each version, two objects representing the pre- and post-order entry are created and inserted into *the same list*. For example, for version  $b$  there are two entries,  $b_{pre}$  and  $b_{post}$ . When creating version  $c$  as child of  $b$ , insert  $c_{pre}$  after the parent’s pre-order entry,  $b_{pre}$ , then insert  $c_{post}$  after  $c_{pre}$ . When one filters the resulting list to contain only the pre-order or only the post-order entries, the result is as follows:

$$\begin{array}{cccccccccccccccccccc} \text{pre:} & a & b & f & n & s & v & t & u & o & p & r & q & g & j & c & e & m & w & k & l & d & h & x & i \\ & & * & & * & & & & \leftarrow & & & & * & & * & & & & & & & & & & & \\ \text{post:} & v & u & t & s & r & q & p & o & n & j & g & f & w & m & l & k & e & x & i & h & d & c & b & a \\ & & \rightarrow & & & & & & & * & * & & * & & & & & & & & & & * & & \end{array}$$

While the previously shown lists were obtained by following the tree in ascending lexicographic order, this algorithm corresponds to a traversal in descending order. One is the 180° rotated image of the other, and both are valid. It can be quickly verified that the above lists indeed correctly represent the ancestor relationships, and  $n$  is still the nearest marked ancestor of  $u$ .

### Ancestor Search as a Geometrical Problem

Unfortunately, being able to query whether one version is an ancestor of another version is not enough. As B. Salzberg and D. Lomet note, being an ancestor is only a necessary but not a

<sup>59</sup>Paul F. Dietz and Daniel D. Sleator (1987), ‘Two Algorithms for Maintaining Order in a List’, in: *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pp. 365–372.

<sup>60</sup>Michael A. Bender et al. (2002), ‘Two Simplified Algorithms for Maintaining Order in a List’, *Lecture Notes in Computer Science* **2461**, pp. 152–164.

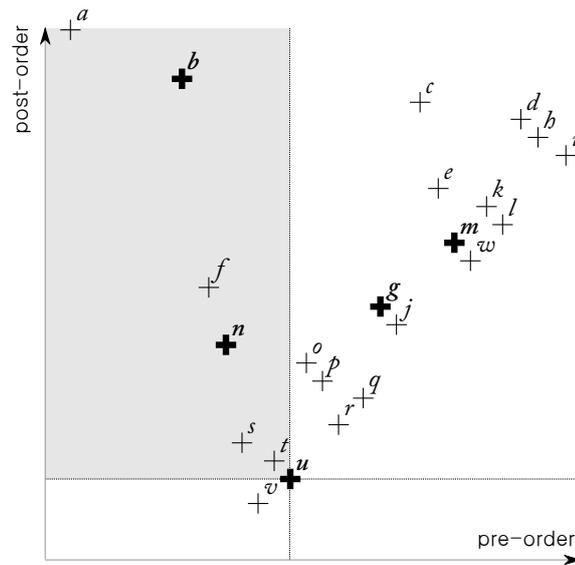


Figure 5.13: Interpreting ancestor lookup as a two-dimensional nearest neighbour search. Centring the coordinate axes around the query point  $u$ , only points within quadrant II (shown in grey) are considered.

sufficient criterion for finding the correct assignment path.<sup>61</sup> Although they devise their own algorithm, they acknowledge that it is only suited for the special case where the number of version branches is small.

Our approach is based on a simple idea: If the pre- and post-order entries are viewed as two orthogonal axes of a plane, we first locate the point corresponding to the query vertex. The plane is divided into four quadrants by this point, and the nearest marked ancestor is then found through a nearest neighbour search in the second of the thus established quadrants (“north-west”). This is illustrated in Fig. 5.13.

Computational geometry has developed many data structures which allow spatial search in two or more dimensions. Some of them, like the R-Tree, have already been mentioned in Sect. 2.6.1 because they can be used to represent bi-temporal data. There are, however, two problems which prohibit a straight forward implementation. Both concern the stability of the points.

First of all, because all spatial data structures, just as a one-dimensional binary search tree, subsequently divide the space along a path of internal nodes, the relabelling in the order main-

<sup>61</sup>Betty Salzberg and David Lomet (1995), *Branched and Temporal Index Structures*, tech. rep. NU-CCS-95-17, Boston: Northeastern University, §4.1.

tenance structure means that the points move inside the plane and invalidate the labels and organisation of these internal nodes, too. This can be solved by removing the affected points prior to their relabelling, and reinserting them with their new labels. Assuming that the spatial insertion just like the relabelling is an  $O(\log n)$  operation, this is technically feasible at the price of degrading to  $O(\log^2 n)$  amortised costs for creating a new vertex.

The second problem arises in the relationship between marked and unmarked vertices. Because we can have any number of mutable fields, in a field query it must be possible to isolate those vertices which are contained in the set of assignment paths for that field. Looking at Fig. 5.13, it is not sufficient to find *any* nearest neighbour of  $u$ —that would be  $t$  here—but it must be a nearest neighbour after removing all unmarked points. A naive idea is to add a third dimension across which the seminal field identifiers are distributed—then we can constrain the search to the slice which manifests assignments in that field. But this would mean that in the case of relabelling, the number of points that need reinsertion will depend not just on the total number of versions in the graph, but also on the number of assignments in each version which needs relabelling. We would need to store all those points in another structure associated with the main tree vertex. Also there is no meaning in the extra spatial dimension—for example in terms of neighbourhood—other than distinguishing unordered elements. Clearly, this is not feasible.

In the regular fat field approach as discussed in the papers about full and confluent persistence, there is no global structure of modifications, but only the local set of assignments are managed. If we follow this approach, the problem manifests itself in multiple order maintenance structures being independently constructed: For a version  $v$ , there will be pre- and post-order entries  $v_{\text{pre}}$  and  $v_{\text{post}}$  in the global version list, but if a field  $F$  is mutated in that version, it will add such entries  $v'_{\text{pre}}$  and  $v'_{\text{post}}$  to its local spatial structure to permit the ancestor search. As soon as a relabelling occurs in either the global version list or any local (marked) version list, these labels diverge.

Luckily, a solution lies in the second of the two observations made earlier on: The *relative positions* within the order maintenance structures still match. The tag list containing the entries of all vertices within a version graph's sub-tree is thus isomorphic to the tag list formed by the

subset of these vertices which filters only those corresponding to assignments for the field  $F$ . What is needed is an extra preparatory step that translates a vertex position from “full tree” to “marked tree”, before querying or updating  $F$ .

Let  $m_i$  be a mark of field  $F$  in version  $i$ , given a previously determined sub-tree of the version graph. The data type `Mark` implementing  $m$  captures the necessary information for this translation. Besides the value  $a_i$  corresponding with the assignment of that mark, it contains a reference both to the full tree vertex position  $v_{full,i}$  and the mark tree vertex position  $v_{mark,i}$ . So  $m_i = (v_{full,i}, v_{mark,i}, a_i)$ . A fat field stores these marks in a binary tree `preOrder`, sorted by the mark tree vertex’ pre-order tag  $v_{mark,pre,i}$ , and in another binary tree `postOrder`, sorted by the mark tree vertex’ post-order tag  $v_{mark,post,i}$ . Then to locate the hypothetical mark tree position  $v_{mark,q}$  of any vertex (marked or not) specified as a position in the full tree  $v_{full,q}$ , one traverses `preOrder` and `postOrder` using the following procedure: Since each node of these trees contains a mark  $m_j$ , we have access to this mark’s full tree vertex. Thus we compare this vertex’ positions  $v_{full,pre,j}$  and  $v_{full,post,j}$  with the query position  $v_{full,pre,q}$  and  $v_{full,post,q}$ . Depending on the comparison, we go down left or right the trees until we find either the query vertex itself or the leaf that is either the greatest—with respect to ordering positions—marked vertex smaller than the query vertex, or the smallest marked vertex greater than the query vertex.

The starting point  $v_{mark,q}$  for the search in the spatial structure is given as follows:

- › For the x-axis corresponding to pre-order traversal, if the mark found in the previous step lies to the *right* of the query position in the full tree, start just before that mark, otherwise start exactly at that mark.
- › For the y-axis corresponding to the post-order traversal, if the mark found in the previous step lies to the *left* of the query position in the full tree, start right after that mark, otherwise start exactly at that mark.

The structure for querying or updating field  $F$ , given that the correct sub-tree has been found in the hashed prefix index search, is given in pseudocode in Listing 5.9, and the algorithm for query and update is given in Listing 5.10. The translation procedure is illustrated in Fig. 5.14.

```
Label:
  def compare(that: Label): Cmp
  def prepend(): Label
  def append(): Label
  def tag: Int

Cmp    = -1 | 0 | +1
Rel    = (Label, Cmp)
Vertex = (lb_pre: Label, lb_post: Label)

Index[A]:
  Mark = (v_full: Vertex, v_mark: Vertex, value: A)
  Iso  = (rel_pre: Rel, rel_post: Rel)

  val list_pre : Tree
  val list_post: Tree
  val spatial: Spatial

Spatial:
  def nearest(point: (Int, Int), direction): A
  def insert(Mark)

Tree:
  def locate(test: Mark ⇒ Cmp): (Mark, Cmp)
  def insert(Mark)

  def query (v_full: Vertex): A
  def update(v_full: Vertex, value: A): Unit
```

*Listing 5.9:* Interface for querying and updating in a version tree

```
Index[A]:
def query(v_full: Vertex): A =
  translate(v_full) match
    case m: Mark => m.value
    case iso: Iso =>
      x = iso.rel_pre .label.tag + iso.rel_pre .cmp
      y = iso.rel_post.label.tag + iso.rel_post.cmp
      spatial.nearest((x, y), north-west)

def update(v_full: Vertex, value: A) =
  m = wrap(v_full, value)
  list_pre insert m
  list_post insert m
  spatial insert m

def wrap(v_full: Vertex, value: A): Mark =
  iso = translate(v_full)
  lb_pre = label(iso.rel_pre)
  lb_post = label(iso.rel_post)
  v_mark = (lb_pre, lb_post)
  (v_full, v_mark, value)

def label(rel: Rel): Label =
  if rel.cmp == -1 then rel.label.prepend() else rel.label.append()

def translate(v_full: Vertex): Mark | Iso =
  (m_pre, cmp_pre) = list_pre .locate(m => v_full.lb_pre compare m.v_full.lb_pre )
  if cmp_pre == 0 then m_pre
  else
    (m_post, cmp_post) = list_post.locate(m => v_full.lb_post compare m.v_full.lb_post)
    rel_pre = (m_pre .v_mark.lb_pre , cmp_pre )
    rel_post = (m_post.v_mark.lb_post, cmp_post)
    (rel_pre, rel_post)
```

*Listing 5.10:* Algorithms for querying and updating in a version tree

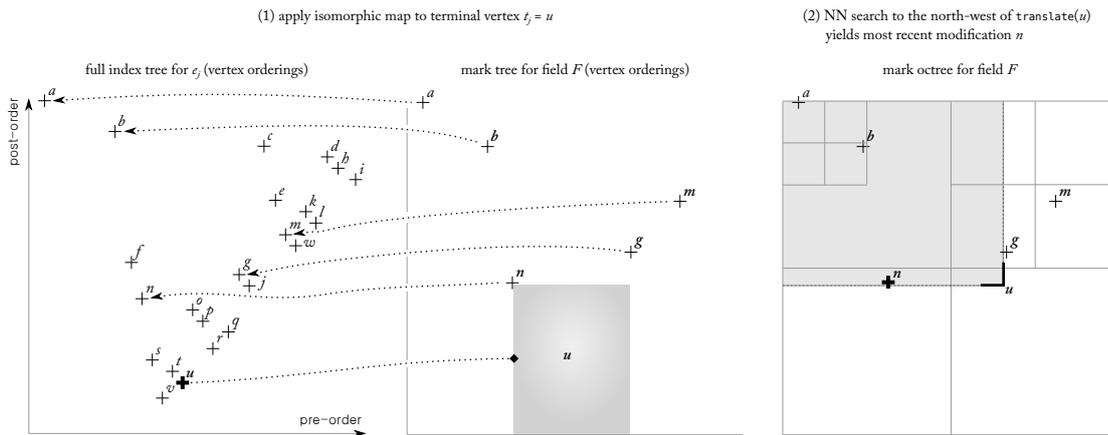


Figure 5.14: Querying a field  $F$  by locating the query vertex  $u$  in the mark tree using an isomorphic map, followed by the constrained nearest neighbour search

### Spatial Structure

We are now looking for a spatial index structure that allows dynamic point insertions, while being self-balancing and maintaining worst case performance guarantees for insertion and nearest neighbour (NN) search. There are simple structures like the  $k$ -d tree<sup>62</sup> which are suitable for offline (static) generation, but require a big effort to remain balanced under dynamic insertion. Another simple structure is the R-tree<sup>63</sup> which can be used in a dynamic fashion, but also does not guarantee worst case costs except under complex enhancements.

Another set of related trees is the quadtree (in two dimensions) or octree (for any number of dimensions).<sup>64</sup> In its main variant, it is based on the idea that recursively each internal node partitions the the space into four (for the quadtree) or  $2^d$  (for the  $d$ -dimensional octree) equally sized sub spaces. Without further modification, octrees have multiple problems, too. For example, their size depends on the locations of the points and not the number of points. This can be alleviated by compressing paths to include only those nodes which actually contain points,

<sup>62</sup>Jon Louis Bentley (1975), ‘Multidimensional Binary Search Trees Used for Associative Searching’, *Communications of the ACM* **18**(9), pp. 509–517.

<sup>63</sup>Antonin Guttman (1984), ‘R-trees: A dynamic index structure for spatial searching’, in: *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, Boston, pp. 47–57.

<sup>64</sup>Hanan Samet (1988), ‘An overview of quadtrees, octrees, and related hierarchical data structures’, *NATO ASI Series F40*, pp. 51–68.

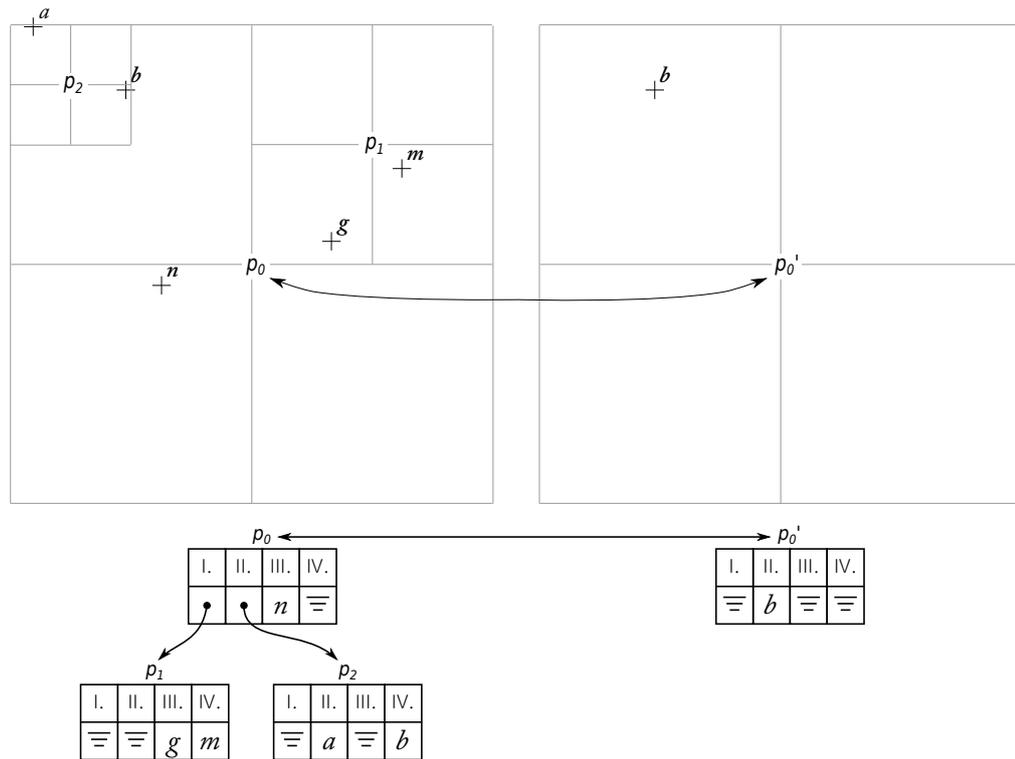


Figure 5.15: Skip octree consisting of a full and one subsampled tree

producing a structure with  $O(n)$  size, which however still has a worst case depth of  $O(n)$  as well.<sup>65</sup>

D. Eppstein, M. T. Goodrich and J. Z. Sun extended the idea of compressed octrees by maintaining a family of subsampled trees.<sup>66</sup> The result are two algorithms, one randomised, the other deterministic, that allow insertion and retrieval in  $O(\log n)$  time, which in the randomised version is expected and in the deterministic case is worst case cost. More importantly, they outline an algorithm for nearest neighbour search in  $O(\log n)$  time as well. We have implemented the deterministic version, which required some adjustments to the original algorithm proposed.

<sup>65</sup>Srinivas Aluru and Fatih E. Sevilgen (1999), ‘Dynamic Compressed Hyperoctrees with Application to the N-body Problem’, *Lecture Notes in Computer Science* **1738**, pp. 21–33.

<sup>66</sup>David Eppstein, Michael T. Goodrich and Jonathan Z. Sun (2005), ‘The Skip Quadtree: A Simple Dynamic Data Structure for Multidimensional Data’, in: *Proceedings of the twenty-first annual symposium on Computational geometry*, ACM, pp. 296–305; David Eppstein, Michael T. Goodrich and Jonathan Z. Sun (2008), ‘Skip Quadtrees: Dynamic Data Structures for Multidimensional Point Sets’, *International Journal of Computational Geometry & Applications* **18**(1 & 2), pp. 131–160.

The left side of Fig. 5.15 shows the non-sampled octree  $Q_0$  containing the four marked vertices  $b, g, m, n$  of Fig. 5.13, as well as the root vertex  $a$ . The root vertex is always explicitly marked in any newly created octree—just using the most recent value valid for that version—in order to ensure that the NN search succeeds even if there has not been any specific assignment for the sub-tree’s entering vertex.

The upper left part shows the spatial partitioning into the three nodes  $p_0, p_1$ , and  $p_2$ . The tree structure is depicted below. Each node contains  $2^d$  children, therefore in the 2-dimensional case there are four slots, sorted by the quadrant indices I, II, III, IV. A slot can be either empty, occupied by a leaf value, or link to another smaller node.

While the down sampling of the octree in the randomised case is controlled by a pseudo random number generator, the deterministic version—which we have implemented—uses an auxiliary data structure for its regulation, a so-called skip list, giving rise to the name skip octree for the spatial structure. A skip list is itself a successively down sampled structure, however one-dimensional. If one thinks of a sorted linked list, a skip list can be imagined as a group of linked lists which were obtained by successively and systematically removing elements from the original list. Elements in higher level (sparser) lists have additional pointers to their corresponding cells in the next lower level. A search for an element begins at the head of the list in the highest level. This list is traversed until either the sought element is detected or an element is found which is greater than the sought element. In this last case, one moves down to the next level and repeats the procedure, until one either finds the element or ends at the lowest level of the hierarchy.

**Skip Lists** In the original proposal by W. Pugh, these lists were probabilistic themselves and therefore only provided expected performance bounds.<sup>67</sup> Another author, T. Papadakis, has developed two deterministic versions of skip lists, one based on the linked list approach of the original structure, the other grouping multiple cells together in horizontal arrays.<sup>68</sup> The title of Pugh’s paper suggests a similarity between skip lists and balanced trees, and Papadakis in fact

<sup>67</sup>William Pugh (1990), ‘Skip Lists: A Probabilistic Alternative to Balanced Trees’, *Communications of the ACM* **33**(6), pp. 668–676.

<sup>68</sup>Thomas Papadakis (1993), ‘Skip Lists and Probabilistic Analysis of Algorithms’, PhD thesis, Waterloo, Ontario: University of Waterloo, ch. 4.

shows that deterministic skip lists (DSL) using the horizontal array technique are very similar to B-trees and  $B^+$ -trees. We have implemented the linked list version in class `LLSkipList` and the horizontal array version in class `HASkipList`. The latter performs better when its nodes are persisted to disk which is the case in our framework.

Without going into detail, the approach of making a DSL self balancing is to ensure that the gap between adjacent cells—which results from down sampling the next lower level list—is never smaller or greater than given bounds. In the most basic linked list version, the gap size varies between 1 and 3 (this is also known as 1-2-3 or just 1-3 skip list). Therefore the size of the list at each level is at most half of the size it has on the next lower level, and it can easily be seen that the “tree” depth is bound by  $O(\log n)$ . Algorithms can be devised which allow a simple top-down insertion which proactively closes gaps on the way down the hierarchy in anticipation of the new element being inserted on the leaf level. Removal is also quite simple with proactively opening additional gaps, the only exception being the removal of elements which occur on higher levels of the hierarchy, in which case a second pass is required.

A 1-3 skip list in horizontal array form which contains the leaves of the previous octree is shown in Fig. 5.16. Each array contains up to four elements. Arrays on the leaf level just contain the stored keys or values themselves, while higher level arrays also contain pointers to lower level nodes. A lower node always ends with the key from which one descended in the next higher level. A special key  $\infty$  is used to terminate the sorted sequence. In this example, the order of the elements is  $a, b, n, g, m$ , and elements have been inserted in their lexicographical sequence. Therefore, the list initially consisted only of one leaf array containing first  $\langle a, \infty \rangle$ , then  $\langle a, b, \infty \rangle$ , then  $\langle a, b, g, \infty \rangle$ , and then became overfull, resulting in a split and the promotion of  $b$  to the next level. In the second level list, there is now an initial “gap” ( $a$  is missing), and a “gap” between  $b$  and  $\infty$  (elements  $n, g$ , and  $m$  appear between these two in the next lower level).

**Down Sampling** Eppstein, Goodrich and Sun’s idea of using skip lists for their octrees is very clever: The lists are not in any way used as search structures in their own right. But instead *the propagation of elements to higher levels is observed*. When  $b$  was promoted to the next level,

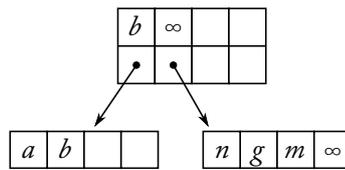


Figure 5.16: A skip list storing the marked vertices is used to control octree decimation. The lowest level corresponds to  $Q_0$ , the left side of Fig. 5.15, the next higher level reflects the sub sampled version  $Q_1$ , the right side of Fig. 5.15.

the first down sampled octree  $Q_1$  was created, containing only that element. It is shown on the right-hand side of Fig. 5.15.

In order to build the skip list of leaves, a total ordering must be imposed on them. Eppstein, Goodrich and Sun use a binarisation of the tree through the introduction of dummy nodes between each two orthants, resulting in a depth-first kind of traversal. However, they also note that a purely lexical ordering of the coordinates could be used. In fact, none of the methods for searching the octree relies on the ordering of the skip list, and neither does their proof of worst case performance. Since in our case no two vertices share the same horizontal or vertical coordinate, a total order is already established through the pre-order positions of the vertices—it therefore is reflected in Fig. 5.16—and the internal nodes of the octree do not participate at all in the ordering, simplifying this step.

**Searching** Since addition and removal follow the algorithms of Eppstein, Goodrich and Sun, we shall only outline the procedure to find the nearest marked ancestor, given the skip octree thus set up. In other words, the algorithm for method nearest in type `Spatial` of Listing 5.10 must be defined.

Eppstein, Goodrich and Sun view the NN search problem as an extension of a range query—which is simpler—however with the radius being not known in advance. During the search, a priority queue is maintained which contains possible squares yet to be explored, where the priority is the minimum distance of a square (i.e. its closest corner or side) from the query point. The algorithm proceeds in iterations, each of which begins by taking off the priority queue the square which is closest. Of all the points (marked vertices) encountered during the search, the closest one is remembered. Given a square  $p$  taken from the queue, one first goes to its

version  $p'$  in the highest level of the sub sampling scheme in which it still exists. As indicated in Fig. 5.15, there are bi-directional cross links between a square  $p_i$  and its version in the next higher level,  $p'_i$ . If  $p'$  contains at least two children which have the same distance from the query point, add them to the queue and start the next iteration. Otherwise, a zig-zag movement is made by either following down a child of  $p'$  which has the same distance to the query point as  $p'$  itself, or by going back to the next lower sub sampling level. When an iteration is over and the queue is empty, the search is completed and the remembered nearest point is returned.

This traversal of the subsampled octree family to the right and then returning in zig zag to  $Q_0$  is crucial to bounding the performance of the algorithm. The authors state in lemma 9 that one such traversal which delimits what they call an “equidistant path” takes  $O(\log n)$  time. Two possible paths are shown in Fig. 5.17. The paths are defined by successively shrinking squares, while the minimum distance between the query point and the squares remains the same. Depending on the distance metric (e.g. Euclidean), only a constant number of squares hanging off this path must be re-inserted into the priority queue, according to lemma 12, bounding the cost of the overall query to  $O(\log n)$ .

The algorithm has two problems. First, the “bent” path of Fig. 5.17b means that some more squares hanging off  $p_1$  and not included in the equidistant path must be included. The paper proposes that for this reason,  $2^d$  additional directional ancestor pointers be maintained for each square. It is claimed that these can be updated in constant time, however we found this not to be true. In a private correspondence with the authors, it was suggested that these pointers are not needed, but that a direct search for the directional ancestors (e.g., finding the lowest ancestor to the east of  $q$ ) in constant time is possible.

The bigger problem seems to be the proof of the algorithm’s running time. We conducted an empirical verification of the number of steps performed, and found at least one pathological case, shown in Fig. 5.18, which violates the bound. In this case, the search takes  $O(\sqrt{n})$  steps. It is unclear what the error in the proof is, but we suspect that it originates in the translation of the range query algorithm, where the search radius  $r$  is constant, to the nearest neighbour search, which is treated as a range query with an unknown and yet to determine radius. The proof for

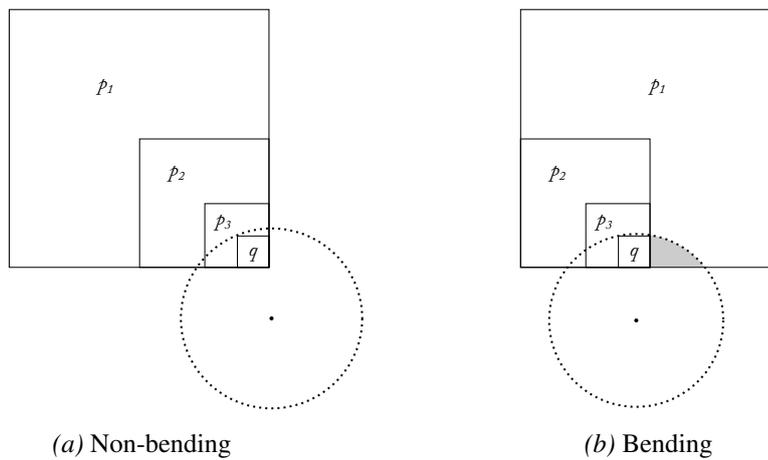


Figure 5.17: Types of equidistant paths

the range query relies on lemma 7 which binds the number of “critical squares” to a constant based on  $r$ . Since in each round of the nearest neighbour search the current radius  $r$  may change (shrink), we think that lemma 7 can no longer be used to bind the performance of the NN search.

While this is an unforeseen and dissatisfying discovery, in all practical scenarios the distribution of points is more random and the pathological case does not appear. In our empirical tests, the performance was quite good and indeed showed logarithmic behaviour. We were able to avoid the directional ancestor search of the first problem altogether by guiding the narrowing of the search radius by a maximum distance metric instead of the minimum distance metric.

Finally, in order to restrict the search to a given quadrant, one can define the minimum distance of squares which are outside the quadrant and the maximum distance of squares which extend beyond the quadrant as  $\infty$ . The calculations can be simplified by using either a Chebyshev metric or a squared Euclidean distance instead of the regular Euclidean distance.

## 5.7 Extensions and Alternatives to Persistence

The persistence approach deals with data structures as the principle abstraction, assuming only one implicit observer which is the same instance as the operator on the structure. Time is reflected by an irreversible and ongoing sedimentation of layers. Actions are only implicit in these resulting layers.

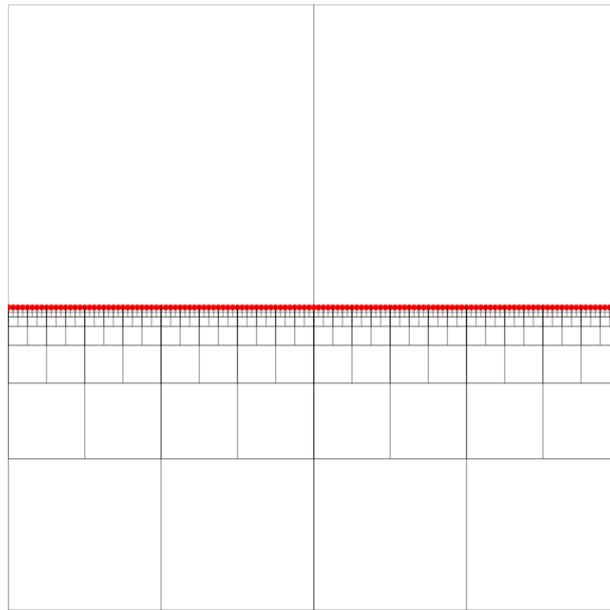


Figure 5.18: Pathological configuration violating the NN performance bounds. Points are evenly arranged on a horizontal line.

To build large systems, on the other hand, requires a modularisation process in which multiple instances of operators and observers may exist, and it is desirable that actions can be performed which influence a variety of concurrent objects, perhaps “retro-actively” in the sense that if two objects  $X$  and  $Y$  rely on some property  $Z$ , we may adjust  $Z$  virtually in the past, so that its change is reflected by all future representations of  $X$  and  $Y$ . For example, imagine that these are objects depending on a random seed  $Z$ , and one wishes to regenerate a structure by retroactively changing the value of  $Z$ .

These scenarios are somewhat *orthogonal* to the perspective that the persistence approach provides. We will briefly review one particular approach that was presented as an *alternative* to persistence, ‘retroactivity’, and how a constrained subset of retroactive behaviour can be realised as a modification of persistence. We will then introduce another perspective shift and see how changes can be observed through event processing which addresses the situation where observer and mutating agent are separated.

### 5.7.1 Critique of the Persistence Approach

Retroactivity as a new paradigm is proposed by E. D. Demaine, J. Iacono and S. Langerman and contrasted with persistence:

«The key difference between persistent and retroactive data structures is that, in persistent data structures, each version is treated as an unchangeable archive. Each new version is dependent on the state of existing versions of the structure. However, because existing versions are never changed, the dependence relationship between two versions never changes ... Thus, the persistence paradigm is ... inappropriate for when changes must be made directly to the past state of the structure.

In contrast, the retroactive model ... allows changes to be made directly to previous versions. Because of the interdependence of versions, such a change can radically affect the contents of all later versions. In effect we sever the relationship between time as perceived by a data structure, and time as perceived by the user of a data structure.»<sup>69</sup>

Technically, this is not completely correct. While typically persistence indeed grows the structure by forking off branches from previous states, a direct overwriting of versions would be possible. Taking the linked list example of Fig. 5.7, one could in a fifth step modify version  $v_3$  by removing the second cell  $w_0$ , and the sequence seen from  $v_3$  would become  $(1,4,6)$ , and seen from  $v_4$  it would now be  $(1,4,6,1,3)$ , so retroactive changes *do* affect future versions.

But Demaine, Iacono and Langerman are right—had we removed the last cell retroactively in  $v_3$ , the sequence would become  $(1,2,4)$  in  $v_3$ , but seen from  $v_4$  it would become  $(1,2,4)$  as well—and not  $(1,2,4,1,3)$ —because the catenation performed in  $v_4$  would have effectively become invisible, as it operates on an element which was already removed from the data structure. Clearly, while the data structure remains “syntactically” consistent, this is not an expected result. In the retroactive framework, the “semantics” would be correctly modelled, because instead of talking about data structures, it talks about «operational history».

<sup>69</sup>Erik D. Demaine, John Iacono and Stefan Langerman (2007), ‘Retroactive Data Structures’, *ACM Transactions on Algorithms (TALG)* 3(2), 13:1–13:20.

Very much like undo/redo trees, retroactive data structures use representations based on the vocabulary of allowed operations, such as keeping a history of insertions and removals of elements from a set. At first glance they seem superior to persistence, but in fact they are orthogonal. A retroactive operation is *ephemeral* in the sense that the system forgets how the structure looked at a certain point in (transactional) time. The claim that data structure time and “user time” are severed is therefore misleading. We have a random access system with respect to  $\mathcal{T}_P$ , but  $\mathcal{T}_K$  is not traced.

The strongest limitation of retroactive data structures is that, unlike persistent structures, no general transform is available that can be applied to any ephemeral structure so that it becomes retroactive. The general theorems given in the paper only hold for very constrained structures, such as those solely composed of commutative and invertible operations—e.g. insertion and deletion of unrelated elements into and from an unordered set—in other words operations which do not have a relation with each other in time. The linked list example, with an operation such as insert  $X$  after  $Y$ , already requires a specially constructed structure and cannot be automatically generated.

Also, the notion of branching time is completely absent, so the temporal evolution degenerates again from a tree or graph to a time-line. The example implementations of Demaine, Iacono and Langerman rely on this single time-line which is represented by segment trees. Other authors have developed more sophisticated data structures based on retroactivity, such as quadtree based range and nearest neighbour searching,<sup>70</sup> but maintain the idea of the one timeline which does not permit concurrent versions or the re-entry of elements. The clear sense of distinguishing valid from transactional time seems to be missing. M. T. Goodrich and J. A. Simons use a video cutting program as example application, and write:

---

<sup>70</sup>Michael T. Goodrich and Joseph A. Simons (2011), ‘Fully Retroactive Approximate Range and Nearest Neighbor Searching’, *Lecture Notes in Computer Science* **7074**, pp. 292–301.

«Queries and updates happen in *real time*, but are indexed in terms of the timeline. For instance, one can ask to mark an object to exist for the first time at time index  $t_0$ , that is, to be inserted at time  $t_0$ . Likewise, one may ask to mark an object so it is identified as removed as of time index  $t_1$ , that is, to be deleted at time  $t_1$ .»<sup>71</sup>

What is presented is a timeline canvas, where the extent and purpose of tracing its evolution remains in the dark.

While from the implementation point of view, persistence is far superior due to the availability of a general transform, both approaches should be carefully examined in terms of their metaphorical and conceptual potential. Persistence is appealing because of its neglect of agents (“who writes” and “signification”), which better resonates with the philosophical outline of Chap. 3, on the other hand retroactivity puts the operations in the foreground—manifest as a set of updates  $U = [u_{t_1}, \dots, u_{t_m}]$ —so the analysis of the past traces becomes easier; it is explicit which steps were performed in a transaction, whereas in the persistence model only heuristics can be used to find out which parts of a data structure were actually affected.

Another interesting extension of retroactivity—despite having the same problems such as lack of general transforms—is *non-oblivious* retroactivity, which gives queries the same status as updates and includes them in the tracing structure, something that resonates well with our conclusion that writing and observing are essentially the same.<sup>72</sup>

Overall, the shift from state to behaviour in retroactivity would indicate a better suitability for representing interactions. In the next sections we will present a limited form of quasi-retroactivity—a retroactivity which may influence future versions by amending instead of overwriting the past.

### 5.7.2 Quasi-Retroactive Transactions

The problem of allowing the propagation of a change to an element “in the past” to all future version is shown in Fig. 5.19. An element  $r_1$ , for example a sounding structure that lasts for a

<sup>71</sup>Goodrich and Simons, ‘Fully Retroactive Approximate Range and Nearest Neighbor Searching’.

<sup>72</sup>Umut A. Acar, Guy Blelloch and Kanat Tangwongsan (2007), *Non-oblivious Retroactive Data Structures*, tech. rep. CMU-CS-07-169, Carnegie Mellon University, School of Computer Science.

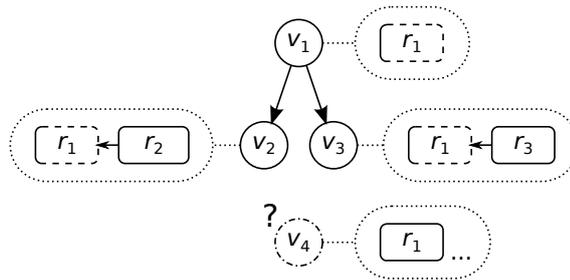


Figure 5.19: Problem of operating retroactively on branched elements

given amount of time, is created in the first version, perhaps with initially undefined duration (hence the dashed outline). Subsequently, the structure is elaborated in two distinct branches  $v_2$  and  $v_3$ , where other elements refer to the duration of  $r_1$ . This is possible by using expressions as explained in Sect. 5.9.

To decide on or update the duration of  $r_1$  in version  $v_4$  calls for something akin to a retroactive operation: a correction or amendment of  $v_1$ . In order to not undermine the preservation of the structure's history, we do not wish to just overwrite version  $v_1$ . We could add two vertices  $v_{4a}$  and  $v_{4b}$  as children to  $v_2$  and  $v_3$ , respectively, but the forward linking approach of persistence does not automatically provide us with a list of all the leaves in the version graph which refer to  $r_1$ , so we cannot simply iterate over them and adjust the value. Even if that was possible, it would mean that we break the one-to-one relation between transaction and access path, as well as having a cost which is linear in the number of versions which refer to a value.

Since the version sub-tree is implicitly formed by a pre-order and post-order traversal list, it is possible to define the retroactive insertion of a vertex between a parent and all its children. It simply requires the addition of a pre-order entry after the parent's pre-order entry and a post-order entry right before the parent's post-order entry. The resulting list which uses the unification of pre- and post-order as discussed in Sect. 5.6.4 would be  $\langle v_{1_{pre}}, v_{4_{pre}}, v_{3_{pre}}, v_{3_{post}}, v_{2_{pre}}, v_{2_{post}}, v_{4_{post}}, v_{1_{post}} \rangle$ . The effect is illustrated in Fig. 5.20.

To preserve the history, we add the incremental version identifier to the octree which thus becomes three-dimensional. The nearest neighbour search is constrained to only the orthants which contain versions with identifiers less than or equal to the query version, excluding marked

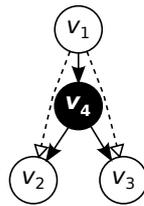


Figure 5.20: A version  $v_4$  is inserted after a parent  $v_1$ , retroactively becoming the new parent of the former parent’s children  $v_2$  and  $v_3$ .

points which lie “in the future”.<sup>73</sup> This implies that the change made in  $v_4$  is not seen in either  $v_1$  or  $v_2$ , however as soon as one descends to a child of these versions, the change becomes effective.

Another idea we had explored<sup>74</sup> is the use of individual components of the path representation as variables, so that one can switch between variants of a structure, while still being able to work on elements which are shared across these variants. This worked by defining a “neutral” version vertex as a fallback when a value is not found in the currently viewed variant. However, it implies that the particular path representation—from the compressed path method—is an explicit part of the user interface of the system and must be specially handled. Instead, we now prefer a generic container based document, where variants can easily be grouped together, possibly by melding different versions in which these variants were originally developed. Variables which should affect all variants are then represented by expression variables which are updated with the quasi-retroactive insertion of Fig. 5.20.

## 5.8 Event Processing

Everything is fine as long as we evaluate (“pull”) the data structure from one master access point. But when there are other observers, such as a user interface which displays values, a transport which scans a structure in real time or a sound synthesis engine, these instances must be notified about changes to the underlying data structure, and this notification must be modular for an application to be scalable. If the user modifies a sound object, in any reasonably sized project the code that issues the modification cannot keep track of the objects which are affected by that

<sup>73</sup>Cf. Hannes Holger Rutz, Eduardo Miranda and Gerhard Eckel (2010), ‘On the Traceability of the Compositional Process’, in: *Proceedings of the 7th Sound and Music Computing Conference (SMC)*, Barcelona, 38:1–38:7

<sup>74</sup>Ibid.

modification. A mechanism which inverts the control flow is needed, so that modifications are “pushed” actively to the dependent objects.

### 5.8.1 Model-View-Controller and Reactivity

A classical concept of interaction is the Model-View-Controller (MVC) paradigm, which was established in the context of object-oriented programming and the design of user interfaces for the *Smalltalk* language.<sup>75</sup>

These three components are seen as separate concerns or modules. A model encapsulates a mutable structure under observation, a view displays certain aspects of a model, and a controller issues modifications of a model. The separation has several advantages. A model can be shown in any number of concurrent views, each perhaps focusing on a different aspect of the model. If the model was a sound object, one view could show its evolution in  $\mathcal{T}_K$ , another view could use one version snapshot and instead use the screen space to show the development in  $\mathcal{T}_P$ , etc. The dependency usually is strictly from view and controller to the model, therefore models can be independently developed and are agnostic of their views. A sound installation could run either headless—without user interface—or with a GUI attached to it. If the user interface is modified, the model is not affected.

The pattern for establishing the dependencies between these components is called observer or publish-subscribe pattern: The model maintains a list of observing or subscribed dependents which are opaque except for a defined callback mechanism, for example they could be functions which take an update message as argument. When an aspect of the model is changed, it publishes this change to the observers. A mutable cell holding an integer value could, when updated, publish the new integer value, or it could publish the old and the new value as a pair.

Although MVC is still widely used—from Java’s *Swing* user interface to web frameworks like *Ruby on Rails*—there are several shortcomings of the underlying observer pattern. I. Maier and M. Odersky provide a concise list of these.<sup>76</sup>

---

<sup>75</sup>Glenn E. Krasner and Steven T. Pope (1988), ‘A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80’, *Journal of Object Oriented Programming* **1**(3), pp. 26–49.

<sup>76</sup>Ingo Maier and Martin Odersky (2012), ‘Deprecating the Observer Pattern with Scala.React’, *Technical Report EPFL-REPORT-176887*. *Ecole Polytechnique Fédérale de Lausanne*.

- › Observers are invoked as callbacks, so their reactions must be implemented using side-effects instead of, e.g. functional composition.
- › As a consequence, observers cannot be composed, although reactions often require the collaboration between multiple observers, e.g. keyboard and mouse handlers participating in a drag-and-drop operation.
- › A great amount of bookkeeping is required, including explicit release when an observing instance disappears.
- › Reactions are often spread across the code base and not clearly separated from the application logic.
- › Cross observer communication is potentially indeterministic and can cause glitches.
- › It is often unclear what an observer’s function is, as opposed to a declarative approach which better captures the meaning of a reaction.

Two approaches which address these issues to different degree might be called event-based and dataflow-based or reactive, respectively.<sup>77</sup>

### Event-based Approaches

Event-based systems address the composition and deterministic behaviour and improve on the declarative aspect. An example of this is *EScala*,<sup>78</sup> which comes in two flavours: A purely library-based solution for *Scala*, and a language extension to *Scala* which adds additional keywords. An example of the latter is shown in Listing 5.11.<sup>79</sup>

Events take a type parameter which represents the update associated with the event. Pure trigger-like events do not carry any additional information and therefore use the `Unit` type. In the example, the `invalidated` event passes the new rectangle of the figure as its update value. Event dispatch can originate either from an “imperative” event, to which the update value is explicitly

<sup>77</sup>Cf. Guido Salvaneschi, Gerold Hintz and Mira Mezini (2012), *REScala: Bridging Between Object-oriented and Functional Style in Reactive Applications*, tech. rep., Darmstadt: Technische Universität Darmstadt

<sup>78</sup>Vaidas Gasiunas et al. (2011), ‘EScala: modular event-driven object interactions in Scala’, in: *Proceedings of the tenth international conference on Aspect-oriented software development*, ACM, pp. 227–240.

<sup>79</sup>This example is taken from *ibid.*, and slightly adapted to include imperative events.

```
abstract class Figure {
  imperative event moved[Unit]
  event resized[Unit]
  event changed[Unit] = resized || moved || after(setColor)
  event invalidated[Rectangle] = changed.map(_ => getBounds())
  ...
  def moveBy(dx: Int, dy: Int): Unit = {
    position.move(dx, dy)
    moved() // imperatively dispatch event
  }
  def setColor(value: Color): Unit = color = value
  def getBounds(): Rectangle
}
```

Listing 5.11: Event definition and composition in *EScala*

applied (method `moveBy` shows this type), or through a before or after hook, an idea the authors adapted from aspect orient programming (AOP). Furthermore, events can be composed with operators such as `||`, where `changed` is dispatched if either of `resized`, `moved`, or `after(setColor)` is observed, and they can be chained with the `map` function.

To observe an event, a function, which takes the update value as argument, is registered as follows:

```
figure.invalidated += { r => println("New_rectangle_is_" + r) }
```

An important aspect is the propagation of updates when an event is dispatched. *EScala* uses a time tag (“id”) which is incremented each time an imperative event is dispatched. The dependency graph is traversed, and at each node the intermediate events are associated with that tag. If an event was already seen within the same cycle, it will not be followed, in order to suppress feedback. Furthermore, dependencies between events are only latent—e.g. in Listing 5.11, the dependency of `changed` on `resized`—they become manifest (“deployed” in their terminology) when an actual observer is attached as leaf to the dependency graph. That way, blind paths are not followed unnecessarily. The leaf reactions are first completely collected, and then executed in a second pass, to avoid glitches such as new reactions being added during the traversal.<sup>80</sup>

---

<sup>80</sup>The implementation of *EScala* is presented in more detail in Vaidas Gasiunas et al. (2010), *Declarative Events for Object-Oriented Programming*, tech. rep. TUD-CS-2010-0122, Technische Universität Darmstadt

## Reactive Dataflow Approaches

An example of reactive systems is *Scala.React*.<sup>81</sup> The concept of events is complemented by a type `Signal[A]` which is basically a dataflow cell embodying an expression that evaluates to some value of type `A`. The correspondence to an imperative event would be a variable cell which is a signal along with an update method to set the value explicitly. A signal thus has an interface very similar to the STM `Source`, and a variable has an interface similar to the combination of `Source` and `Sink` into a reference cell, as introduced in Listing 5.4.

A signal-based version of the *Akka* dataflow example from Listing B.5 is shown in Listing 5.12. Apart from the few syntactical variations, the difference is that an *Akka* dataflow variable can be initially undefined and can be written only once, whereas signals must have an initial value, so `v2` is set to zero first, and they can change any number of times. Composed signals are created through the `Lazy { }` construct. Evaluations or updates must be performed within a specific context, created via a `schedule` function. The actual event propagation is only performed when the `runTurn()` function is called, allowing for the successive scheduling of different closures to be executed on the same logical turn.

Publish-subscribe, events and signals are not mutual exclusive mechanisms, but rather different abstraction layers. Event systems are implemented using publish-subscribe patterns, and signal changes are propagated through an event system. In Listing 5.12, change events are observed through the `observe` function. Similarly, in *REScala*—which combines *EScala* and *Scala.React*—signals are turned into events by means of a method `aSignal.changed`, and events can become signals through `anEvent.hold`, a sample-and-hold operation.<sup>82</sup> This duality of signals as functions of time and events as instants in time follows closely the concept of *behaviours* and *events* in functional reactive programming (FRP) as described by C. Elliott and P. Hudak.<sup>83</sup>

Event streams and reactive cells each have their advantages and disadvantages. While the latter may appear generally favourable, one should consider the observation of a simple data structure

<sup>81</sup>Maier and Odersky, ‘Deprecating the Observer Pattern with *Scala.React*’.

<sup>82</sup>Salvaneschi, Hintz and Mezini, *REScala: Bridging Between Object-oriented and Functional Style in Reactive Applications*.

<sup>83</sup>Conal Elliott and Paul Hudak (1997), ‘Functional reactive animation’, *ACM SIGPLAN Notices* **32**(8), pp. 263–273.

```
val v2 = Var(0)
val v1 = Lazy { v2() + 10 }
val f = Lazy { v1() + v2() }

schedule {
  new Observing {
    observe(f) { p => println("Observed:_" + p) }
  }
  v2() = 5
}
runTurn()
```

Listing 5.12: Reactive signals with *Scala.React*

such as a mutable list: In designing what G. Salvaneschi, G. Hintz and M. Mezini call a «signal-enabled data structure»,<sup>84</sup> there are simple elements such as the list’s size, which can be easily represented as signals; but the other examples they give, inspecting the head element of the list or having a filtered version of the list, are fairly difficult. What happens when the head element disappears as the list becomes empty? Implementing a filtered list by rebuilding it each time the underlying full list changes, can be costly (more so, when the list is persistent). And how can operations—e.g. insertion—be performed on a filtered list so that they are consistently reflected by the full list? In *REScala*, the solution for a filtered list is to simply make it read-only, so it never needs to mutate the full list it depends on.

An additional case is not discussed at all in these papers, although it will be relevant for us: When a collection of mutable objects is observed, the observer typically wants to react to changes in the collected objects. If we want to avoid the scenario that the observer has to iterate over all elements and keep track of additions and removals, this requires that the collection itself dispatches events which bundle all events arriving from any of the contained objects.

---

<sup>84</sup>Salvaneschi, Hintz and Mezini, *REScala: Bridging Between Object-oriented and Functional Style in Reactive Applications*.

### 5.8.2 Implementation of the Event Layer

Our system is based on events in the first place, on top of which, in a second step, expressions can be built which are similar to signals. We introduce the event layer in this section. The design was made with the following aspects in mind:<sup>85</sup>

- › Events are transactionally embedded. Events originate from manipulations of mutable objects within an ongoing transaction. Reactions must be able to access objects within the same transaction, and it must be possible to encapsulate any number of reaction chains within a single transaction, so that they are logically coalesced into the same version.
- › If realtime sound synthesis is seen as “view” in the MVC pattern, the sound synthesis must consistently react to sound object mutations. If for example a group of objects is evenly spatialised across a number of speakers, the action of removing an object from the group must be followed by a reaction in the sound synthesis view that adjusts the DSP chain. This might trigger other events within the DSP apparatus, all of which must happen in the same transaction. If anything goes wrong, the whole transaction must rollback as if the event was never emitted.
- › If several views exist for different branches in  $\mathcal{T}_K$ , events must be isolated to propagate only to observers in the same temporal branch, leaving other branches unaffected.
- › If multiple observers react to a change in an object, their views of the object must be consistent. This is in conflict to the first two points—we want all reactions to happen within the same transaction, and observers are free to mutate objects, thus strict isolation is not possible.

Events are integrated with the persistent layer mainly through additional classes, requiring only a small new hook in the `Sys` and `Txn` types. A mutable object which emits events, maintains a `Node` which is able to route messages to events, using a slot identifier which is unique for each event. The node also contains a `Targets` object which holds the immediate dependents of the

<sup>85</sup>Cf. Rutz, ‘A Reactive, Confluently Persistent Framework for the Design of Computer Music Systems’

node. A mechanism is in place, similar to what was done in *EScala*, that lazily builds up the dependency network.

The dependents list in `Targets` uses a special variable type which differs from a regular `S#Var` by having limited durability. Values written to an event variable are integrated into the same system: for a confluent system, the dependents are stored in a confluent variable, for an ephemeral system, they are stored in an ephemeral variable. However, they are held in a separate database file which is reset when an application is started. As a result, we can safely quit an application anytime and will not have “dangling” or “orphaned” observers. Views are considered ephemeral and have to be recreated when the application launches.

Since our persistent layer uses a top-down serialisation approach—an instance *A* holding an object *B* knows how to deserialise *B*, so without the presence of *A* we are not able to parse the serialised form of *B*—a solution must be found that enables a bottom-up (inverted control) mechanism such as event propagation to operate. The essence of MVC was exactly that from a model’s perspective its observers are opaque, so it cannot possibly know how to deserialise them. Moreover, the system cannot be closed over the type of possible observers but must be able to accommodate any kind of observer, otherwise it would not be modular and scalable.

The solution we propose is to see event propagation as a “tunnel” from a source (model) to an observer (view). All the intermediate instances where events are composed and chained together lie in the darkness of the tunnel. The model is always known, because a controller mutates the model. The leaf observers all correspond to “live” views and may be kept in memory, so they need not be serialised. Only the things inside the tunnel must be serialised and deserialised “blindly”.

### **Network Connection**

As an example, and partly forestalling expressions, we will use a random number generator that can be triggered, along with a mapping of its outcome—adding a number to it—and an observer reacting to the mapped value. The interfaces are shown in Listing 5.13. Similar to the signals in *Scala.React*, an expression can always be evaluated using the `value` method, and events are

```

trait Expr extends Writable {
  def value(implicit tx: S#Tx): Int
  def changed: Event[S, Int, Expr]
  def +(that: Int)(implicit tx: S#Tx): Expr
}

object Random {
  def apply(min: Int, max: Int)(implicit tx: S#Tx): Random = ???
}

trait Random extends Expr {
  def update()(implicit tx: S#Tx): Unit
}

def example()(implicit tx: S#Tx): Unit = {
  val r = Random(0, 10)
  val f = r + 64
  f.changed.react { implicit tx =>
    n => println("New_value:_" + n)
  }
  r.update() // roll dice.
  r.update() // once more.
}

```

*Listing 5.13: Events in an observed expression*

obtained through an associated `changed` method. The `Random` expression has an additional `update` method to trigger the calculation of a new number.

The example shows how an observer is attached to a compound expression  $f$  made from a random expression  $r$  to which a binary plus operator is applied. The `react` method attaches the observer which is a function from a transaction to a function from the event type to `Unit`. Executing the example will successively print two random numbers between 64 and 73.

The tunnels are established by the first `react` call to an event, and will be torn down with the removal of the last observer. Internally these methods map to `--->` and `-/->` which trigger `connect` and `disconnect` messages on the source node for the first added and last removed selector. Listing 5.14 shows how the implementation of the adding expression may look. When it is connected, it enables the link to its source expression, and when disconnected, it cuts that link again. The `value` method obviously just evaluates the source and applies the binary operation. Similarly, the `pullUpdate` method, which resolves an event in the pull phase, pulls

```
class AddImpl(val targets: Targets[S], ex: Expr, a: Int)
  extends Expr with impl.StandaloneLike[S, Int, Expr] {
    ...

    def connect    ()(implicit tx: S#Tx): Unit = ex.changed ---> this
    def disconnect()(implicit tx: S#Tx): Unit = ex.changed -/-> this

    def pullUpdate(pull: Pull[S])(implicit tx: S#Tx): Option[Int] =
      pull(ex.changed).map(_ + a)

    def value(implicit tx: S#Tx) = ex.value + a

    def changed = this
  }
```

*Listing 5.14: Engaging event chains and mapping updates*

from the source and then applies the binary operation. `pullUpdate` returns an optional value, so that events may be filtered or swallowed, for example if a resulting value does not constitute a nominal change. Finally, the trait `StandaloneLike` can be used when a node has exactly one event, as it conveniently combines and implements the two, hence allowing the aliasing `def changed = this`.

## Dispatch

Dependents are represented by small proxies called *selectors*. The observers at the leaves of the network are embodied by `ObserverKey` selectors. These carry a unique identifier which is used as a key to look up the actual observer in an in-memory dictionary. Events, through which messages pass in the tunnel, are subtypes of `VirtualNodeSelector`, the form in which they may be serialised. This type of selector stores a `VirtualNode` which may be either an opaque serialised node, `Raw`, or the fully recovered node. In short, selectors and raw virtual nodes can be deserialised on the way from model to view without any additional knowledge being required about the type or shape of the participating nodes.

This first “push” phase of event dispatch is shown in Fig. 5.21a. At the end of this phase, all observers have been collected. In order to devirtualise the raw nodes and to process the actual update message, a “pull” phase is issued in the opposite direction, as shown in Fig. 5.21b.

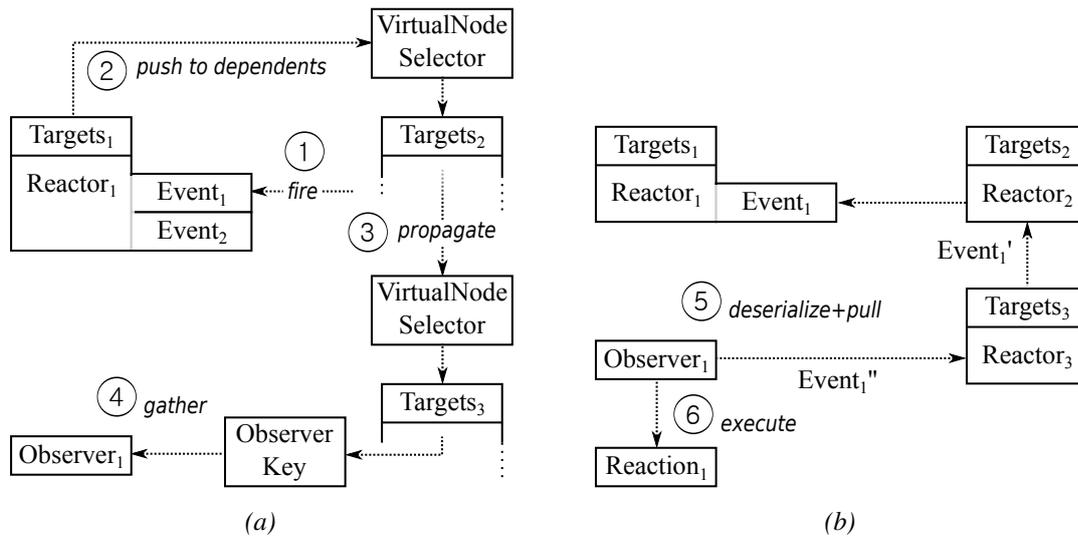


Figure 5.21: Push and pull phase in event dispatch

### Accessing Models and Mapping Views

Reacting to an immediate model or part of a model is straightforward: The event is dispatched inside a transaction, and if the event update type includes the model, an observer may immediately access that model. For example, in the sound processes framework (Sect. 5.11) the `Proc` class defines an event `changed` with the following update type:

```
case class Update[S <: event.Sys[S]](proc: Proc[S], changes: IndexedSeq[Change[S]])
```

So any observer reaction has a fresh accessor to the emitting process via the `proc` field. Two problems remain: First, a controller may have to access a model due to user interaction such as clicking a button, not having direct access via an event update. And second, a view may observe a collection of models and associate each with an auxiliary data structure. When the model is updated, it must find this auxiliary data structure.

The first problem is solved by creating a *handle* to the model, which acts as an STM source (Listing 5.4). For example, a handle for a sound process `proc` is created via

```
val hndl = tx.newHandle(proc)
```

This serialises the process in memory. Within a different transaction—which must be a descendant of the version in which the handle was created—such as the one that transports an event

update, the process can be freshly accessed (deserialised) through `hdl()`. One can think of handles as decentralised access pointers in the Fiat and Kaplan model.

The second problem also concerns the “freshness” of the model. Here, an auxiliary structure  $V$  is created in version  $v_i$  and associated with a model  $M$  which was accessed in that version  $v_i$ ; an event arrives in version  $v_k$ , a descendent of  $v_i$ , and the task is to find  $V$ , given  $M$  accessed in  $v_k$ . Obviously, a regular dictionary will not suffice, as the key identity changes due to the version progression. The key which was used for storing  $V$  is indeed a prefix of the access path of  $M$ , so the maximum prefix search procedure is applicable. The structure we define is an `IdentifierMap`, which is either maintained in memory or made durable. Both variants have the same interface. Here is an example of an in-memory map:

```
val map = tx.newInMemoryIDMap[Proc[S]] // map using maximum prefix keys

// for each new process in a collection, associate an auxiliary structure (view)
def procAdded(proc: Proc[S])(implicit tx: S#Tx): Unit = {
  val view = ...
  map.put(proc.id, view)
}

// when the process changes, update the view
def procChanged(proc: Proc[S])(implicit tx: S#Tx): Unit =
  map.get(proc.id).foreach { view => ... }
```

## 5.9 Composable Expressions

We have implemented a thin layer on top of events which provides a functionality similar to *Scala.React* signals. Like an STM source, an `Expr[S, A]`—an expression of type  $A$  in system  $S$ —has an access method to transactionally retrieve a value of type  $A$ . This method is called `value` instead of `apply`, because the latter is reserved for the equivalent of a reference cell, an `Expr.Var[S, A]`. The important difference is that an expression variable holds another *expression* instead of a flat value of type  $A$ . Flat values are lifted to expressions in the form of an `Expr.Const[S, A]`. These three basic types are shown in Listing 5.15.

Expressions provide an optional event through the `changed` method. An `EventLike` can be either a regular `Event` or a dummy with an interface compatible to `Event`. For example, constants do not actually dispatch events, as they are immutable. When seen as opaque expressions, one may

```

object Expr {
  trait Var[S <: Sys[S], A] extends Expr[S, A] with stm.Var[S#Tx, Expr[S, A]] {
    def changed: Event[S, Change[A], Expr[S, A]]
  }

  trait Const[S <: Sys[S], +A] extends Expr[S, A] {
    def changed = Dummy[S, Change[A], Expr[S, A]]

    protected def constValue: A
    def value(implicit tx: S#Tx): A = constValue

    def dispose()(implicit tx: S#Tx) = ()
  }
  ...
}
trait Expr[S <: Sys[S], +A] extends Writable with Disposable[S#Tx] {
  def changed: EventLike[S, Change[A], Expr[S, A]]

  def value(implicit tx: S#Tx): A
}

```

*Listing 5.15:* Expressions, and their special forms variable and constant

still attach observers to them, although this is a no-op. The event update type is `Changed` which simply wraps the previous and the new value of the expression:

```
case class Change[+A](before: A, now: A)
```

Expressions allow the composition of dynamic relationships between objects. This was discussed in a previous paper<sup>86</sup> under the name of a “fluent reference”: for example, one can specify that the beginning time of an object  $r_2$  (with respect to some timeline) is the ending time of an object  $r_1$  plus a certain gap. Whenever the interval of  $r_1$  changes, the interval of  $r_2$  is automatically adjusted. A hypothetical code could look like this:

```

def placeAfter(pred: Expr.Var[S, Span], succ: Expr.Var[S, Span], gap: Expr[S, Long])
  (implicit tx: S#Tx): Unit = {
  val newStart = pred.stop + gap
  val newStop  = newStart + succ().length
  succ()      = Spans(newStart, newStop)
}

```

<sup>86</sup>Rutz, Miranda and Eckel, ‘On the Traceability of the Compositional Process’.

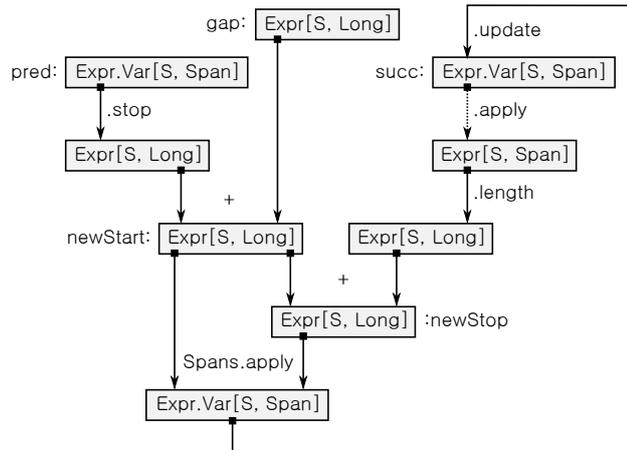


Figure 5.22: Expression chains produced by function `placeAfter`.  
Arrows point in dataflow direction from dependency to dependent.

The intervals of the predecessor region  $r_1$  and the successor region  $r_2$  are given as expression variables. We will see in Sect. 5.12 that this is a useful predisposition for editable processes. The interval type is `Span`, which is a value consisting of two `Long` fields `start` and `stop`, denoting the interval start (inclusive) and stop (exclusive) given in sample frames. The length of a span is thus `stop-start`. By using integer sample frames, values in  $\mathcal{T}_{(P)}$  are represented at a high resolution while operations are not exposed to floating point noise.

No primitive values, only expressions appear in the example: The selection `pred.stop` yields an `Expr[S, Long]`, and so does the binary operation for `newStart`. The content of `succ` will be overwritten, but we want to preserve the previous duration of `succ`. Calling `apply` on the expression variable is essential, as the resulting expression is not depending on that variable. If one looks at a graphical representation of the expression chains in Fig. 5.22, the variable application is shown dotted, indicating that the dependency is disconnected thereby. The disconnection prevents an infinite feedback which would otherwise occur with the arrow going back from the newly formed expression to `succ`.

As can be seen in the flow diagram, a modification either of the interval of  $r_1$  (`pred`) or of the `gap` expression will propagate all the way to the interval expression for  $r_2$  (`succ`).

```

trait BiExpr[S <: Sys[S], A] extends Expr[S, (Long, A)] {
  def time: Expr[S, Long]
  def mag : Expr[S, A ]

  def timeValue(implicit tx: S#Tx): Long
  def magValue (implicit tx: S#Tx): A
}

```

*Listing 5.16:* A bi-temporal expression associates a magnitude with a point in time.

## 5.10 Performance Time

Using expressions, directional relations between entities can be succinctly specified, breaking out of the “inanimate” perspective of a pure top-down data structure which forms the basis of persistence. This goes for relations in performance time as well, as has been shown in the previous section: Temporal expressions can be arbitrarily complex, but ultimately *evaluate to* equally spaced sample frames with respect to a given sample rate—e.g. 44.1 kHz— and a given anchor (reference to a timeline), a perspective useful for viewing, rendering and real-time sound synthesis.

One can now associate an expression of any type  $A$  with a moment in time, in order to place them in  $\mathcal{T}_{(P)}$ . For example, if we were to describe the development of a sound’s frequency, one can think of it as breakpoints across a timeline. We define a type for this pairing, `BiExpr`—for bi-temporal expression, since  $\mathcal{T}_K$  is already taken care of through system  $S$ —whose interface is shown in Listing 5.16. A `BiExpr` is itself an expression of the tuple type  $(\text{Long}, A)$ , allowing for constant, variable and combinatory bi-temporal expressions.

We define two collection types for bi-temporal expressions: `BiPin` and `BiGroup`. The former corresponds to a one dimensional breakpoint function, where no more than one nominal magnitude exists at the same time instant. Its interface is shown in Listing 5.17.

The three methods `at`, `floor`, and `ceil` return the elements at a given time or perform a nearest neighbour search in either direction. Because temporal values are specified as expressions, a strange situation may occur: Two elements might have a temporal expression which evaluates to the same point in time. But if this expression changes for either of the two elements, these

```
object BiPin {
  case class Update[S <: Sys[S], A](pin: BiPin[S, A], changes: IndexedSeq[Change[S, A]])

  sealed trait Change[S <: Sys[S], A]

  sealed trait Collection[S <: Sys[S], A] extends Change[S, A] {
    def value: (Long, A)
    def elem: BiExpr[S, A]
  }
  case class Element[S <: Sys[S], A](elem: BiExpr[S, A],
    elemUpdate: event.Change[(Long, A)]) extends Change[S, A]

  case class Added [S <: Sys[S], A](value: (Long, A), elem: BiExpr[S, A])
    extends Collection[S, A]
  case class Removed[S <: Sys[S], A](value: (Long, A), elem: BiExpr[S, A])
    extends Collection[S, A]

  trait Modifiable[S <: Sys[S], A] extends BiPin[S, A] {
    def add (elem: BiExpr[S, A])(implicit tx: S#Tx): Unit
    def remove(elem: BiExpr[S, A])(implicit tx: S#Tx): Boolean
  }
  ...
}
trait BiPin[S <: Sys[S], A] extends Writable with Disposable[S#Tx] {
  type Elem = BiExpr[S, A]

  def modifiableOption: Option[BiPin.Modifiable[S, A]]

  def at (time: Long)(implicit tx: S#Tx): Option[Elem]
  def floor(time: Long)(implicit tx: S#Tx): Option[Elem]
  def ceil (time: Long)(implicit tx: S#Tx): Option[Elem]

  def changed: EventLike[S, BiPin.Update[S, A], BiPin[S, A]]
  ...
}
```

*Listing 5.17:* BiPin, a bi-temporal breakpoint function

elements move to distinct points in time. Therefore, the underlying data structure must not discard an element even if another element moves to the same location, as it may well be “uncovered” at a later point in  $\mathcal{T}_K$ . The querying methods therefore return the nominal element at a time, which is the element at that time which was most recently updated.

The plain `BiPin` interface is read-only, providing an optional modifiable interface with methods for addition and removal of elements through method `modifiableOption`. The reduced functionality of the read-only collection allows it to appear as a transformed or filtered collection, similar to the «complex inspectors» proposed for *REScala*.<sup>87</sup>

In order to answer the queries efficiently, the implementation is backed up by a skip list (Sect. 5.6.4). This comes with a problem, however: The `BiPin` must observe those changes in the elements’ time positions which require a reinsertion for maintaining the ordering, a situation similar to the relabelling problem with the octree. We have experimented with different solutions. One can establish the dependency links from element to collection eagerly and permanently, instead of lazily waiting until the collection itself is observed. Then in a push phase, when reaching an unobserved collection, it must be either updated—requiring additional knowledge about deserialising a `BiPin`—or marked “dirty” so that the collection can refresh its knowledge of the time position upon the next query or observer registration.

The disadvantage is a more complex maintenance structure, and a single dirty marker flag means that upon verification, the whole list of elements must be traversed to ensure consistency. We are therefore sticking to a simple lazy solution, whereby the collection traverses the list of elements when the first observer is attached, setting up both the dependency links and verifying the time positions, which is equally expensive. Future work may examine this situation to provide a solution that performs better.

Finally, we shall look at the event structure of `BiPin`. Contrary to the fine-grained events discussed in the *EScala* paper,<sup>88</sup> we prefer to bundle the different types of updates under one

---

<sup>87</sup>Salvaneschi, Hintz and Mezini, *REScala: Bridging Between Object-oriented and Functional Style in Reactive Applications*.

<sup>88</sup>Gasiunas et al., ‘EScala: modular event-driven object interactions in Scala’.

compound event, more akin to event dispatch for example in Scala Swing.<sup>89</sup> Because an event can cause multiple related updates now—e.g. a time expression change may affect multiple elements—the update type is a sequence of `BiPin.Change` values. These in turn can be either collection related changes, such as an element being Added or Removed, or *forwarded updates* from an `Element` itself. This simplifies the observation of a collection, because there is one central observing function which is notified about any element in the collection, and the views do not need to add and remove observers on the individual elements as they appear and disappear in the collection.

The second collection type, `BiGroup`, sorts elements by intervals. Type `SpanLike` is a super type of `Span` which also includes unbounded intervals. For example, `Span.From(10000)` is an interval  $[10000, \infty)$ , a concept that is useful for real-time generation of sound processes which may be started at some point without knowing yet when they will end. Furthermore, any number of elements may overlap or coincide in time. The interface of `BiGroup`, shown in Listing 5.18, is similar to `BiPin`, except that the temporal annotation is a span instead of an instant, and queries return sequences of elements. Again the basic type is read-only, and `modifiableOption` provides an extended type with methods for adding and removing elements.

Elements stored are not necessarily expressions—indeed the main application is to collect the sound process instances `Proc`—and therefore an explicit element event update type `U` is required. The equivalent of `BiExpr` now is `TimedElem`. Events again combine collection insertions and removals with forwarded element updates.

The underlying data structure for efficient queries is a two-dimensional skip octree storing start and stop values of the spans in their respective axes. The root square spans the entire 64-bit range of possible locations (minimum and maximum encode the open interval boundaries), and appropriate rectangular shapes can be constructed for range queries, so as to find all intervals overlapping with a given other interval, or finding all intervals starting or ending inside a certain interval.

---

<sup>89</sup>Ingo Maier (Nov. 2009), *The scala.swing package*, Scala Improvement Process (SID) #8, URL: <http://www.scala-lang.org/sid/8> (visited on 29/06/2013).

```

object BiGroup {
  type Leaf[S <: Sys[S], Elem] = (SpanLike, IndexedSeq[TimedElem[S, Elem]])

  trait TimedElem[S <: Sys[S], Elem] extends Identifiable[S#ID] {
    def span: Expr[S, SpanLike]
    def value: Elem
  }

  trait Modifiable[S <: Sys[S], Elem, U] extends BiGroup[S, Elem, U] {
    def add(span: Expr[S, SpanLike], elem: Elem)(implicit tx: S#Tx): TimedElem[S, Elem]
    def remove(span: Expr[S, SpanLike], elem: Elem)(implicit tx: S#Tx): Boolean
  }
  ...
}
trait BiGroup[S <: Sys[S], Elem, U] extends event.Node[S] {
  import BiGroup.Leaf

  def modifiableOption: Option[BiGroup.Modifiable[S, Elem, U]]

  def intersect(time: Long)(implicit tx: S#Tx): stm.Iterator[S#Tx, Leaf[S, Elem]]
  def intersect(span: SpanLike)(implicit tx: S#Tx): stm.Iterator[S#Tx, Leaf[S, Elem]]

  def nearestEventAfter (time: Long)(implicit tx: S#Tx): Option[Long]
  def nearestEventBefore(time: Long)(implicit tx: S#Tx): Option[Long]
  def eventsAt(time: Long)(implicit tx: S#Tx):
    (stm.Iterator[S#Tx, Leaf[S, Elem]], stm.Iterator[S#Tx, Leaf[S, Elem]])

  def changed: EventLike[S, BiGroup.Update[S, Elem, U], BiGroup[S, Elem, U]]
  ...
}

```

*Listing 5.18:* BiGroup, a bi-temporal interval tree

```
trait Proc[S <: Sys[S]] extends event.Node[S] {  
  def graph: Expr.Var[S, SynthGraph]  
  
  def scans      : Scans      .Modifiable[S]  
  def attributes: Attributes.Modifiable[S]  
  
  def changed: evt.Event[S, Proc.Update[S], Proc[S]]  
}
```

*Listing 5.19:* Interface of a sound process

There are two variants of `intersect` which provide an iterator over all elements that exist *at* a given point, as well as elements overlapping a given interval (useful for painting a view port of a timeline display). The `floor` and `ceil` methods of `BiPin` have been changed for `nearestEventBefore` and `nearestEventAfter`. They only return a point in time, whereas the actual changes occurring at that moment can be queried using `eventsAt`. It returns an iterator for intervals stopping at that point, and an iterator for intervals starting at that point.

## 5.11 Creating Sound Processes

Using the persistent and the event layer, the notion of expressions and bi-temporal data structures, we are now ready to define sound processes. As already mentioned: The basic entity representing such processes is the `Proc` type, shown in Listing 5.19. With its three members and one event, this is a noticeable reduction from the first iteration of the *SoundProcesses* framework—which was ephemeral—where the interface had more than two dozen members.

Three factors helped to gain this rather minimal appearance. First, we apply the concept of MVC to “sounding representations” as well. Processes can be defined and manipulated now independently of a sound synthesis server running. Instead, any number of “aural views” (or `AuralPresentations`, as we call them) may be created which provide the actually sounding surface of a `Proc`. Second, the structure is more modular, with most functionality encapsulated by `scans` and `attributes`, which will be discussed in the next sections. Third, we reduced the number of mandatory information fields in favour of a general parameter map `attributes`, taking the advice from Eaglestone et al. that the composer should be able to make «free associations

```

val sg = SynthGraph {
  val sig  = PinkNoise.ar
  val bus  = graph.attribute("bus" ).ir(0)
  val mute = graph.attribute("mute").ir(0)
  val env  = graph.FadeInOut("fade-in", "fade-out").ar
  val amp  = env * (1 - mute)
  Out.ar(bus, sig * amp)
}

```

Listing 5.20: Example SynthGraph generating pink noise

between audio material». <sup>90</sup> It also keeps the interface open for inclusion of additional data, something we have used for the multitrack application *Mellite* (see Sect. 4.6).

### 5.11.1 SynthGraph as the Sound Descriptor

At the core of a sound process lies a declaration of a signal processing network, using *ScalaCollider*, a *Scala*-based client for the *SuperCollider* server. Since it is written in the same language as the rest of the framework, for the composer there is a seamless transition between the two, and it allowed us to integrate the APIs of *ScalaCollider* and *SoundProcesses*.

*ScalaCollider* has been described in detail elsewhere, <sup>91</sup> so we wish to focus here on the aspects relevant for the integration with the *SoundProcesses* framework. A *SynthGraph* is made from an anonymous function which declares a group of interconnected DSP building blocks, which typically constitute a self-contained sound entity. These building blocks, called graph elements and denoted by type *GE*, in most cases correspond to non-expanded *SuperCollider* UGens. Expansion is the process of unfolding graph elements into other graph elements or producing a number of UGens, often involving the duplication and variation of structures across multiple channels—called “multichannel expansion” in *SuperCollider*. Listing 5.20 shows the definition of an example *SynthGraph*.

Each line results in a graph element, sometimes composed of multiple elements, as is the case for the amplitude statement (two nested binary operator elements). Besides the familiar blocks from *SuperCollider*, the *PinkNoise* generator, the binary operators, and the *Out* element, we

<sup>90</sup>Eaglestone et al., ‘Information systems and creativity: an empirical study’.

<sup>91</sup>Hannes Holger Rutz (2010), ‘Rethinking the SuperCollider Client...’, in: *Proceedings of the SuperCollider Symposium*, Berlin.

```
def configure(proc: Proc[S])(implicit tx: S#Tx): Unit = {
  val bus      = Ints .newVar[S]( 0)
  val fadeInLen = Longs.newVar[S](44100)
  val fadeIn   = FadeSpec.Elem(fadeInLen, Curve.sine, 0.0)
  proc.attributes.put("bus"      , Attribute.Int      (bus  ))
  proc.attributes.put("mute"     , Attribute.Boolean (false ))
  proc.attributes.put("fade-in" , Attribute.FadeSpec(fadeIn))
  proc.graph() = sg      // (as previously defined)
  ...
  bus()         = 1      // adjust bus
  fadeInLen()   = 22050 // adjust fade-in duration
}
```

*Listing 5.21:* Configuring parameters of a sound process

introduce a number of new elements in the graph package. The function `attribute` looks up a numeric scalar value in the `attributes` field of a sound process. Most attributes are based on expressions, so they obey dataflow rules. Listing 5.21 shows how a process is configured with the `SynthGraph` of Listing 5.20 and some of the related attributes.

Here `bus` and `fadeInLen` are expression variables, while `fadeIn` is a composed expression, and the `mute` attribute is set to a constant expression. Because of the heterogeneous nature of attributes, retrieval from the map only returns attributes when they match the expected type, so it is important to establish conventions for the names and types of attributes. The graph elements always require a meaningful default value. Where `graph.attribute` has a function similar to controls in *SuperCollider*, the `FadeInOut` element takes attribute names and generates an envelope spanning the duration of the process in a group (see Sect. 5.11.4). Since the fade-out attribute is not specified, the envelope will just end abruptly. The same would happen if the key was present but the value type was not `FadeSpec`.

### 5.11.2 Graphemes

Attributes evolve in  $\mathcal{T}_K$  but not  $\mathcal{T}_{(P)}$ . This makes sense for some of them, e.g. how would a fade-in specification vary while the sound is playing? Others arguably could be conceived as varying across a performance timeline. Standard harddisk recording applications allow the automation of a mute flag. They do not allow the automation of the bus routing (other than

panorama position or send volume), because it would present some challenges for the graphical user interface, but technically they could.

For bi-temporal values, this is where *graphemes* come in. Evidently inspired by Derrida's terminology, a grapheme is a material trace produced by a writing operation. The type `Grapheme` is a specialisation of the `BiPin` data structure—thus a sort of one-dimensional time function—where elements are bi-temporal expressions evaluating either to `Grapheme.Value.Curve` or `Grapheme.Value.Audio`. A `Curve` is a break point magnitude associated with a slope type such as step function, linear ramp, exponential curve, etc. The magnitude can be a single element or a vector to produce multichannel expansion. An `Audio` element is a tuple consisting of an audio file *artefact*, an offset into that file and a gain factor. In other words, a grapheme produces a performance time signal (monophonic or multichannel), composed of segments which are either break point functions or audio file fragments or combinations thereof.

An audio file `Artifact` is a reference to a file on harddisk, associated with an `Artifact.Location`, relative to which the file is found. This allows a project to be transported to another harddisk. If subsequently the location is updated, the artefacts should be found again in the new location. An `Audio` element also caches information about the artefact, such as duration, sample rate and number of channels, so the code can still operate on it in the absence of the physical file.

### 5.11.3 Scans

If graphemes are understood as wax cylinders, it remains to define a needle to carve into the wax and to scan the wax. This is done by a `Scan` instance. Scans are situated within the `scans` field of a `Proc`, which—similar to attributes—maps between names and these `Scan` instances. A scan is a connecting point, it administrates sinks (the writing action) and sources (the reading action). A sink or source may be either a grapheme or another scan. A grapheme, since it is a trace, can be accessed both in real-time and offline. The scan output on the other hand always denotes a *real-time* signal. A scan signal is produced either by linking the scan's source to another scan's sink—thus establishing “bus routing” between processes—or a grapheme input, or it is produced by the process itself, using the special assignment `scan("name") := signal` inside the process' graph function. This is illustrated in Fig. 5.23.

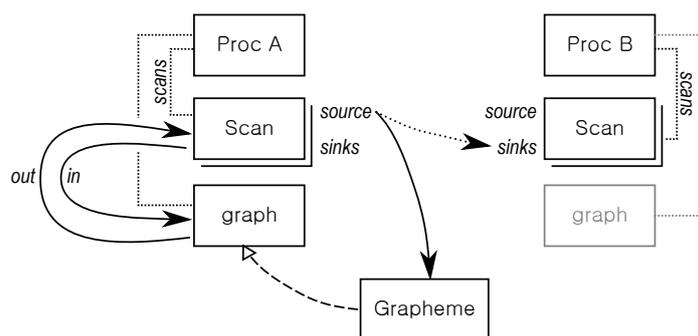


Figure 5.23: Interaction between scans, graphemes and graph functions

In order to turn a real-time signal into a manifest trace, one must link a scan sink with a grapheme. This functionality is not implemented yet in the current version, but it can be easily worked around: One allocates a new artefact (file location) and writes the signal directly to it using a standard `DiskOut` UGen. Upon finishing the recording, the artefact is added to the grapheme. This is indicated by the dashed arrow in Fig. 5.23. Future versions will elaborate this interface, when we have gathered more experience with using *SoundProcesses* in live improvisation and installations. For the experiments conducted with the *Mellite* application, this type of routine was not necessary.

#### 5.11.4 Transport

Having bi-temporal structures is satisfactory for offline viewing, but a mechanism is still missing that provides a clock for  $\mathcal{T}_P$  to pass. This is this the responsibility of `Transport`. Its interface, shown in Listing 5.22, is straightforward: There are methods to start and stop the transport, to query and update its current position, a sample-rate reference, and a method `react` to register observers. Internally a transport keeps a `BiGroup` of the elements which are to be transported.

Technically, transports do not define an event, because they cannot be serialised—their time pointer continually advances—but `react` has the familiar form. The update value dispatched can be either of `Transport.Play`, `Transport.Stop`, or `Transport.Advance`, the last carrying a large amount of information: The current time of the update, whether the advancement was due to a re-positioning (`seek`) or a regular scheduler event, the elements added and removed at the update time, as well as any changes in the bi-temporal data of the elements themselves. For instance,

```

trait Transport[S <: Sys[S], Elem, U] extends Disposable[S#Tx] {
  def play()(implicit tx: S#Tx): Unit
  def stop()(implicit tx: S#Tx): Unit
  def isPlaying(implicit tx: S#Tx): Boolean

  def seek(time: Long)(implicit tx: S#Tx): Unit
  def time(implicit tx: S#Tx): Long

  def sampleRate: Double

  def react(fun: S#Tx => Transport.Update[S, Elem, U] => Unit)
    (implicit tx: S#Tx): Disposable[S#Tx]
  ...
}

```

*Listing 5.22:* The Transport interface scans a group of processes in  $\mathcal{T}_P$ .

if the transport carries Proc elements, a change might be that a break point in any graphemes scanned by the sound processes was reached.

A transport maintains a last reported position (LRP), and the list of currently active elements (CAE), i.e. the elements whose interval contains the LRP. Furthermore, after each advancement it calculates the next interesting position (NIP) which is the next point in time where any sort of update event must be emitted.

The `seek( $t$ )` command may be implemented as follows:

- › If  $t > \text{LRP}$ : In the BiGroup, find the elements which end in the interval  $(\text{LRP}, t]$  and which start in the interval  $(\infty, \text{LRP}]$ . Remove these elements from the CAE. Find the elements which start in the interval  $(\text{LRP}, t]$  and which end in the interval  $(t, \infty)$ . Add these to the CAE.
- › If  $t < \text{LRP}$ : In the BiGroup, find the elements which start in the interval  $(t, \text{LRP}]$  and which end in the interval  $(\text{LRP}, \infty]$ . Remove these elements from the CAE. Find the elements which end in the interval  $(t, \text{LRP}]$  and which start in the interval  $(\infty, t]$ . Add these to the CAE.

For the CAE, three structures are maintained:

- (1) An identifier-map (cf. Sect. 5.8.2) from elements to a map from scan keys to grapheme segments which are the sources of these scans. A grapheme segment is the segment between the two break points containing the LRP.
- (2) A skip list used as priority queue, where keys are time values denoting the NIPs. The values stored are maps from element identifiers to a map from scan keys to grapheme segments (as in the previous structure).
- (3) An identifier-map from elements to themselves, which is used to get fresh copies of them.

Furthermore, an information structure is updated at each advancement, which stores the system realtime clock, the current frame position, the state (whether the transport is stopped or playing), an NIP for processes beginning and ending, and an NIP for grapheme changes.

Based on these structures, the transport can determine at each step what the next stopping point is, and upon arrival, which kinds of updates are to be performed. Because transactions may change the transported elements at any point in time—between advancements—a transaction local logical time variable is maintained. In order to issue transactions, the transport also has access to a cursor (Sect. 5.6.2). In real-time mode, when the scheduler executes the next advancement, it issues a cursor step and sets the local time variable to the exact value corresponding with the reached sample frame. If a B1Group update is observed, the logical time is interpolated according to CPU elapsed since last advancement.

Revalidating the information structures, such as NIPs, current grapheme segments, and CAE—and emitting the corresponding transport update event—is a rather involved and lengthy process, considering all the possible updates of sound processes. As an indicator, the implementation of Transport has approx. 800 SLOC<sup>92</sup>.

Finally, the transport can also be run in offline mode. Here, instead of a realtime clock based scheduler, the advancement is issued explicitly by the use site. Offline transports can be used for debugging, but more importantly to “bounce” sound processes to disk, e.g. using *SuperCollider*’s non-realtime mode.

---

<sup>92</sup>Source lines of code, excluding blank lines and pure comment lines.

### 5.11.5 Aural Presentation

So far, the transport is only a scheduling mechanism that emits updates about which elements appear and disappear over time, and how their internal structure (graphemes) changes. In order to make sound processes audible, an instance is needed that observes a transport and builds sound synthesis representations for the Proc elements. This is the purpose of `AuralPresentation`.

To construct sound synthesis objects, it uses an API similar to *ScalaCollider*, but with transactional semantics. When calling `play` on a `Synth`, when creating or deleting a `Group`, a `Bus`, or a `Buffer`, these would have immediate side effects in *SuperCollider* or *ScalaCollider*: a node, bus or buffer identifier would be allocated, OSC messages would be sent to the server (an action that cannot be revoked)...

All these objects—synth definitions, nodes, buses, buffers—implement a `Resource` trait which tracks their status on the server side, along with a time stamp corresponding to the last logical update made to the resource (e.g. allocating a buffer, reading contents to a buffer). Instead of sending OSC messages directly to the server, they are managed by the transaction. In the *SoundProcesses* framework, a new system sub type is defined that introduces a method `addMessage` to the transaction. Along with the resource and OSC message, a list of dependencies on other resources is passed to this method.

Playing a `Synth` will depend on the `SynthDef` being online, and perhaps a `Buffer` having been allocated and filled with content. The transaction will sort the messages—which, since they occur within the same transaction, belong to the same logical moment in time—according to their dependencies. Only when the transaction successfully completes, will these messages be bundled and sent out, taking care to inject synchronisation for asynchronous commands such as `/d_recv`, `/b_alloc`, or `/b_read`.

The task for `AuralPresentation`, however, is more difficult. Within a transaction objects may appear and disappear, and the interdependencies between processes are *declarative*. For example, if a process  $p_1$  is created and added to a transport, it is observed by the aural presentation which will try to build the sound synthesis representation of that process. The `SynthGraph` might be set

only in the next step—all this happens in the same transaction!—so that will also be observed, and the view needs to be adjusted. The synth graph may refer to a scan that is not yet initialised, so in the case of insufficient information the view building procedure needs to be put on hold. If a scan is connected to another scan—linking an output of one Proc to an input of another Proc—the scan’s number of channels cannot be determined until the synth graph function of the dependency is known. If at the end of the transaction some Proc views are left dangling, it must be ensured that any audio buses allocated for them are released again.

For minimum latency, the processes should ideally be topologically sorted. This was done in an early version of the framework, while in the current version this sorting is omitted for reasons of simplicity. Mapping buses to controls via `/n_mapa` uses a mechanism similar to the `InFeedback UGen`, so there is no strict need to have the topological sorting. In fact, other libraries such as *SuperCollider’s JITLib* use the same relaxed conditions.

Having `AuralPresentation` as a separate entity in the sense of a view in MVC offers several advantages. An application can be run “headless” without any actual sound production, while the transport is still fully functional, as well as other views attached to the transport, such as timeline or realtime visual view. The model is also cleanly separated from the sound synthesis invocation, making the code less cluttered. It is possible to run multiple aural presentations for the same model—perhaps a live mix versus a headphones mix—and it allows the swapping of the standard implementation for others without affecting the rest of the code base. To conclude, the aural presentation can run in offline mode, too, allowing a straight forward “bouncing” of anything that is carried by a transport.

## 5.12 Editing Sound Processes

We want to emphasise that *SoundProcesses* is a *framework* not a monolithic application. There are various ways to build computer music applications using this framework. Chap. 4 has shown how we used it bare-bones in the *Voice Trap* installation, and we also built and used a graphical multitrack editor *Mellite* to compose different tape pieces. Other installations reviewed used previous versions of the framework and thereby informed the design as it was presented in the preceding sections.

In *Mellite*, a “document” contains a tree of document elements and a tree of cursors for branching into  $\mathcal{T}_K$ . The idea of the document element tree is again to allow a free organisation of a workspace by the composer, and we envision eventually having an interface similar to *Open-Music*, where objects can be spatially arranged and associated with other attributes such as colours and names. In the current version, elements are associated with names and organised in a tree view.

Elements can be `BiGroups` (timelines), audio file artefacts, code fragments and simple expressions such as numbers or strings. See Sect. 4.6 for more details and screenshots. Audio files and timelines can be opened in visual editors resembling common multitrack applications. Using the cursors view, multiple versions or branches can be opened at a time, although we have not experimented yet with graphically dragging and dropping elements across branches.

Audio file regions are represented by instances of `Proc` associated with a grapheme containing the file artefact. Process attributes are used to control fade-in and fade-out curves, mute status and gain. The interface assumes `Expr.Var` for editing most of these attributes, and this has proven to be a useful approach, as the expressions propagate automatically to different views and, although the graphical editing of expressions is currently limited to entering constants for variables, composed expressions are automatically accommodated. One can for example define an integer variable in the elements tree view, and drag and drop it to assign it as the channel parameter of multiple processes. Then changing that variable will be reflected in all those processes.

Furthermore, there are function regions defined by a custom synth graph written as a code fragment compiled on the fly by an embedded *Scala* compiler. Scans can be visually connected through patch cords. Global processes with an indefinite duration (`Span.All`) are separately listed outside the timeline and can be used to implement panning and routing of regions.

### 5.13 Summary

This chapter has shown how a general computer music framework can be designed which allows the tracing of the manipulations of objects represented therein. We have designed an

abstraction called system  $S$  which provides the minimum necessary interface to accommodate different modes of observation: in-memory and ephemeral, durable and ephemeral, durable and confluent persistent. Mutable objects have unique seminal identifiers  $S\#ID$  and histories  $S\#Acc$  and the parts which can mutate are represented by variables  $S\#Var$ . Furthermore, the system provides atomic isolation through transactions  $S\#Tx$  which may accommodate a durable back-end, here realised with a key-value store.

A number of data structures and algorithms have been implemented, and one should be reminded of the famous quote from D. E. Knuth: «Beware of bugs in the above code; I have only proved it correct, not tried it.»<sup>93</sup> There is a huge gap between devising such structures and algorithms and actually making them work, often requiring adjustments or replacements of certain assumptions. A large part of the chapter describes our solution to implementing a confluent persistent system which can also be efficiently stored on disk and which behaves transparently from the usage side of things. One of the most beautiful aspects of this process was to see how the data structures re-enter the system recursively: For example, the octree, used in our geometric solution to the marked ancestor problem, is implemented *against the system façade*, using the mutable object model described above; we then instantiate the octree using the durable and ephemeral sub-system in order to build to infrastructure of the confluent super-system. A brief discussion and visualisation of this development process can be found in Sect. 6.2.1.

Our system was then equipped with a reactive event layer and we experimented with expressions inspired by the dataflow paradigm. Bringing in  $\mathcal{T}_{(P)}$ , we extended these to bi-temporal expressions, and added collection types  $BiPin$  and  $BiGroup$  as performance-time indices. While navigation in  $\mathcal{T}_K$  occurs through a  $Cursor[S]$  which initiates transactions, the real-time unrolling of  $\mathcal{T}_P$  is accomplished through a  $Transport$  which couples  $\mathcal{T}_K$  (a cursor) and  $\mathcal{T}_{(P)}$ . Our suggestion for computer music systems was a basic type  $Proc$  carrying an open dictionary with parameters, a set of signal ports called scans, and a DSP function implemented in *ScalaCollider*. We further suggested decoupling transport from sound synthesis production by realising the

---

<sup>93</sup>Donald E. Knuth (Mar. 1977), 'Notes on the van Emde Boas construction of priority deques: An instructive use of recursion', *Classroom notes Stanford University*.

latter as an “aural view” attached to the model of a `BiGroup` of `Proc` instances which is read by a transport (which in turn could be understood as the controller in MVC).

There are many aspects to develop in a future iteration of the framework. To name a few:

- › How can indices such as `BiGroup` maintain their correct ordering when the indexing expressions change? Currently, the indices must be observed, i.e. have a view or transport attached to it.
- › How can we devise a general interface for defining and maintaining indices, not just over  $\mathcal{T}_{(P)}$  but any parameter?
- › How can the expression system be opened and possibly include function parameters, e.g. as would be required for monadic operations such as `map` or `flatMap`?

These and more questions are also found in the final discussion of Sect. 6.4 and Sect. 6.5.



In fact, Marker died during the thesis writing process, on 29 July 2012, and I remember reading his obituary which was published in *The Guardian* the following day.

## 6.1 Discussion

The “perfect” memory, of course, is an illusion which has been discussed in two ways. First, due to the irreducible limits of control. Observation is always partial and leaky (Fig. 3.7), it equally signifies observed and observer. In cybernetics, memory and control are linked in such way that building a memory of a system’s observed (partial) past states may compensate for the inability to observe the whole system as it improves prediction and thus control.<sup>2</sup> So to be precise, what is an illusion is the perfect *exploitation* of a memory. The idea that there could be complete control is debunked in the following quote from William S. Burroughs’ essay *The Limits of Control*: «All control systems try to make control as tight as possible, but at the same time, if they succeeded completely, there would be nothing left to control.»<sup>3</sup>

The control dilemma lies in the fact that if a subject is completely controlled, it does not exhibit any form of resistance or disobedience, so one can hardly speak of control. In terms of an artistic endeavour, artists often work precisely with the opposition or friction of the material, they subject themselves sometimes to artificial constraints, because this opposition functions as a generator of ideas. The perfect computer music system is not only an illusion, it is also not something worth pursuing. But paradoxically, designing a non-perfect computer system is a difficult task. Every computer program essentially *is* a control machine in terms of the technology it relies on, such as data structures and algorithms, at least in the discourse of computer science which embeds these technologies. We can think of a Deleuzian rhizome, but we can only implement genealogical data structures, trees, directed graphs etc. This is perhaps a choice we made, since there are other non-hierarchical elements out there, evolutionary algorithms, neural networks and so on. But they will be encumbered with other double-binds such as goal directedness and correspondence of patterns.

<sup>2</sup>cf. W. Ross Ashby (1956), *An introduction to Cybernetics*, London: Chapman & Hall, §6/21; Ashby is careful enough to put “memory” in inverted commas.

<sup>3</sup>William S. Burroughs (1978), ‘The Limits of Control’, *Semiotext(e): Schizo-Culture* III(2), pp. 38–42.

Burroughs draws an interesting analogy between the fully controlled subject and a *tape recorder*. He argues that control ceases, because when the subject loses its will, it is no longer a subject but merely machine. The machine as a known sub-routine appears as the technical object in Rheinberger's dichotomy. In order to produce knowledge, it must be destabilised, so it can oscillate between this state of technical object and the indeterminate state of epistemic thing. Therefore it is important that the computer framework we developed, while an achievement in its own right, is embedded with the other writing processes of this thesis to become a vehicle for the generation of epistemic value. These writing processes are: The investigation into the double nature of composition, signified by the interaction between two layers of time, the critical writing which exposes the methodological implications of ascertaining 'process', the writing of music pieces and sound installations, and their re-writing in terms of the transversal reading of Chap. 4.

The second illusory aspect of memory is representation. We have a handle, a signifier, which claims to represent another thing. Re-presenting, bringing back to presence, is often thought of as a constant, a repetition of the same. This is the tape machine, the mechanical reproduction Walter Benjamin talks about.<sup>4</sup> In electro-acoustic music, we are sound surgeons who can produce precise cuts. They can be stored in the trim bin, or any digital replacement of this random access container.<sup>5</sup> The trim bin is our illusion of being outside time (Fig. 2.9), detached from the composition-as-product which makes us forget that we are attached to composition-as-process. What I tried to show in Chap. 3 is that the models and representations we have of the compositional process do not challenge this detachment. Even if they might have emancipatory aspirations, they hardly go beyond the diaphragm model of composition (Fig. 2.1) in the sense that the diaphragm acts as the gravity centre which attracts all explanation. The only option for ignoring that model seems to be a cognitive approach, but it in turn ignores the essence of the compositional process because it identifies it with an immaterial creative process which is located somewhere in the "mind". If tracing machines are used, then they appear in the function

---

<sup>4</sup>Walter Benjamin (1936/1963), *Das Kunstwerk im Zeitalter seiner technischen Reproduzierbarkeit*, Frankfurt a.M.: Suhrkamp.

<sup>5</sup>Cf. Hanns Holger Rutz, Eduardo Miranda and Gerhard Eckel (2011), 'Reproducibility and Random Access in Sound Synthesis', in: *Proceedings of the 37th International Computer Music Conference*, Huddersfield, pp. 515–522

of tape recorders which are merely surrogates for a “direct” observation which is stained by the introduction of an “unnaturalness” of the inquiry situation.

Representations are powerful and necessary nevertheless. The handles they provide allow for a suspension of time, but more than the suspension of the translation procedure of  $\mathcal{T}_K$  into  $\mathcal{T}_{(P)}$ —what I have called spatialisation—the crucial suspension is the one purely within  $\mathcal{T}_K$ . In a first approximation, we can think of delay line memory, an interesting component of early electronic engineering. Perhaps because Spencer-Brown also had a background in electronic engineering, this aspect appears in the *Laws of Form* as some of the paradoxes are resolved by way of claiming time to pass between the left and right side of an equation. Luhmann, certainly inspired by Spencer-Brown’s work and that of the cyberneticists, at one point defines memory as the «capacity to delay the repetitive use of forms.»<sup>6</sup> As another example, general systems theory can be mentioned, where this kind of memory would be the stable oscillation as opposed to a resonance disaster or a damped oscillation which would “forget” the past.

In *(Inde)terminus* I tried to exploit the suspension in  $\mathcal{T}_K$  as the potentiality of recursion. The necessary representation is one which is operationally closed, so that the product of one iteration can be the input of the next. The other, technologically less manifest suspension is the “interface” between human composer and computer. Any “derivative intentionality” is baggage that belongs to the double-bind scenery. Or as Donna J. Haraway put it in her *Cyborg Manifesto*: «It is not clear who makes and who is made in the relation between human and machine. It is not clear what is mind and what body in machines that resolve into coding practices.»<sup>7</sup> It is unfortunate that the mind/body distinction is still used despite being questioned. We want to go further and only ever speak of the material traces produced by the indirection of having an inter-face. In the text produced with respect to a series of pieces from my early *Residual* to *Zeichnung* and *Dissemination*, I tried to observe these traces of process by looking at the eigen-dynamics of procedures and notions applied such as ‘mobile’ or ‘similarity’. The representation is thus a

<sup>6</sup>Niklas Luhmann (1993), ‘Deconstruction as Second-Order Observing’, *New Literary History* 24(4), pp. 763–782.

<sup>7</sup>Donna J. Haraway (1991), ‘A Cyborg Manifesto: Science, Technology, and Socialist-Feminism in the Late Twentieth Century’, in: *Simians, Cyborgs and Women: The Reinvention of Nature*, New York: Routledge, pp. 149–181.

translation, and as the observation continues, new diagrams and new translations are produced. In other words, we employ a differential reproduction which is upheld by temporal coherence—the continuity of the observation—groping and grasping for differences which produce a contour of the otherwise unobservable space or medium of ‘process’ which pervades the selected pieces.<sup>8</sup>

With this I conclude the overarching discussion. The next section will talk about the process of the thesis itself, as the re-entry of the form is a critical part of its methodology. Sect. 6.3 and Sect. 6.4 will more narrowly recount the contributions and limitations of my investigation, and finally Sect. 6.5 convenes a number of junctures from which future research may depart.

## 6.2 Process of the Thesis

I would like to have this thesis understood as the delineation of something that obviously spans a larger temporal frame than the last four years of my life. It acts as a formalisation of ideas which can be traced back to the earliest piece discussed, *Residual* from 2002. Even though it lies back more than ten years, and everything has changed and shifted over time, it was founded on elements which are still relevant today. Foremost, the possibility to compose music and sound art by building computer programs, but not as a correspondence of a formal plan or intention we construct in our heads. Instead these programs are our experimental systems, our laboratory apparatus, part of the “whole commitment”, as Rheinberger calls it.

The apparatus is necessary for experiments to succeed, it is thus false to think of it as a utility for an activity which could be otherwise carried out with paper, pen and “mind”. Sometimes the effort put into the apparatus seems disproportionate to the amount of work “realised” with it. But, to remain with Rheinberger, the maintenance of the apparatus ensures that the epistemic thing does not «dissipate», and he remarks that for an empirically working scientist the servicing of the apparatus can easily take up to 90% of his working hours.<sup>9</sup>

---

<sup>8</sup>Cf. Hans-Jörg Rheinberger (1997), *Toward a History of Epistemic Things: Synthesizing Proteins in the Test Tube*, Palo Alto: Stanford University Press, ch. 5

<sup>9</sup>Hans-Jörg Rheinberger (2nd July 2008), ‘Epistemische Dinge—Technische Dinge’, *Bochumer Kolloquium Medienwissenschaft*, URL: <http://vimeo.com/2351486> (visited on 28/08/2012), 16’.

### 6.2.1 On the Evolution of the Apparatus

The manifest temporal traces of the development and “servicing” of the software framework are captured by the *git* versioning system and illustrated in Fig. 6.1. When I say development and servicing, I am hinting at the oscillation between technical and epistemic level which happens in this process. In the beginning of each trajectory, there is still very much the groping and grasping without knowing exactly how the development will play out and which paths it will take. A close look at the individual transaction commits can reveal how crystallisation happens, the arrival of plateaus of relative stability characterised mainly by “bug fixes” or “refactoring”, and a decrease in the frequency of radical changes.<sup>10</sup>

The illustration of the *git* repositories runs into the same problems which have been demonstrated with the piece *Dissemination*: Only a selection of components is shown, and therefore the decision about when a particular (sub-)process begins is ambiguous. The software components use prior experience gained through the development of earlier components, for example the programs *Fscape*, *Bosque*, *Wolkenpumpe* and *Kontur*, which have been mentioned in previous chapters, are not shown. Neither are the repositories of the code for the individual pieces shown or sketches made before. Likewise the thesis text itself, which is shown in the last row, looks isolated and as if it takes off only in spring 2012, when in fact it incorporates a number of texts previously written, most literally the conference papers referenced, but more fundamentally all the pieces of sound art which were translated. It is thus useful to view these diagrams rather as representations of selected “form parts” of the overall writing process. Perhaps the most important part missing are my notations of papers and books I read, which participate in the upper half-wave of the phase model of writing (Fig. 4.1).

Some general motions can be observed, nevertheless. For example, there is a form of confluence in the writing of the software. *ScalaCollider* is the earliest component here, along with the first version of *SoundProcesses*. Although the latter is shown to join in July 2010, its first commit already establishes a large body of classes, so its writing began earlier and I just lost track of the initial phase. Concurrently I had started work on elaborating a confluent persistence

---

<sup>10</sup>Such detailed analysis is beyond the scope of this concluding chapter, and thus we will just highlight a few examples in a general form.

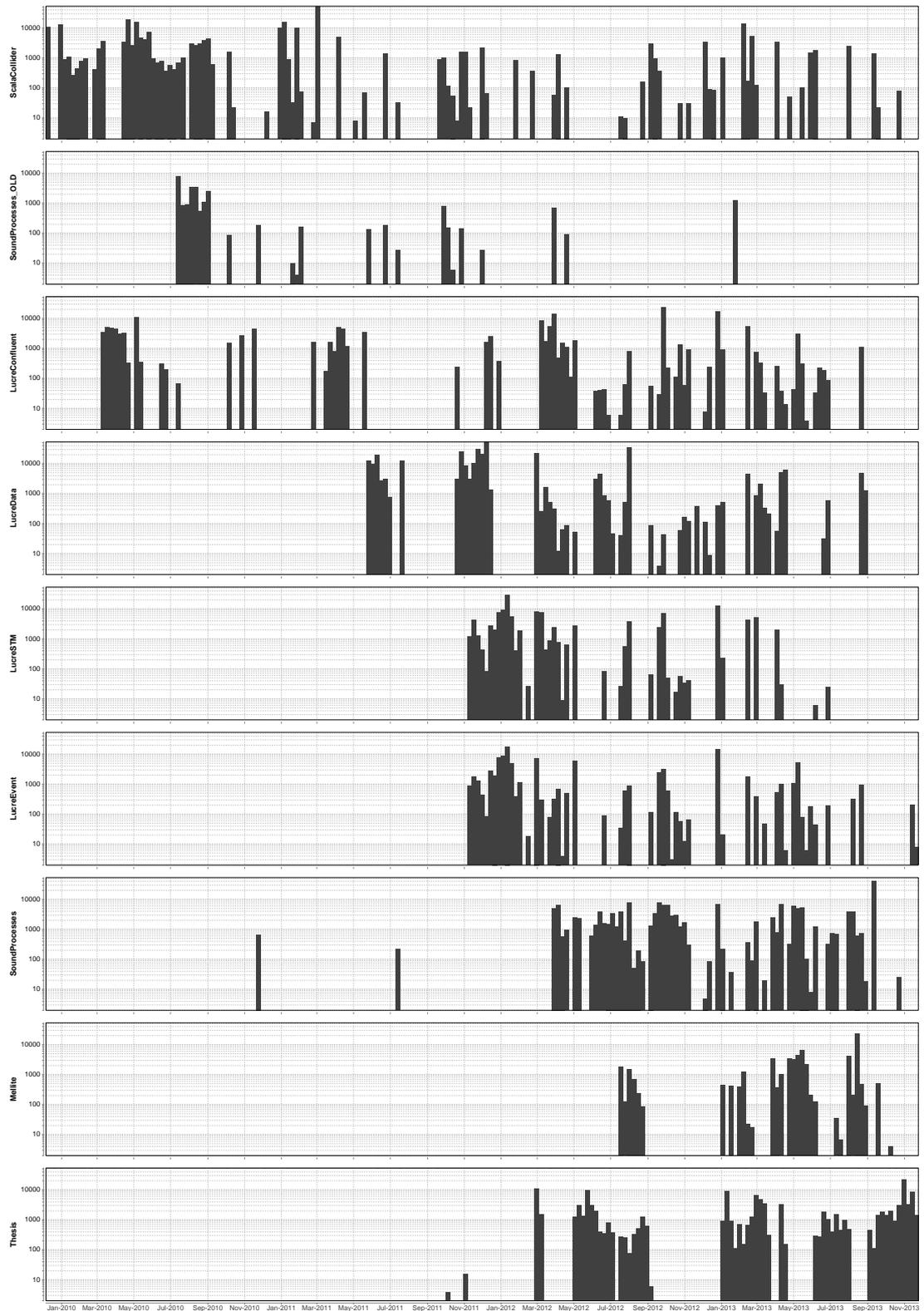


Figure 6.1: Work on the software framework over time. The git history of the major components is shown with line insertions and deletions aggregated per week. The development of the thesis'  $\LaTeX$  source is shown for comparison in the last row.

based system (*LucreConfluent*, the third row). The first big batch of work prepares the SMC 2010 conference paper. At that time, it still had a *different name*, *TemporalObjects*. Only with the modularisation of the separate *LucreSTM* branch, this component began, in September 2012, to act like a piece in the overall jigsaw puzzle (merged with the new API). Likewise, the *SoundProcesses* stream with the instantiation of its new version in May 2012 received its fixed position in the puzzle.

Another thing to observe is how forms are drawn out of other forms (or functionally differentiated in terms of Fig. 5.1). *LucreData* started in May 2011 with the attempt to implement some data structures described by Alstrup, Husfeldt and Rauhe. It goes without saying that there was an entire process of learning and understanding what data structures are, and how computer scientists think about and write about them; in sum, how they “approach” and “produce” the idea of a data structure. In this particular case, there were some unclear parts regarding a partitioning into micro trees and the main authors could not be reached for specific comments, as they had abandoned their career in computer science or did not remember the circumstances.

I went on with the new idea—merely *keeping the name* of the project—of using a spatial data structure for NN search, implementing the quadtree, then the compressed quadtree, then the randomised skip quadtree, then the total order, skip list and deterministic skip quadtree. In June I began elaborating the idea of isomorphic mapping between full trees and marked trees. I came across the path bending problem and the need to modify the NN search. In October, the quadtree was generalised to an octree, and the actual marked ancestor search and quasi-retroactive modification structure appear. In November, work began on using STM cells instead of mutable fields, taking the skip list as a relatively easy structure first. Then a step is made which gives birth to the new project and name *LucreSTM*, abstracting a new interface from *ScalaSTM* which will then be used back in the data structures project.

Similarly, writing the event system and dataflow expressions emerge as part of *LucreSTM* in December 2011, and only in September 2012 the repository is “forked” into a new project *LucreEvent*, filtering the relevant subset of the source code.<sup>11</sup>

---

<sup>11</sup>Thus in the figure, the diagrams for *LucreSTM* and *LucreEvent* look almost identical in the beginning, the differences coming from the way the directories are filtered.

### 6.2.2 Exceeding the Semantic Horizon

What do I mean when I say the thesis writing produced a re-entry? After all, is not every writing about writing a re-entry? There are two crucial differences concerning the dissolution of the meta-level (Sect. 3.3). First, there is a reciprocal exchange between these different layers of writing, and second there are certain motions in one layer which are reproduced in another, even though the context changes—writing a scientific document versus experimenting with computer science and engineering a piece of software versus composing a piece of music.

That these streams ran concurrently over the last few years and not in a strict succession is obvious from the discussion of Chap. 4 and Fig. 6.1, even if the conference papers, sketches, pieces etc. are not included in this figure. What is also missing in the figure is the vertical motion between these streams. The analysis of the pieces of music in Chap. 4 is an evident motion from those pieces to this text, and there were several occasions where it should have become clear that the ongoing writing of the framework became effective in the more recent pieces, too. Some translations were very manifest, such as *Dots* using the source code as its material, *Unvorhergesehen–Real–Farblos* using the purple visualisation technique of self-similarity, and all of the pieces of the last few years addressing in some form the intrusion of the compositional time into the performance situation.

There are many more threads which run between the different types of “texts”, too many to be included in this limited space. One of the strongest observations I was able to make is that essentially I write this text *like* I compose and vice versa. Whether this is a psychological or cognitive condition or rather the intrinsic play of the material and the writing machines may be debatable. But since these two approaches, the first of which I rejected and the second of which I embraced, are not to be misunderstood as mutually exclusive, it is perfectly possible to establish evidence—or to construct narratives!—which support either perspective. We did not make the effort to develop a motiongram similar to the ones shown in Fig. 4.51 for the thesis text itself, although this might be an interesting future project,<sup>12</sup> but I will exemplarily highlight this structural reappearance.

---

<sup>12</sup>If one wanted to use the existing *git* history, one would have to develop a LaTeX parser to be able to produce an approximation of these motiongrams.

It begins with the spreading out of materials. Since the “canvas” is just a set of LaTeX files, these materials usually appear in empty spots towards the end of a respective chapter, and there is at first no particular ordering except for blocks which are separated by white space or LaTeX comment lines. Then as I develop parts of continuous text, I repeatedly scan these blocks of ideas, move or copy the relevant parts into the location at which I am currently writing. Ultimately there are things left over which are relevant but “do not fit”. They remain back up for other parts of chapters of the text. If I do not use them for a while, I comment them out, like I mute a sound region in a tape composition, so that they are still represented in the editor although not in the product. Most of the misfits are deleted at some point.

The rhythm is a play between periods with constant typing of characters and periods characterised by almost infinite interruptions between two or three words or sentences, caused by my searching for something that I vaguely remember but cannot grasp. I may end up in three or four nested levels of references, all of which produce tiny material elements, small notations, which may not go into this insurmountable gap between the two words that triggered the search, but which end up again in a sketch file or in one of the raw blocks of thoughts at the end of each chapter file. For example, while I was writing the last sentence, the word “linearisation” entered my consciousness. I am lost as to where this comes from; using the apparatus of my notations and a search index, I come to find this reminder of Luhmann from the preface of *Art as a Social System*:

«One should not expect a linear order, progressing from important to less important issues or from prior to subsequent events. I hope that the reader’s understanding will benefit from the recognition that conceptual or historical materials reappear in different contexts.»<sup>13</sup>

I *know* that this is not the occurrence of text linearisation I was thinking of before, but I have to go on, and so I take what the *interface* produced and incorporate it. In the *suspended time*

---

<sup>13</sup>Niklas Luhmann (2000), *Art as a Social System*, trans. by Eva M. Knock, Stanford: Stanford University Press, p. 5.

of writing, diverted by searching for the term “linear”, I added the following to the “heap” of unsorted things at the end of this chapter file:

the incredible sensation that by writing this sentence (on 1 December 17:52h), the diagram could be reactualised and look different

This sentence will appear probably in Sect. 6.3. I have to stop here, because I am creating an infinite feedback at this point, so let me advance to the next example. But before I do so, I will commit the current version of this chapter to its repository. I am developing a sense of rhythm for the text through the existence of this tracing system (*git*) which asks me to define “transactions”. (period. `git commit -a`)

I will at a later point rescan this chapter’s text until this point, and rewrite certain words or sentences, perhaps extend something. I keep things separated, by using section headings. B. Latour says about the «new object» that emerges in a scientist’s laboratory—it resists description and appears to be similar to Rheinberger’s notion of epistemic thing—that it only gets «named after what it does». <sup>14</sup> But I would argue that the reference handle embodied by a name like any other artefact in a trace points both into the past and the future (cf. Sect. 3.3.3). The name gets “filled” over time, the signifier assumes its own space. I always found this particularly comprehensible in the style of writing of *Rhizome*, the first chapter of *A thousand plateaus*. <sup>15</sup> Deleuze and Guattari take an existing word, for example ‘machine’ or ‘lines’, and make more and more connections in which the machine and the lines appear, until they are actual forms from which new forms can be generated. More so as the text is reduced to a subset of fragments which appear in ever new permutations as one of the sound layers in the installation *Kalligraphie* (Sect. 4.2.1).

Very much the same is true when I introduce a symbol, for example  $\mathcal{T}_K$ , or a term, for example ‘diaphragm model’ or ‘double bind’, some associated with drawn diagrams. Although these might be linguistic or graphical constructions, they are somehow assuming a materiality as

---

<sup>14</sup>Bruno Latour (1987), ‘Laboratories’, in: *Science in action: How to follow scientists and engineers through society*, Cambridge, MA: Harvard University Press, pp. 63–100, p. 87.

<sup>15</sup>Gilles Deleuze and Félix Guattari (1987), ‘Rhizome’, in: *A thousand plateaus: Capitalism and schizophrenia*, trans. by Brian Massumi, Minneapolis: University of Minnesota Press, pp. 3–25.

soon as they are introduced by their names; they grow contours as the text proceeds—the total text, including the music pieces, the software etc. The same is true for terms introduced in the software, for example ‘Strugatzki’ is not just the tool or algorithm but a something which occupies a larger space. When I compose with concrete sounds, they might also receive such handles, ‘Schnattertier’ (quacking animal) or ‘Helicopter’,<sup>16</sup> which then peel off the sound they originally denoted to embody a very sensual or material quality, despite remaining “representant”.<sup>17</sup>

A second kind of structural reappearance is made by the rhythm of the presented canvas, the “finished” piece or text. I was listening today to the third recording from *Unvorhergesehen–Real–Farblos*. It is interesting in several ways. It begins with a recourse to the other two previous recordings—which are not “previous” but concurrent in the installation—noting the change of weather, the disappearance of workers from a particular roof, etc. It describes the outside atmosphere as perceived during the recording, then goes on to discuss an idea for the very recording which was ultimately dismissed (or not, depending on your reading, as the presentation of the dismissal of course does include the idea). Around eight or nine minutes into the spoken text, the ductus begins to be more fluid, the “theme” is established along with a number of connecting points, the permutations of which allow for this fluidity, as I am capable with this setting of improvisation to develop the different threads. Around halfway through the recording, the topics of control and losing control, embodiment, the conceptual history of process, the ideas of pure motion, rewriting and reproduction, the problematic concepts of intention and communication enter the monologue.

Listening to this text, I have the feeling that in this very moment I have reproduced this piece *as* the whole written thesis text; and vice versa. Still both are distinct, the piece is constructed in

<sup>16</sup>Not necessarily must these be nouns.

<sup>17</sup>This reminds me of a story Robin Minard told me once. He was amused by a composer—whose name I have forgotten—who titled an electroacoustic music piece *Sweet Potatoes*, because in the multitrack editor he used, the waveform for him looked just like that. I think there is more to it than amusement, it probably was an instance of such a materialisation of representational handles. In the interview with Rheinberger conducted by M. Schwab, a similar phenomenon is described, whereby in the experimental situation «the material itself somehow comes alive. It develops an agency...»: ‘Forming and Being Informed: Hans-Jörg Rheinberger in conversation with Michael Schwab’ (2013b), in: *Experimental Systems. Future Knowledge in Artistic Research*, ed. by Michael Schwab, Leuven: Leuven University Press, pp. 198–219, p. 198

a poetic way, it will be perceived in a very different setting—within an exhibition, on a couch, using the in-head localisation of my voice, in my mother tongue. While this text in front of you obeys the configuration of scientific writing with its way of structuring and referencing. The rhythm is similar, with the development of a small set of threads, here assigned to chapters 2 to 5, which are subtly interconnected, giving them “blurry edges” and fibres of cross-references. The way one adapts, the amount of time it takes until one adapts one’s reading to the kind of writing appears very similar to the way one adapts one’s hearing to the kind of speaking. Between 43’ and 49’ appears a recapitulation, a contraction of all that had been said before, very similar to this chapter of conclusion. The speaker also chooses to move to the meta perspective of talking about the piece’s own text.

The overall theme of *Unvorhergesehen...* is ‘achromaticity’ (Farblosigkeit). At one point (19’), I find that «colourlessness has to do with the fact that while foreground and background are still parted (geschieden) they are not disparate (verschieden). That is to say the vagueness (Schemenhaftigkeit) is maintained in both cases [visual and aural perception], and the background guarantees the vagueness of the foreground.» The function of the background perhaps in the written thesis is occupied by the various diagrams, whereas in the sound piece it is established by the soundscape behind my voice; it produces the possibility of an achromaticity in the foreground because on the one hand it is indifferent—the “uneventful” and steady sound of the city traffic—on the other hand it pervades the porous front, giving it the possibility of being pure and uniform sound instead of text. Later in the piece the vagueness is understood as an inclination which permits the listener (or reader) to actively isolate shades from the play between background and foreground. He retains merely the boundary between the two which themselves continue to be vague.

It is difficult for this text to provide by itself this quality of becoming pure sound and rhythm, process in the sense of the operation of its inner time.<sup>18</sup> If we were capable of reading it several times, at different speeds and magnification levels, the rhythmic texture would become more acute and corporeal. Illustrations such as the bifurcation dance of Fig. 3.8 provide some com-

---

<sup>18</sup>Cf. Hans-Jörg Rheinberger (1994), ‘Experimental Systems: Historiality, Narration, and Deconstruction’, *Science in Context* 7(1), pp. 65–81

pensation for this lack of representational forms of the text. This diagram includes a double recursion; immediately visible for someone that knows the logistic function, the procedure of evaluating this function unfolds itself from left to right, showing the emergent vascillating behaviour which opens a space, the aesthetic value of which cannot be captured by the representational form of a text equation. The second recursion lies in the fact that I *grafted* the diagram onto the text of the surrounding paragraphs, while at the same time including the notion of grafting in the diagram through the labelling of the horizon with the quotation from Derrida's *Signature Event Context*. Imagine for a moment the paper being penetrated by threads of fabric, spanning the physical distance between the point where the diagram is printed and the page on which this sentence is written. Imagine other threads, for example from the bifurcation to the point where I rediscovered *Signature Event Context* in my sketch book.

Reinforcing this first thread, the word 'excess' can be simultaneously seen as the graft in the diagram (in the graft of the diagram) and the *cruft* in this text document, as witnessed by Fig. 6.2. Derrida used the idea of excess in disputing J. L. Austin's speech act theory, in which anything that is not supporting the author's intention is accidental or gets degraded to a form of citation, such as declamations of actors on a stage which are given a context which prevents the performative aspect of their speech to be conflated with "serious" utterances. Derrida in contrast embraces all forms of "parasitism" which produces a cleft in the horizon. The word 'excess' was found in the heap of this chapter's materials because Rheinberger refers to it. In his reading it is chiefly an *intrusion*:<sup>19</sup> The contour between background and foreground which I wanted to emphasise in the diagram as well as the narrative of *Unvorhergesehen...*, it is produced by the transgression from the outside to the inside. We have seen this motion in Luhmann's actualisation of the form within the medium or in Spencer-Brown's indication of the inside of the spatial cleft. Whether I compose a piece or I write this text, the process reveals itself as the intrusion of graft/cruft, some of which I permit to remain as the material condensation ("residual"), some of which I suppress.

---

<sup>19</sup>Rheinberger, 'Experimental Systems: Historiality, Narration, and Deconstruction'.

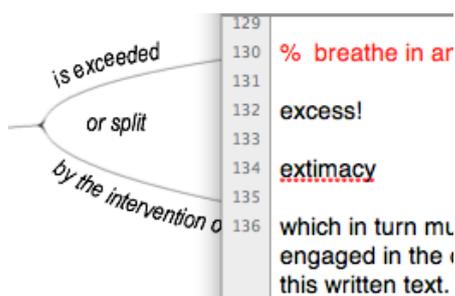


Figure 6.2: Excess of context towards the end of Chap. 4 and dumped in the notes for Chap. 6 (screenshot with thesis preview and LaTeX “source code” editor)

### 6.3 Contributions

This thesis advances the field of computer music through two major contributions; first, it deepens and shifts the knowledge of what is to be understood by the compositional process; second it shows ways in which this process can be traced and produces a novel computer music framework to support tracing in such a way that it becomes accessible not only for ex-post analysis but for the incorporation into the compositional process itself, thereby providing for new ways of aesthetic expression.

To clarify the notion of process we have first surveyed different voices. The topic has always been in the air, but never assumed more than a marginal position. We used Koenig’s essay of the same title to show that there is a double nature of the word “composition”, but that it comes with an asymmetry because composition the activity is seen as an argument of the function yielding composition the “piece”, and the description of processes in composition is reduced to a procedure to follow. Likewise, Xenakis was aware of the dedicated locus from which the composer can operate on his musical material “outside time”.

What process means has shifted over centuries, and both composers and researchers have used particular renderings of this word. A tentative definition of process clarifies that it is a coherent temporal pattern driven by an inner dynamic, and engaging with this dynamic demands what could be called an “experimental spirit”: the immersion of an experimenter into the play of material traces, generating an epistemic semiosis which is not primarily linguistic or

cognitive.<sup>20</sup> While many models and representations for “musical time” in the narrow sense have been proposed, the time in which a composition is created is denied musical status, not thoroughly analysed and not represented in computer music systems.

We introduce the notion of creation time  $\mathcal{T}_K$ , the time in which compositional decisions are made, which is either linked to performance time  $\mathcal{T}_P$  in a real-time situation or—generally—orthogonal to it, allowing for a situation which can be described as a random access model of musical data. We propose to formalise  $\mathcal{T}_K$  in the framework of atomic transactions, a notion imported from database research. Synthesising ideas from temporal databases and versioning systems, we can model the relationship between  $\mathcal{T}_K$  and  $\mathcal{T}_P$ . The embedding of transactions in a computer systems allows us to remove the agency in composition from the question of whether a human composer or an algorithm issues these transactions, an “ignorance” which will become productive in the theoretical underpinning of understanding process.

Toward this understanding, we first contribute a critique of previous attempts to define and observe it. While these attempts are useful in their respective domains, such as modelling “creativity” or “music cognition”, they are limited by their subscription, implicit or explicit, to methodological frameworks which according to our argumentation fail in two regards. These thought systems can be characterised by the axiom of control and communication, deriving from mid-20th-century psychology and cybernetics. The compositional “process” is reduced to a signification chain suspended between intention and goal attainment (or more carefully expressed, problem solving), violating the process’ property of being irreducible. The second failure is the assumption of a naturalistic enquiry which attempts to separate the researcher as an uninterested observer. As a result, there is a lacking relevance of this research for the advancement of the arts.

There is a new field timidly establishing itself which is sometimes called “artistic research” or “research in the arts”.<sup>21</sup> While I think that it has great potential in the future landscape of computer music and the digital arts, its relevance is currently rather limited, which I believe has

<sup>20</sup>Cf. Schwab, ‘Forming and Being Informed: Hans-Jörg Rheinberger in conversation with Michael Schwab’, p. 198; and Rheinberger, ‘Epistemische Dinge—Technische Dinge’, 48’

<sup>21</sup>For an overview, see Michael Biggs and Henrik Karlsson, eds. (2011), *The Routledge Companion to Research in the Arts*, Abingdon and New York: Routledge

to do with a lack of consensus as to what defines artistic research. What I propose with this thesis is explicitly not the acceptance of existing modes of artistic production as “equivalents to scientific research”. Instead I believe we need an area of artistic/scientific research which uses a new methodology where the subjects assume multiple roles between which they may switch (Fig. 3.5). In this respect, I have elaborated the idea of the re-entering of observations of the compositional process into the continuation of this very process.

To facilitate this recursive mode of engagement or “complicity” between researcher and artist—whether they are the same person or not—I investigated the interaction between forms of representation and operations on representations and the material traces which grant an indirect way of observing the unobservable (process as becoming). A point of departure which proved fruitful was the early work of Jacques Derrida, particularly *Of Grammatology* in which we find an acute concept of writing and tracing, capable of accommodating our endeavour. It might appear for some an affront to propose what goes under the unfortunate name of “post-structuralism” as a methodological replacement in an area, computer music, which is dominated by solid engineering on one side and positivist empiricism on the other. Having gone a long way to the end of this text, I am convinced that “clearing the air” was the right approach. I started from a position that disregards artistic intention (and the rediscovery thereof) altogether, I also did not look at the social system that art forms, neither did I have any musicological or psychological ambitions. When I was analysing a selection of pieces in Chap. 4, neither did I make these pieces superfluous—as happened in the empirical studies of compositional processes conducted before—nor did I dwell on an individual exegesis.

At the centre of this text is a bricolage which brings together a number of people who in some way or another have overlapping ideas which I pieced together to illuminate the nature of the “guest” (Lyotard) or “pervading space” (Spencer-Brown), the “otherness” in the apparatus which suspends composer and composition and which as inter-face continually displaces what we are doing, inscribing into the process a crucial element which establishes computer-based sound art as something that is irreducible to any compensatory forms of actions outside the computer.

As a bricolage, the text exhibits the same kind of “excess” that characterises any experimental writing process, a writing which aims at creating future.

I specifically do not advocate a single form of writing, but a multiplicity in which the elements of one layer are differentially reproduced in another. In this sense the whole thesis can be seen as an experiment of applying the concept of the experimental system as developed by Hans-Jörg Rheinberger. The methodology I proposed and implemented instead of being a text *or* an artwork *or* a software development is a texture which interweaves these three layers. If there is going to be a real collaboration between arts and sciences, having conducted this interweaving I am more convinced than ever that, instead of forming interdisciplinary teams in which everyone assumes their predefined role, the only substantial solution is to tarnish the boundaries between these roles and put into the centre of attention the translation processes which happen between these roles and their material manifestations.

Lastly, a solid and quantifiable work of computer science research and engineering is contributed. We presented a novel computer music framework which allows one to encode sound objects and their manipulations in a manner that permits their storage and retrieval from a database system. This system, which combines software transactional memory and a transparent key-value store with a model of confluent persistence, extended by quasi-retroactive modifications and a reactive event layer, contains data structures which have been for the first time implemented and used in practice as well as a number of novel concepts such as: a system façade API which allows the development of structures and algorithms independent of the choice of tracing backend; an efficient search structure for the solution of the nearest marked ancestor problem; a push-pull event bus system which can operate on structures persisted to disk; the translation of the model-view-controller concept to “aural views”, the loose coupling of realtime sound synthesis to declarative sound process objects; a model of musical objects embodied by signal processing functions, embedded in  $\mathcal{T}_P$  through dataflow time span expressions which allow both the representation on timeline editors, as shown with the graphical *Mellite* front-end, as well as their algorithmic control and update and usage in a realtime generative process; both scenarios have been demonstrated on actual composed pieces.

## 6.4 Limitations

It lies in the nature of all processes that for them to be presented as a volume, one has to interrupt them at some point and define the “product”. Not all questions could be answered, not all answers were satisfactory or exhaustive, new questions arose during the work which had to be deferred. Although I do not personally see this as a limitation, all that has been observed and interacted with did not directly involve any other composer than myself. Having both a theoretical and technological framework at hand now, it would be interesting to see in another iteration if similar observations can be made in collaboration with other composers.

The software framework developed has in its current form only been tested in fixed media composition and sound installation. In particular, my live improvisation interface *Wolkenpumpe* has not yet been ported to the new infra structure. While the theoretical underpinnings apply to live improvisation just as any other form of interaction of  $\mathcal{T}_K$  and  $\mathcal{T}_P$ , it is a limitation that the system could not yet be explored in practice in this configuration.

It should have become obvious from the discourse that this interface is all but a neutral instance. While I believe that the possibilities it plays out are not so much an expression of my personal aesthetics, but rather the workings of the system itself—the interplay of its components, the peculiarities of its application programming interface, but also its own “inertia” and perhaps clumsiness and “bugs”—it is undeniable that these possibilities may resonate better or worse with each individual composer. It is likely that the system would need to grow and be adapted when put into interaction with different composers.

While the old version of *SoundProcesses*, as it was used in *Dissemination*, had a rudimentary concept of reactions, whereby the algorithm could be triggered by sonic events happening in the signal processing, a corresponding interface in the new, confluent version is still missing. As a result, the possibilities for composing generative sound installations are greatly restricted when it comes to their traceability. Of course, any imaginable structure can still be programmed in the host language, but this will go unobserved by the tracing structure, while the ultimate vision was to model the construction of the individual algorithmic components in the object

language, something that is currently only supported in the limited form of expressions of primitive dataflow values, leaving a lot of room for improvement.

There are various questions in the framework that need to be studied better to fully comprehend them. The interaction of quasi-retroactive manipulations and the event system has not been studied. In general, the experiments conducted with the melding capability of the confluent persistence were limited to a few examples which made it into the conference papers, but which have not been studied thoroughly enough in all their practical consequences. If melding was to be employed heavily, the impact on the database performance needs to be assessed, as the path representations (finger trees) are currently written to and read from disk as complete entities. A number of questions elaborated theoretically in the random access paper<sup>22</sup> have not been answered in the form of actual implementations. For example, the distinction between genuine signals originating outside the tracing system and those from inside the system is not yet managed by the system, something that will be especially important in live improvisation and live electronic pieces.

## 6.5 Recommendations for Future Work

The limitations thus outlined can be directly read as recommendations for follow up work. In terms of the experimental configuration, I imagine that collaboration between two or even three researchers/composers could produce a shift of dynamics still coherent enough to function as a reproduction for the overall experiment here carried out and forceful enough to produce many new questions and correctives. One would observe the difference between self-pollination and cross-pollination.

The interaction of a composer/performer in real-time with the developed framework should in particular be thoroughly studied. This will on the one hand open up the question of the human computer interface in the narrow sense, and I would find the question of graphical display in this scenario most challenging. On the other hand it would lead to more complex and elaborate models for the representation of sound objects and their interconnections within the framework.

---

<sup>22</sup>Rutz, Miranda and Eckel, 'Reproducibility and Random Access in Sound Synthesis'.

In general the most interesting aspect of the framework is its ability to produce structural feedback in the sense that the musical material may react to or incorporate its own history. Since the technology developed during the thesis has finally arrived at a point where it could be considered stable enough to be “sub-routine” of future work, I imagine that a systematic and concentrated development of a sequence of pieces may illuminate the space thus created in the compositional action repertoire, a space which was only insufficiently lit by the later pieces of the thesis process.

In terms of the tracing system I would recommend any future research project to try to subject as many additional artefacts and as much data as possible to the automatic database registration. That is, all forms of documentation which happen electronically during a project should ideally be incorporated into this database, so as to synchronise the disparate elements but also to allow their reuse in the artistic work. This in turn requires that the interface for querying and managing the version history, which now is very rudimentary, should be improved. I imagine that a promising approach would be the definition of a general indexing API so that different indexes could be built on demand, subsuming the bi-temporal orderings of `BiGroup` and `BiPin` as special cases. Also: What happens if multiple composers work, perhaps concurrently, on a composition or use the system to improvise together; can we associate transactions/cursors with different users? What is the nature of distributed transactions or do we need to constantly merge multiple distributed transactions?

There are many further paths, of course, to explore from the perspective of music informatics. I have already mentioned the graphical user interfaces. In that respect the question of how interconnected dataflow expressions should be represented and edited seems an interesting one. How do we convey links and dependencies between different elements across the user interface, without resorting to a banal Pure Data type of canvas? How continuous are the transitions between a live improvisation view and a tape editing view? What is the relation between code fragments and graphical, symbolic or iconic elements? Etc.

Regarding the programming language, the overall question is how far the observation can go and if the distinction between an embedded domain specific language and the host language can

be upheld if we allow the fine-grained representation of syntax elements. What can we learn or incorporate from ‘live programming’,<sup>23</sup> a term coined to denote the development of programs in an environment which has a reactive compiler adjusting the current code representation as elements of the syntax tree change? The Scala language is also constantly evolving. Macro programming<sup>24</sup> and language virtualisation<sup>25</sup> could help alleviate the ceremony and error-proneness of writing manual serialisation and event code, something that became quite annoying during development, and keep the user facing side of the API simpler.

In terms of programming paradigms, a generalisation of the dataflow model with logical variables modelling constraint satisfaction problems (CSP)<sup>26</sup> seems an interesting direction. These types of variables are initially only known by their bounds or the domain of values they can *possibly* take on. Comparable to the way in which we construct expression chains with single valued variables, these logical variables can be composed. Furthermore, special operators establish constraints between them. For example, instead of saying that a sound object starts this much time after another sound object, something covered by the currently implemented expressions, we can just generally say that it starts *after* that sound, or we could say it starts *at most* this and this much time after that sound. CSP on the other hand always assume a perspective that goes from undetermined to determined, following the ideology of problem solving. It could be interesting to explore how this approach can be opened to randomisation, and it is a challenge to develop constraints solvers operating in an online fashion, especially if constraints may be dynamically removed from the store.

Expanding on the idea of CSP, I imagine a research project which would investigate working with classical algorithmic composition approaches, such as generative and evolutionary algorithms, cellular automata etc., in an environment that brings two crucial changes: First, translating these algorithms into a form suitable for the tracing framework, so that we can observe how one

---

<sup>23</sup>Sean McDermid (2007), ‘Living it up with a Live Programming Language’, in: *ACM SIGPLAN Notices*, vol. 42, 10, pp. 623–638.

<sup>24</sup>Eugene Burmako (2013), ‘Scala Macros: Let Our Powers Combine!’, in: *Proceedings of the 4th Annual Scala Workshop*, New York.

<sup>25</sup>Tiark Rompf and Martin Odersky (2010), ‘Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs’, in: *ACM SIGPLAN Notices*, vol. 46, 2, pp. 127–136.

<sup>26</sup>See for example Edward Tsang (1993), *Foundations of Constraint Satisfaction*, London: Academic Press

approaches these algorithms, puts them together and composes with them. Second, looking for the possible scenarios in which these algorithms work on electroacoustic music and sound art instead of the symbolic notation of instrumental music. This unification could create a broader notion of ‘signal processing’, especially when the structural feedback of the tracing system is made productive.

Looking at the theoretical foundation, there is much literature that had to remain uninspected due to its sheer volume. In particular, I regret not having had sufficient time to adequately include the work of Gilles Deleuze. I have spent some time (although not enough) on his early work *Difference and Repetition*, which closely touches upon the topic of differential reproduction. For another project which tries to tighten the different threads found in this thesis to “excess”, I believe that his work is an important if not central piece.

At last, considering the discourse on artistic research, it must be possible to bring this discourse into the communities which are less rooted in artistic practice and philosophical discourse, i.e. the academic computer music communities. The current interest in Rheinberger’s work from an artistic research perspective<sup>27</sup> is an encouraging signal that a theory-building process is on its way. What I have seen so far, for example during the Research Festival I attended at the Orpheus Institute Ghent in October 2013, has however been rather restricted to a narrow definition of artistic research as a specific type of practice carried out by musicians and performers or by visual artists in their studios. The amount of engagement with Rheinberger’s theories on experimentation often does not go far beyond a «homology between scientific and artistic activity» that he is critical of himself.<sup>28</sup> On the other hand, his own scepticism regarding a wider applicability of his theory in the arts I consider a strong point for future directions in two ways. First, the gap between “his” discourse and the artistic research discourse, also the gap between time scales—the lifetime of a scientist as experimenter versus the timeframe within which an artist realises a particular set of works—may play out its intrinsic capabilities as a “crucial suspension”, only by which the translation from one domain to the other succeeds. Second,

---

<sup>27</sup>See the recent publication Michael Schwab, ed. (2013a), *Experimental Systems. Future Knowledge in Artistic Research*, Leuven: Leuven University Press

<sup>28</sup>Schwab, ‘Forming and Being Informed: Hans-Jörg Rheinberger in conversation with Michael Schwab’, p. 206.

it emphasises that indeed we need to depart from our current notion of artistic practice and develop new forms in which art is truly grafted onto science and vice versa, producing thus «a fundamental complicity».<sup>29</sup>

---

<sup>29</sup>Rheinberger, 'Experimental Systems: Historiality, Narration, and Deconstruction'.

## Chapter 7

### Afterword

The question that never fails to confuse me is the one of my audience: How do you imagine the “recipient” of your music piece or sound installation? What type of reader do you *target* with this text? As one of the examiners remarked, the word ‘audience’ is virtually absent in this thesis. It may appear hermetic in the sense of being occupied with its own making, and thus I would like to use this opportunity to clarify who this text addresses and to invite potential readers to engage with it.

The field of study is computer music and sound art, specifically the emerging (and therefore not yet delimited) subject of artistic research within this field. I argue that by definition computer music should be transdisciplinary, bringing together computer science, engineering and signal processing, and the theory and praxis of music and sound art. To understand transdisciplinarity, we may look at the account given by Jürgen Mittelstraß:<sup>1</sup> He describes it as a domain where problems «do not do us the favour of defining themselves in the terms of a particular discipline or subfield», their solutions cannot arise from the sum of particularities. With respect to the researcher, «[c]ompetencies acquired in individual disciplines remain a fundamental precondition for tasks defined transdisciplinarily, but they no longer suffice to successfully tackle research projects which extend beyond the established fields.» The disciplines must undergo changes themselves.

Mittelstraß lists four requirements for true transdisciplinary work. First, one must have an «unconditional will to learn» outside one’s own discipline. One must then “productively immerse” in the other disciplines and, consequently, be ready to reconsider one’s previous approaches.

---

<sup>1</sup>Jürgen Mittelstraß (2011), ‘On transdisciplinarity’, *Trames: A Journal of the Humanities and Social Sciences* 15(4), pp. 329–338.

Finally, a «common text» must be produced that unifies the different disciplines. I tried to follow these steps to a certain degree. As a result, if one points the finger into the list of references, one finds such heterogeneous entries as a review of Chris Marker's film work (J. Cushing), signal processing using wavelets (I. Daubechies), philosophy (G. Deleuze), and computer science (E. Demaine et al.) in immediate vicinity. To dispel the impression of eclecticism, I want to explain the strategy behind this breadth of sources.

Let us remain with these four references. Chris Marker seems the most unconnected at first. He appears in the epigraph of the concluding chapter: «On ne sait jamais ce qu'on filme.» As unmotivated as it sounds, it brings us right to the centre of Chap. 2, the strange double nature of composition signified by the confusion between *composition the activity* and *composition the product*. The self-perception of the composer composing or the film-maker filming is detached from the perception of the audience hearing the composition or seeing the film. A process has come to an end.

But it is also a statement about knowledge—or a lack thereof. It supports the critique of the cognitive modelling approach to creativity that is central to Chap. 3. Neither is observing the compositional process about the reconstruction of an original intention, nor does it suffice to look at the conceptualisation of the composer. Instead, we must focus on the intrinsic play of the material traces produced in the process. This focus on “materiality” is not yet widely spread among the computer music research community, and we thus build on the experience of artists and philosophers. At one point, we referred to Hans-Jörg Rheinberger's acknowledgment of an *agency* of the material in an experimental system. It is precisely reflected in another statement by Marker regarding the making of *La Jetée* (1962):

«... I photographed a story I didn't completely understand. It was in the editing that the pieces of the puzzle came together, and it wasn't me who designed the puzzle. I'd have a hard time taking credit for it. It just happened, that's all.»<sup>2</sup>

<sup>2</sup>Samuel Douhaire and Annick Rivoir (2003), 'Marker Direct: an interview with Chris Marker', trans. by Antoine de Baecque, *Filmcomment* (3), pp. 38–41, URL: <http://www.filmcomment.com/article/marker-direct-an-interview-with-chris-marker> (visited on 31/07/2014).

One could go on now with *La Jetée*, perhaps noting its usage of a paradoxical re-entry, the protagonist meeting himself in the past, a situation only resolvable through death. Problems and paradoxes of time are another important aspect in the thesis which ultimately proposes a compositional approach that introduces its own history into the ongoing making of a piece. Clearly, the situation would escalate if I were to follow each of these threads to their exhaustion, and therefore I decided to produce the texture of the thesis in an open-ended way; explicit enough to understand its form, but compact enough to get the ideas across within a limited space. The texture I aimed at should be both transparent and dense at the same time. For example, going back to the quote from *Le fond de l'air est rouge*: It allowed me to talk about my activity of note taking; to anchor it in time; to anchor it to the end of one thread, Marker's death, and to the beginning of another, my birth. This is what interests me: the appearance of the compositional time within a performance. And since the thesis text should be produced in the same manner, we have here an example of such recursions.

But also the choice of the film medium is not arbitrary. Apart from an overlap with electroacoustic music with respect to methods in material treatment, it is essential to stop relying solely on a musicological discourse and music as the only art form. Having recently worked on a different research project that included several composers and theoreticians of New Music, I observed a strong tendency to exclude anything from the discourse that is not traditionally accepted as a "musical question", leading to a rather self-asserting situation. This closedness can become a problem of relevance of an already marginalised community. Far beyond my personal perception, this problem was for example highlighted in Diedrich Diedrichsen's keynote *The Re-Materialization of the Music-Object* at the 2012 ICMC.

The reader I imagine has an integral concept of the arts. She or he may be primarily a theoretician (computer music researcher, musicologist, computer scientist. . .) or a practitioner (composer, sound or media artist); ideally a mixture of both. I would like to appeal to the aesthetic sense, to a reflection on how we define materials and arrive at forms. These are questions that apply to various art forms and "genres", even if most examples are taken from the domain I work in, electroacoustic music and sound art. What this text and the accompanying pieces offer is

the elaboration of aesthetic strategies based on a materially rooted self-reflection, an attempt to overcome the antagonism between a “pure sound cosmos” and a non-material conceptualism.

The second reference in the list is ‘Retroactive Data Structures’. Back in 1976 Barry Truax posited that data structures are essential for computer music systems, because these systems necessarily embody a model of the musical activity that affects the composer.<sup>3</sup> The choice and design of data structures determines the representations, atomicity, level of detail and relations that one can work with. With the inscription into data structures, the compositional process becomes manifest and observable. However, Truax does not assume the full potential of this insight:

«... the importance of the theorist as observer needs to be discussed since the machine system itself is presumably not designed for observability, i.e. for the theorist. Ideally, the benefits should go both ways. On the one hand, theoretical insight into the effectiveness of a given system should assist in the design of new systems, and insight into the implications of a given system should assist the composer in choosing tools wisely, knowing both their limitations and potential.»<sup>4</sup>

Here, a clear distinction is drawn between theorist/designer and composer. Observation of the compositional process is not a design goal in itself, and it is reserved for the theorist. The composer can merely “choose” a tool but is not considered to be part of its design and observation. We take an opposite stance in the thesis. The relegation of the composer to a “user” of a prestructured software can be unmasked as an ideology dressed under labels such as “user-centered design” or “human-computer-interaction”.<sup>5</sup> But viewing the compositional activity as an experimental system, the engineering of representations, data structures, algorithms etc. becomes a major part of it.

By dedicating, in turn, a major part of this thesis to the review, understanding and development of data structures and algorithms that operate on them, not only do we contribute to the field

<sup>3</sup>Barry Truax (1976), ‘A Communicational Approach to Computer Sound Programs’, *Journal of Music Theory* 20(2), pp. 227–300.

<sup>4</sup>Ibid., p. 230.

<sup>5</sup>Michael Hamman (2002), ‘From Technical to Technological: The Imperative of Technology in Experimental Music Composition’, *Perspectives of New Music* 40(1), pp. 92–120.

of computer science, but we exercise what M. Hamman calls «foregrounding of representation and interaction».<sup>6</sup> In the HCI perspective, an interface should be culturally rooted so that the user does not have to think about it but can engage with it based on an implicit knowledge. The foregrounding, in contrast, means that the materials and the «epistemological frame» of the interface receive an «explicit problematisation».<sup>7</sup> I want to go even further and show an aesthetic quality of the data structures and algorithms beyond the constitution of an observation framework for computer music. This quality reveals itself both in diagrammatic representations and in their composability.

The reader I imagine is someone who is interested in the study of the agency genuinely produced by the data structures and algorithms inserted into the composition process; interested in the critical, innovatory and aesthetical role they play. The level to which she or he wants to immerse in Chap. 5 may vary. The formalisms presented in that chapter are naturally limited by my own lack of academic education in computer science. Nevertheless, I believe that readers in the field of computer science will be interested to learn how I approached the difficult task of designing a general online tracing system for the manipulation of objects. Computer musicians conversant with programming languages will have no problem to follow the argumentation in that chapter and will hopefully share the inspiration in understanding data structures as “mouldable material”. Even a reader without any training in computer science or programming languages should sense the aesthetic quality of these structures and grasp the idea of a hybrid artistic endeavour that bridges the seemingly cold “objectivity” of algorithms, their “uninterested” forms, and the materially rooted electroacoustic sound discourse. In this reading perspective, chapters 4 and 5 close ranks.

If we extend the notion of algorithmic composition beyond the generation and manipulation of symbolic scores to include the electroacoustic substrate, we must understand signal processing as a specific category of algorithms. Indeed, many algorithms can operate on a continuum of data, from sets of nominal symbols to the evenly sampled stream of numbers typical of signal

---

<sup>6</sup>Michael Hamman (1999), ‘From Symbol to Semiotic: Representation, Signification, and the Composition of Music Interaction’, *Journal of New Music Research* 28(2), pp. 90–104.

<sup>7</sup>Ibid.

processing. The same problem of the designated use of an algorithm versus an experimental foregrounding applies. In order to “engineer a breakdown” (Hamman), in order to reappropriate an algorithm for artistic exploration, one first has to understand its canonical use. But similar to the transdisciplinary requirement, this precondition must be exceeded to address the new aesthetic questions. When I referred to the wavelet transformation of I. Daubechies,<sup>8</sup> it concerned precisely the material implications, such as the way the timbre and rhythm is altered in the repeated signal decimations as well as the new form obtained from concatenating the individual decimations coming out of the so-called “pyramid” algorithm.

The reader I imagine orients her or his understanding of algorithmicity not towards a reasoning by logic, but towards a mode of experimentation embodied by an ‘interface’ between human and computer. Whether familiar with the framework of signal processing or not, she or he is interested in the transformative aspect of representations, their capability of re-writing a trace, a capability exploited in Chap. 4 to observe the motion of the compositional process across different pieces.

Questions of representation run through the entire text. The concept of ‘process’ seems irreconcilable with representationality. How can a pure form of ‘becoming’ be observed from an outside, how can it be disembedded from and re-embedded in time? An apparent solution lies in techniques of ‘modelling’, i.e. the reduction of process to a generality, a stable concept with individual processes only being instances thereof. This way, the compositional process becomes a problem-solving activity, the negotiation of and convergence towards ‘goals’. We learn something about non-material and non-temporal objects such as ‘creativity’ or the ‘mind’. But is there no alternative to this fallacy—pretending to talk about process when we really don’t? The crisis of representation is a central topic in post-structuralist philosophies that attempt to replace concepts of identity with those of difference, most evidently in the work of Deleuze (whom I refer to only peripherally) and Derrida (whom I refer more to frequently, since his idea of graphematicity appears suitable for application in art production and is well reflected in the experimental systems of Rheinberger). It is important to incorporate these philosophies for

---

<sup>8</sup>Ingrid Daubechies (1988), ‘Orthonormal Bases of Compactly Supported Wavelets’, *Communications on Pure and Applied Mathematics* 41(7), pp. 909–996.

several reasons. First, the computer music discourse is still under the hegemony of cognitive science and linguistics. While these areas are certainly important and without doubt contribute to the field, it is dangerous to assume a monopoly on the episteme of computer music and computer sound art.

The reader I imagine is both of a sceptical and of a speculative temper. I believe this to be the case of all good artists and theoreticians. Neither must we become too comfortable with what appear to be the established truths of a field, nor must we reject a thought just because it is not based on empirical “fact”. Instead we should probe that thought for its connectivity, the way it can be unfolded and the way it interacts with and perhaps rewrites other thoughts. This is how I understand Derrida’s deconstructivism and the “dice throw” in Deleuze’s method.<sup>9</sup>

Another reason lies in a new interpretation of agency in the compositional process. Instead of being tied to the human composer, motions and dynamics may now indicate a non-human agency, examples of which were already given before. Furthermore, these philosophies not only stress the irreducibility of processes, they also pave the way for a new thinking that is sometimes called “new materialism”.<sup>10</sup> Finally, we obtain ideas and methods for deparadoxifying the observation problem of processes, e.g. localisation in the “corner of one’s eye”, embracing “extimacy”, transformation of an epistemic thing into a technical object.

In short, what should become clear is that “false” representations can be powerful tools. An example is the power series: It is enabled by an operational closure such as presented to us by Luhmann or Ashby. In studying it using Spencer-Brown’s *Laws of Form*, we come to see how it can illustrate the consumption of time of a process as well as the illusion of an origin. The re-entry occurs in the making of pieces in the form of general transformations that can operate recursively on their own results. In the design of the tracing system, it occurs in the F-bounded quantification that surprisingly ties the type system together. . .

---

<sup>9</sup>James Williams (2003), *Gilles Deleuze’s Difference and Repetition: a Critical Introduction and Guide*, Edinburgh: Edinburgh University Press, pp. 13–22.

<sup>10</sup>In this text it is mainly represented through Rheinberger’s work, but further study could look at other proponents such as Karen Barad.

The reader I wish for is not necessarily an expert in every single aspect explored in this thesis, but rather someone who is interested in observing how these underlying concepts—transformative representations and the re-entry being two examples—keep appearing within the various contexts, thereby incrementally obtaining their shapes.

## **Appendix A**

### **Contents of the DVD**

The DVD accompanying the thesis contains sound excerpts from the pieces discussed in Chap. 4, and photographic material where appropriate (sound installations). The purpose is on the one hand to give evidence of the practical works realised for this thesis, and on the other hand to complement the textual discussion and enhance the understanding with examples of the otherwise irreducible sound works.

Furthermore, the DVD contains the source code of the software framework developed in this thesis. As with the sound works, to be complete, it also contains software developed prior to the thesis project, but which was relevant in the discussion. All software is released as open source under GNU GPL or LGPL license terms, as indicated on the disk.

#### **Sound Works**

The following is a list of the sound works in the chronological order of their “creation”, and given with reference to the main section of the thesis which discusses them. Sound works which were composed during the thesis period (after 2009) but did not find their way into the text are not included, but are listed in Appendix C.

<b>Title</b>	<b>Year</b>	<b>Section</b>	<b>Page</b>
Residual	2002	4.4.1	130
Zeichnung	2005	4.4.2	137
Zelle 148	2006	4.2.1	115
Kalligraphie	2007	4.2	109
Command Control Communications	2007	2.3.2, 4.2.1	31, 117
Amplifikation	2009	4.2	111
Dissemination	2010	4.2	109
Inter-Play / Re-Sound	2011	4.4.3	142
Writing Machine	2011	4.4.4	146
Leere Null (2)	2012	4.4.5	150
Voice Trap	2012	4.5.1	165
Dots	2012	4.5.1	168
Unvorhergesehen–Real–Farblos	2012	4.5.1	170
(Inde)terminus	2013	4.6	171
Machinae Coelestis	2013	4.6.1	176

## Software

The software included comprises of *Sound Processes* and the components of Fig. 5.1, the graphical front-end *Mellite*, as well as older applications *FScape*, *Wolkenpumpe*, and *Kontur*.

## Appendix B

# Survey of the Scala Programming Language

The Scala programming language<sup>1</sup> was created by Martin Odersky and his team at the École Polytechnique Fédérale de Lausanne (EPFL). It is a blend of object-oriented and functional constructs and compiles to byte-code executed on the Java Virtual Machine (JVM). Its design began in 2001 with the first version released in 2003.<sup>2</sup> Version 2 was released in 2006, where the compiler became self hosting (it was written in Scala itself). Version 2.6 in 2007 introduced existential types, structural types, and lazy values. The next big step was version 2.8, released in 2010, which among other things brought an extensive and redesigned collections library (data structures) and introduced named and default method arguments. Version 2.10 is the latest as of this writing, which brought many improvements on the library level as well as streamlining of the language, the details of which are not of interest at this point.<sup>3</sup>

We assume that the reader is superficially familiar with basic concepts of programming languages—such as state, encapsulation, concurrency, polymorphism, composition and inheritance—and is otherwise referred to the overview given by P. Van Roy.<sup>4</sup> Van Roy also identifies a multitude of programming *paradigms*, and says: «One approach that works surprisingly well is the dual-paradigm language: a language that supports one paradigm for programming in the small and another for programming in the large.»<sup>5</sup> *Scala* falls into this sweet spot of

---

<sup>1</sup>Martin Odersky et al. (2006), *An Overview of the Scala Programming Language*, tech. rep. LAMP-REPORT-2006-001, École Polytechnique Fédérale de Lausanne (EPFL).

<sup>2</sup>Martin Odersky (9th June 2006), *A Brief History of Scala*, URL: <http://www.artima.com/weblogs/viewpost.jsp?thread=163733> (visited on 04/01/2013).

<sup>3</sup>The version history of Scala can be found at <http://www.scala-lang.org/node/43> (visited on 07/01/2013)

<sup>4</sup>Peter Van Roy (2009), 'Programming Paradigms for Dummies: What Every Programmer Should Know', in: *New Computational Paradigms for Computer Music*, ed. by Gérard Assayag and Andrew Gerzso, Paris/Sampzon: IRCAM/Éditions Delatour France, pp. 9–47.

<sup>5</sup>*Ibid.*, p. 11.

dual-paradigm languages, combining object-orientation and functional programming. We will compare it here with *SuperCollider*, because the latter is a language commonly used in computer music and sound synthesis. Both share many similarities, owing partly to a common inspiration in the *Smalltalk* language.

When McCartney introduced the *SuperCollider* language, he identified a number of abstractions which he considered important. *Scala* shares most of these, including named symbols, control structures such as conditional branching and iteration, single message dispatch and lexical scoping. Examples for argument passing, object-oriented and functional constructs in both languages are shown in Table B.1. *Scala* however goes way beyond these abstractions, bringing a more powerful object model and functional toolkit to the table. These will be briefly introduced so that the reader can become familiar with the language, along with the main trait that distinguishes *Scala* from *SuperCollider*: Its strong static type system.

## B.1 Basic Syntax and Types

*Scala* deliberately uses a syntax akin to *Java* in order to ease the transition from this mainstream language (and *C++* which served as blueprint for *Java*). It uses curly braces { } to group statements into blocks, and method are called via `object.method(arg1, arg2, arg3)`. It shares the same primitive data types, although they are not treated differently than any other user defined types and therefore their names are written capitalised: `Byte` (8-bit signed integer), `Short` (16-bit signed integer), `Int` (32-bit signed integer), `Long` (64-bit signed integer, a literal can be written as `1234L`), `Float` (32-bit floating point, a literal can be written as `1234.5f`), `Double` (64-bit floating point, a literal can be written as `1234.5`), `Boolean` (with values `true` and `false`), and `String` (a literal is written with double marks `"abcde"`). Arbitrary precision integer and decimal numbers are expressed as instances of `BigInt` and `BigDecimal`, while rational and complex numbers, vectors and matrices are not provided by default, but can be added through libraries. *SuperCollider*, on the other hand, only supports 32-bit integers and 64-bit floating point numbers, and provides a class for complex numbers.

A function or method definition in *Scala* is written using the `def` keyword, followed by the function's name and zero or more argument lists followed by the result type and the function's

Table B.1: Common abstractions in the *SuperCollider* language and *Scala*. Entries marked \* are not available in the *SuperCollider* interpreter.

	SuperCollider	Scala
<b>Argument passing</b>		
Named and default arg.	~quantize = { arg in, q = 1.0, tol = 0.05; ... }	def quantize(in: Double, q: Double = 1.0, tol: Double = 0.05) = ...
Variable length arg.	~quantize.(3.1415926, q: 0.1) ~postAll = { arg ... x; x.postln } ~postAll.(1, 2, 3)	quantize(3.1415926, q = 0.1) def postAll(x: Any*) = println(x) postAll(1, 2, 3)
<b>Object-orientation</b>		
Inheritance	Single* Sin0sc : UGen { ... }	Single class Sin0sc extends UGen { ... }
Everything is an object	✓ e.g. -4.abs	✓ e.g. -4.abs
Extension methods	✓ * + Number { half { ^this * 0.5 } }	Through implicit conversion implicit class extDouble(d: Double) { def half = d * 0.5 }
<b>Functional constructs</b>		
Closures		implicit class CanLoop(n: Int) { def loop(body: => Unit) = for(j <- 1 to n) body }
First-class functions	x = "hello"; 10.do { x.postln } (1..100) select: { arg i; i % 7 == 0 } f = { arg i; i % 7 == 0 } (1..100) select: f	val x = "hello"; 10.loop { println(x) } (1 to 100) filter { _ % 7 == 0 } def f(i: Int) = i % 7 == 0 (1 to 100) filter f

body of expressions. There is no clear distinction between functions and methods, stemming from the fusion of object-oriented and functional concepts in *Scala*. We prefer to use the word ‘function’ but will say ‘method’ when we refer to a function as a member of an enclosing class type.<sup>6</sup> An argument list is a pair of parentheses enclosing a number of comma separated arguments each of which is a name followed by a colon and its required type. The function’s return type follows the argument lists separated by another colon. The function body then follows after an equals sign =. The following illustrates this by defining a function to return the maximum of two integers:

```
/* This is a multi line comment.
 * The following method takes two arguments 'x' and 'y'
 * and returns the greater of the two.
 */
def max(x: Int, y: Int): Int = {
  if (x > y) x // single line comment. 'x' is greater than 'y', so return 'x'
  else y      // otherwise return 'y'
}
```

*Scala* makes extensive use of type *inference*, meaning that in many cases the compiler can automatically determine the type and thus type annotations can be omitted. For the above function this holds for the return type. Also the function body here only consists of one expression (if-then-else), allowing to omit the curly braces. A shorter version is thus:<sup>7</sup>

```
def max(x: Int, y: Int) = if (x > y) x else y
```

A semicolon ; to end a statement is optional and therefore usually dropped. It is only needed when multiple statements are written on the same line. Likewise, a return keyword is not needed at the end of a block. Recall that within *SuperCollider* class methods a return must be explicitly signalled by a caret character ^result, whereas a function just as in *Scala* evaluates automatically to the last expression of its body: ~max = { arg x, y; if (x > y, x, y) }.

Numeric operators and operator precedence follows *Java* conventions, with arithmetic and logic operators +, -, \*, /, <, <= (less-than-or-equal), >, >= (greater-than-or-equal), == (equal), != (not

<sup>6</sup>Technically, a function value is an instance of class `Function`, and such a value is automatically constructed from a method member when calling a higher-order function (a function which takes another function as argument).

<sup>7</sup>Example taken from Martin Odersky, Lex Spoon and Bill Venner (2008), *Programming in Scala: a comprehensive step-by-step guide*, Mountain View, CA: Artima Inc, 62f

equal), ! (boolean NOT), ~ (bitwise NOT), & (boolean or bitwise AND), | (boolean or bitwise OR), and ^ (boolean or bitwise XOR).  $3 + 4 * 5$  evaluates to 23, whereas in *SuperCollider* evaluation proceeds strictly from left to right, yielding 35, requiring explicit parentheses  $3 + (4 * 5)$  if multiplication should take precedence.

## B.2 Scoping and Nesting

*Scala* tries to be a very “regular” language in that almost all constructs can appear in any position. For instance, functions and classes can be nested. The following example is adapted from the slides of the online course *Functional Programming Principles in Scala* by Martin Odersky:<sup>8</sup>

```
// Newton's method to calculate the square root of a number
def sqrt(x: Double): Double = {
  def improve(guess: Double) = (guess + x / guess) / 2
  def isGoodEnough(guess: Double) = math.abs(guess * guess - x) < 0.001
  def sqrtIter(guess: Double): Double =
    if (isGoodEnough(guess)) guess else sqrtIter(improve(guess))

  sqrtIter(1.0) // start recursive search
}
```

The inner functions `improve`, `isGoodEnough` and `sqrtIter` are encapsulated within the outer function `sqrt` and invisible from the outside. Equally important for controlling name space in large projects is the possibility to nest symbols within modules:

```
object a {
  object b {
    val c = 7
  }
}
println(a.b.c)           // descend into the inner objects (name spaces)
import a.b.c             // alternative: import a symbol from a name space...
println(c)               // ...so that afterward it can be directly referred to
import a.b.{c => alias}   // to avoid name clashes, symbols can be renamed
println(alias)           // ...during an import
```

Here, `object` defines a module, also known as singleton object. One can think of it as an instance of a traditional class in object-oriented programming, but apart from this one unique instance no further instances may be created. Singleton objects therefore serve also as containers for static symbols (corresponding to *Java*'s `static` modifier or class methods and fields in *SuperCollider*).

---

<sup>8</sup>Slides and video lectures available with login at <https://class.coursera.org/progfun-2012-001> (visited on 04/01/2013)

A value definition with keyword `val` is similar to a function definition with keyword `def`, but evaluates the right-hand side only once (see Sect. B.3.2).

Using a package statement, symbols can be repeatedly defined within a name space across different files. Packages are similar to modules, however do not allow to directly define functions but only classes within them:

```
package de.sciss.synth
class Buffer
```

which is shorthand for

```
package de {
  package sciss {
    package synth {
      class Buffer
    }
  }
}
```

The `Buffer` class thus defined can be referred to with a package prefix as `de.sciss.synth.Buffer`. For convenience, all symbols within a namespace can be imported at once using the wildcard character `_`:

```
import de.sciss.synth._ // import all symbols within package 'de.sciss.synth'
new Buffer // symbol 'Buffer' is now known and does not need namespace qualification
```

Visibility of symbols can be further controlled with modifiers `private`, `private[<package>]` and `protected`. In contrast, namespace in *SuperCollider* is flat, leading to awkward prefix schemes to avoid clashes, e.g. `JSC` for the `SwingOSC` extension classes or `Q` for the Qt graphical user interface toolkit.

## B.3 Functional Aspects

On the functional side, apart from support of closures, function values and higher order functions, it is worth introducing pattern matching, lazy evaluation, and type classes via implicit conversion.

### B.3.1 Pattern Matching

Pattern matching can be seen as a more powerful `switch` statement to branch according to the type or value of an object, and also as the functional alternative to object-oriented polymorphic

dispatch. The matching expression begins with a receiver followed by the keyword `match` followed by a block of case statements which are checked from top to bottom until a match is found:

```
def shapeID(name: String) = name match {
  case "step"      => 0
  case "linear"    => 1
  case "exponential" => 2
  case _          => sys.error("Shape_not_supported:_" + name)
}
```

This function returns an integer suitable for *SuperCollider*'s EnvGen unit generator, according to the name of a shape provided as a string argument. The wildcard character `_` can be used as a final "catch all" branch, similar to a `default:` in *Java*. If left out and no case matches, a runtime exception is thrown. The *Scala* compiler is capable to detect missing cases when the matched type is sealed, meaning that no further subtypes may be defined at a later point:

```
sealed trait Shape // sealed: all subtypes of Shape must be defined below
case object Step   extends Shape
case object Linear extends Shape
case object Exponential extends Shape
case object Sine   extends Shape

def shapeID(s: Shape) = s match {
  case Step      => 0
  case Linear    => 1
  case Exponential => 2
}
```

Here the compiler will warn that the case of `Sine` is not handled. Note that the subtypes of `Shape` were defined using a case modifier. This indicates that we want to use these types in pattern matching. It is particularly useful when dealing with classes instead of singleton objects, as pattern matching can now be used to destructure these classes. For example, imagine a class to define Open Sound Control messages:

```
case class Message(name: String, args: Any*)
```

`Any` is the root type in the *Scala* type hierarchy, thus arguments could be integers, floating pointing numbers, strings, etc. The asterisk in `Any*` indicates a variable length argument: zero or

more arguments may occur in this position. Now a responder could look for particular messages using pattern matching, e.g. react to trigger messages from the *SuperCollider* server:

```
def handle(m: Message): Unit =
  m match {
    case Message("/tr", nodeID: Int, trigID: Int, trigValue: Float) =>
      println("Received_trigger_from_node_" + nodeID)
    case _ => // ignore other messages
  }
```

The first case deconstructs the message. It only matches when the message name is `/tr`, and the message contains two integer arguments followed by one floating point argument. These arguments are automatically bound to symbols `nodeID`, `trigID`, and `trigValue`, so they can be used in the case body. If the responder wanted to make sure that the trigger comes from the node with identifier value `1000`, the match could be:

```
def handle(m: Message): Unit =
  m match {
    case Message("/tr", 1000, _, trigValue: Float) =>
      println("Received_trigger_value_" + trigValue)
    case _ => // ignore other messages
  }
```

A wildcard `_` can be used to emphasise that a particular message argument (the trigger identifier) is unused in the case body.

### B.3.2 Eager, Lazy and Repeated Evaluation

Like the functional programming language *ML* but unlike *Haskell*, *Scala* uses eager or strict evaluation by default. An eager value assignment happens when using the `val` statement, however adding the `lazy` modifier makes that assignment lazy. Only when the lazy symbol is used for the first time, the right-hand side is evaluated ('call-by-need'). In contrast, a function `def` always evaluates the right-hand side, whenever the function is used:

```
class Evaluation {
  val a = { println("Evaluating_a"); 1 }
  lazy val b = { println("Evaluating_b"); 2 }
  def c = { println("Evaluating_c"); 3 }
}

val e = new Evaluation
println("Calling_a:_ " + e.a)
println("Calling_b:_ " + e.b)
println("Calling_b:_ " + e.b)
println("Calling_c:_ " + e.c)
println("Calling_c:_ " + e.c)
```

The output of this program will thus be:

```
Evaluating a // immediately when the class is instantiated
Calling a: 1 // a is not evaluated again
Evaluating b // b is used for the first time, forcing the lazy value to be calculated
Calling b: 2
Calling b: 2 // b is used again, and the value is not re-evaluated
Evaluating c // a function call will always evaluate the function body
Calling c: 3
Evaluating c // ...even when repeated
Calling c: 3
```

In order to allow for deferred evaluation of function arguments, *Scala* allows call-by-name parameters, where the type is preceded by a double-right arrow  $\Rightarrow$ . The type of such an argument is essentially a parameterless function, also known as a ‘think’. The interplay of lazy values and call-by-name can be observed when using *Scala*’s collection type `Stream`—a lazy sequence—to create infinite generators. Similar to lists in Lisp, a stream is constructed from a (strict) head and a (lazy) tail element. Listing B.1 shows a simplified factory and interface for streams.

This definition contains four new elements. First, there is a seeming duplication of a type `Stream` which appears both as object `Stream` and as abstract class `Stream`. In such a constellation, the singleton object is called *companion object* of the class; as in the above case, companion objects often carry constructor methods. Here `cons` uses call-by-name for the tail argument which will only be evaluated once when the `lazy val tail` is resolved.

```

object Stream {
  def cons[A](hd: A, tl: => Stream[A]) = new Stream[A] {
    def head = hd
    lazy val tail = tl
    ...
  }
}
abstract class Stream[A] {
  def head: A
  def tail: Stream[A]
  def take(n: Int): Stream[A]
  def foreach(fun: A => Unit): Unit
}

```

*Listing B.1:* Skeleton of a purely functional lazy stream

Second, there are *type parameters* which are written in brackets [ ]. The `Stream` class has a type constructor parameter `A` which represents the type of the elements in the stream. Consequently the `head` method returns a value of the virtual type ‘`A`’. Type parameters are discussed in Sect. B.4. Third, there is type `Unit`. This is a type which only has one value, `()`, and is used for functions which do not wish to return a particular result—they will have side effects such as printing to the console. It is equivalent to *Java*’s `void`, but in contrast is a real type with a real value (albeit only one). There is a shortcut for defining a function that has a unit result, although it is considered bad style by many:

```

def fun1: Unit = { println("hello") }
def fun2 { println("hello") } // short version of ‘fun1’

```

Fourth, the type  $A \Rightarrow B$  is the *type of a function* that takes one argument of type `A` and returns a value of type `B`. The stream’s `foreach` method thus takes a function which is applied with each element in the stream. This can be used to iterate over the stream, similar to *SuperCollider*’s `do` method:

```

// SuperCollider: aStream.do { arg elem; elem.postln }
aStream.foreach { elem => println(elem) }
aStream.foreach(println) // shorter version

```

With this interface, for example the sequence of Fibonacci numbers can be defined:

```
val fib = {  
  def loop(h: Int, n: Int): Stream[Int] = Stream.cons(h, loop(n, h + n))  
  loop(0, 1)  
}
```

Because the sequence does not terminate, we need to be careful not to iterate over it directly, as this would produce an infinite loop. The `take` method truncates the stream after a given number of elements, similar to *SuperCollider*'s `keep` although being lazy. Stream like the other standard data structures in *Scala* is immutable (purely functional) which means that operations such as `take` do not alter the original stream, but instead produce a new stream which terminates after the given number of elements. Consistent provision of immutable structures is an important advantage over *SuperCollider* in which mutable structures dominate.

To print the first ten Fibonacci numbers:

```
fib.take(10).foreach(println)
```

### B.3.3 Implicits for Type Classes, Context and Extension Methods

Type classes are a mechanism for ad-hoc polymorphism, originally developed for the Haskell language.<sup>9</sup> This sort of polymorphism occurs when a function should accept heterogeneous types of arguments but requires them to respond to a common mechanism. As an example, the types should be comparable, perhaps allowing them to be ordered:

```
def max[A](x: A, y: A): A = if (x > y) x else y
```

This does not compile because the greater than operator `>` is not defined for *any type* `A`. The object-oriented solution would be to require the argument type to conform to some interface (in *Scala* called 'trait') that permits the comparison:

```
trait Ordered[A] {  
  def >(that: A): Boolean  
}  
def max[A <: Ordered[A]](x: A, y: A): A = if (x > y) x else y
```

Here `A <: Ordered[A]` means that type parameter `A` must be a subtype of (conform to) `Ordered[A]`.

We could now implement that trait say for a type `Duration`:

---

<sup>9</sup>Philip Wadler and Stephen Blott (1989), 'How to make *ad-hoc* polymorphism less *ad hoc*', in: *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Austin, TX, pp. 60–76.

```

case class Duration(seconds: Double) extends Ordered[Duration] {
  def >(that: Duration) = this.seconds > that.seconds
}

```

```

max(Duration(3), Duration(4)) // result: Duration(4)

```

There are three problems in this approach. First, we must have access to all the types that should conform to being ordered. It cannot be applied to existing types, such as the `Int` or `String`, instead we would need to subclass these types (which is impossible for sealed or final classes). Second, we might consider different orderings in different applications, for example strings could be ordered alphabetically ascending, or descending, distinguish or not distinguish between upper and lower case characters, and so forth. Third, the behaviour might not just depend on one receiver, but multiple objects:

```

def max[A, B](x: A, y: B): Either[A, B] = ???

```

Clearly, the ordered behaviour should be detached from the type it operates on. The idea of the type class is to define a behavioural interface and then add implementations of the behaviour for each required type. In *Scala*, the behavioural concept will be represented by a trait, and the implementations by ordinary classes. Type class resolution is achieved by implicit conversions.<sup>10</sup>

First the trait and the redefined maximum function:

```

trait Ordering[A] {
  def >(x: A, y: A): Boolean // is 'x' greater than 'y'?
}

def max[A](x: A, y: A)(ord: Ordering[A]): A = if(ord.>(x, y)) x else y

```

The new `max` function uses two parameter lists which is allowed in *Scala*. Usually multiple parameter lists are used to allow for partial function application and to aid the type inference, but here it is merely a visual aid and serves to prepare for the next step of the refinement.

Now it is possible to define an instance for case insensitive alphabetically ascending string ordering:

---

<sup>10</sup>For a full discussion see Bruno C. d. S. Oliveira, Adriaan Moors and Martin Odersky (2010), 'Type Classes as Objects and Implicits', in: *ACM Sigplan Notices – OOPSLA '10*, vol. 45, 10, pp. 341–360

```
object CaseInsensitiveStringOrdering extends Ordering[String] {  
  def >(x: String, y: String) = (x.toUpperCase compare y.toUpperCase) > 0  
}
```

```
max("hello", "World")(CaseInsensitiveStringOrdering) // result: "World"
```

Because it is cumbersome to explicitly pass in the ordering parameter, *Scala* introduces the concept of implicit values and implicit parameters. If the last parameter list begins with the modifier `implicit`, one does not need to explicitly state the arguments of this list. Instead, the compiler searches the current name scope for values of the required type which are also marked as `implicit`:

```
object StringOrderings {  
  implicit object CaseSensitive extends Ordering[String] {  
    def >(x: String, y: String) = (x compare y) > 0  
  }  
  implicit object CaseInsensitive extends Ordering[String] {  
    def >(x: String, y: String) = (x.toUpperCase compare y.toUpperCase) > 0  
  }  
}
```

```
def max[A](x: A, y: A)(implicit ord: Ordering[A]): A = if(ord.>(x, y)) x else y
```

To bring the desired implicit into scope:

```
import StringOrderings.CaseSensitive  
max("hello", "World") // result: "hello"
```

In practical applications, standard implicits may be pre-defined in certain places such as the companion objects of the applied types, making the import statements superfluous:<sup>11</sup>

---

<sup>11</sup>Again, for the full details of implicit scope: *ibid*.

```

trait Txn // the context type -- a currently open transaction

trait Transport {
  def play()(implicit tx: Txn): Unit
  def stop()(implicit tx: Txn): Unit
}

trait Cursor {
  def atomic[A](fun: Txn ⇒ A): A
}

class Example(c: Cursor, t: Transport) {
  c.atomic { implicit tx ⇒
    t.play()
    t.stop()
  }
}

```

*Listing B.2:* Using an implicit argument list for the transaction context

```

object Rate {
  // when requiring an implicit Ordering[Rate], the compiler
  // will look for a suitable value in the companion objects of
  // Ordering and Rate, and finds the following:
  implicit object RateOrdering extends Ordering[Rate] {
    def >(a: Rate, b: Rate) = a.id > b.id
  }
}

trait Rate { def id: Int }
object AudioRate extends Rate { def id = 2 }
object ControlRate extends Rate { def id = 1 }
object ScalarRate extends Rate { def id = 0 }

max(AudioRate, ControlRate) // result: AudioRate

```

A second application for implicits is to require a certain context for methods while freeing the user from explicitly stating it. Listing B.2 gives an example, using the transactional context of the software transactional memory discussed in Sect. 5.3.1.

Here the cursor’s `atomic` method takes a function which is invoked with a fresh transaction. If the function body is written with modifier `implicit` in front of its argument—here the transaction `tx`—it is marked so that the compiler will find it when resolving implicit arguments within the function body, which happens here when calling the methods `play` and `stop`. Having to

explicitly pass the transaction context argument to every transactional (state mutating) method would clutter the code enormously.

A third case is the use of implicits to convert between types. Given a receiver having a method with an argument of type A, and a caller invoking this method with an argument of type B which does not conform to the required argument type A. Before the compiler aborts with an error, it searches for an implicitly declared function  $B \Rightarrow A$ . For example, in *SuperCollider* UGen graph building, one could define a type for allowed UGen inputs, UGenIn. Similar to type classes, one could define the behaviour “act as a UGen input” for heterogeneous types, but this time through conversions. A floating point number could be a constant (scalar) UGen input, a string could be interpreted as a named control UGen, etc.:

```
object UGenIn {
  implicit def numberAsInput(value: Double) = Constant(value)
  implicit def stringAsInput(name: String) = NamedControl(name)
}
trait UGenIn
trait UGen extends UGenIn // a UGen can be plugged into another UGen
case class Out(bus: UGenIn, signal: UGenIn) extends UGen
case class SinOsc(freq: UGenIn) extends UGen
case class NamedControl(name: String) extends UGenIn
case class Constant(value: Double) extends UGenIn

// in the following, "bus" will be converted using 'stringAsInput',
// and 441 will be converted using 'numberAsInput':
Out("bus", SinOsc(441))
// result: Out(NamedControl(bus), SinOsc(Constant(441.0)))
```

Finally, a fourth case for implicits is to provide extension methods to existing types. This has already been shown in Table B.1. Unlike the third case, the type to which a value is converted is an ephemeral wrapper which does not appear anywhere explicitly. The wrapper is a short-lived stateless class whose methods return other types (such as the type thus extended):

```
implicit class RichDouble(val d: Double) {
  def ampdb: Double = math.log10(d) * 20
  def dbamp: Double = math.pow(10, d / 20)
}

0.25.ampdb // result: -12.04 ; equivalent to new RichDouble(0.25).ampdb
12.dbamp // result: 3.98
```

## B.4 Type System

The type system is perhaps *Scala*'s strongest and most elaborated part, and this short overview can only scratch the surface. A good outline is also given by M. Odersky and M. Zenger.<sup>12</sup> Here, the aim is again to provide sufficient information to read and understand code and follow the design decisions for our implementation.

Types have occurred in the previous text multiple times: Defining an object or a class declares a corresponding type. Types are required annotations to function arguments. Functions and classes may also take types themselves as parameters. Like value arguments these are separated by commas, but appear within square brackets [ ] right after the function or class name.

### B.4.1 Type Inference

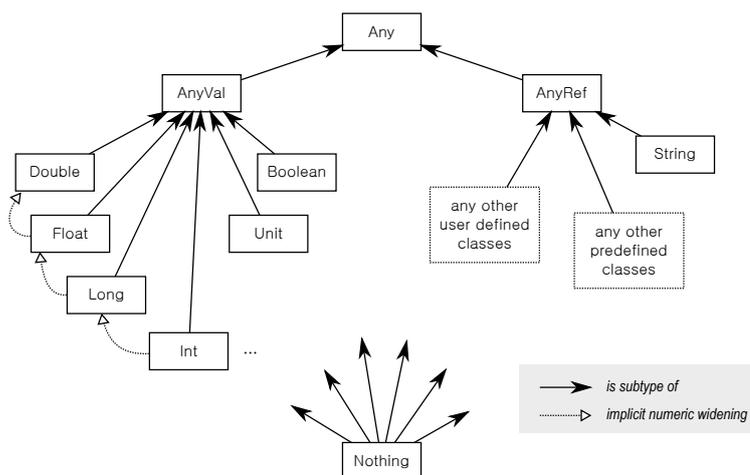
Types tend to make a statically typed language more verbose than a dynamically typed language such as *SuperCollider* which neither uses type annotations nor has the concept of type parameters. On the other hand, statically typed languages often use type *inference*, which means that type annotations can be omitted when the compiler can infer them from other knowledge. The strongest type inference is associated with the Hindley–Milner method found in *Haskell* and *ML*. Because *Scala* also deals with object-oriented sub-typing, it cannot use Hindley–Milner and has a weaker inference algorithm. Designers of libraries are still required to declare types, but as a pure user of a library, explicitly written types are often not needed, making *Scala* feel almost like a dynamically typed language and making it suitable to embed domain specific languages:

```
// note that 'List' is defined as 'List[+A]', but type parameter 'A' is inferred
// both for the constructor call and the assignment of the value
val verbose: List[Int] = List[Int](1, 2, 3)
val less: List[Int]    = List(1, 2, 3)
val lesser            = List(1, 2, 3)
```

### B.4.2 Type Hierarchy

Types are organised in a hierarchical graph specifying which type is sub- or super-type of another type. Fig. B.1 shows the periphery of this graph. At the top is *Any*, therefore any type is subtype

<sup>12</sup>Martin Odersky and Matthias Zenger (2005), 'Scalable Component Abstractions', in: *ACM SIGPLAN Notices*, vol. 40, 10, pp. 41–57.

Figure B.1: Type hierarchy of *Scala*

of `Any`. Below `Any`, there are `AnyVal` and `AnyRef`. `AnyVal` gathers the primitive types of the JVM, along with the unit type (`Unit` has been mentioned in Sect. B.3.2). These types are final and cannot be extended.<sup>13</sup> All user defined classes instead inherit from `AnyRef`, corresponding to `java.lang.Object` in *Java*. The `Null` type again is a legacy from *Java* and will not be discussed. An interesting type is the bottom type `Nothing` which is the subtype of every other type. It does not have any embodying values, but corresponds to an exception being thrown. This will be discussed further below.

Creating a subtype of a type `A` simply involves extending that type:

```

class B extends A
// or
trait B extends A

```

The difference between `class` and `trait` is that the former may take constructor arguments, such as `class B(name: String) extends A`, while the latter may participate in mixin composition. In a mixin composition a subtype extends more than one type and therefore has multiple super types:

```

class D extends A with B with C // where 'B' and 'C' must be traits

```

Such an *intersection* of types may be stated as type annotation:

<sup>13</sup>A special exception are so-called *value classes* introduced in *Scala 2.10*, a concept which is not of concern here.

```
def play(ugen: UGen with HasOutput): Unit
```

Here `play` can only be called with an argument that satisfies both types `UGen` and `HasOutput` (it must be a subtype of both). Consequently, the body of the `play` function may treat the `ugen` argument as if it was either of the two types, for example it can invoke a method defined on `UGen` or a method defined on `HasOutput`.

On the other hand, union types are currently not supported. If a function should accept disjunct types, a common super type for these must be defined and mixed into the accepted types:

```
def contains(value: A, tree: Leaf | Branch): Boolean // hypothetical, not allowed

trait LeafOrBranch
trait Leaf extends LeafOrBranch
trait Branch extends LeafOrBranch
def contains(value: A, tree: LeafOrBranch): Boolean
```

Where such common super types can not be retrofitted, method overloading or a type class approach can be used.

### B.4.3 Variance

Variance describes the treatment of type parameters under subtyping. Types can occur either in *covariant* or *contravariant* position. In a function definition, the types of the arguments are in contravariant and the function result type is in covariant position. This is easy to understand when thinking about what one can do with or expect from a type:

```
trait Sink[A] { def update(value: A): Unit }
trait Source[A] { def apply(): A }
```

To test the constraints on type parameter `A`, one can assume three related concrete types:

```
trait Top { def name: String }
trait Middle extends Top { def age: Int }
trait Bottom extends Middle { def location: String }
```

If one had a `Sink[Middle]`, its `update` method could be called with an instance of `Middle` obviously. The body of that method can assume that the argument responds *at least* to all methods on `Middle`, for example the `age` method. Then it should be allowed to pass an argument of type `Bottom` as well, since as a subtype of `Middle` it will also respond to `age`. On the other hand, passing in `Top`

cannot be legal, because it would not be possible to call `age` on it. The same goes for the return type in `Source`. If one had a `Source[Middle]` the method may legally return a more specific type such as `Bottom` but not a less specific type such as `Top`. So actual arguments and return values may always have a subtype of the specified type.

Variance comes into play when asking about the sub- or super-type relationship between `Sink[Middle]` and `Sink[Bottom]` or `Source[Middle]` and `Source[Bottom]`. If any of these occur in a function signature, which are legal subtypes? For example:

```
trait Handler[A]
  def store(value: A, sink: Sink[A]): Unit
  def retrieve(source: Source[A]): A
}
```

Assuming that we have a `Handler[Middle]` and we call the `store` method, according to the above reasoning the `value` argument could be either a `Middle` or a `Bottom`. If `Sink[Bottom]` were a subtype of `Sink[Middle]`, it could also be used as argument `sink`. It is easy to see that this cannot be the case, because if `value` was `Middle` and the `store` method's body called the `sink`'s `update` method with that value, the `sink` could assume the existence of `Bottom`'s method `location`. But on the other hand, it would be fine to pass a `Sink[Top]` as the `sink` argument, because that `sink` would only assume the existence of method name in `Top`. Therefore, `Sink[Top]` can be treated as a subtype of `Sink[Middle]` and `Sink[Bottom]`, and `Sink[Middle]` can be treated as subtype of `Sink[Bottom]`, but not the other way around. Type parameter `A` would then be treated as *contravariant* (as the outer type gets more specific, the inner type becomes less specific and vice versa).

By default, type parameters are *invariant*, so a when a `Sink[Middle]` is requested, it is not legal to use a `Sink[Top]`. To allow this, the type parameter must be marked as contravariant with a preceding minus - symbol:

```
trait Sink[-A] { def update(value: A): Unit }
```

In a similar manner, one can show that for types occurring only as method return types, such as type `A` in `Source`, these may be treated as *covariant*. In other words, a `Source[Middle]` can be treated as a subtype of `Source[Top]`, and a `Source[Bottom]` may be treated as a subtype of

Source[Middle]. To do so, the type parameter must be marked as covariant with a preceding plus + symbol:

```
trait Source[+A] { def apply(): A }
```

If both traits are combined as `trait Ref[A] extends Sink[A] with Source[A]`, both a covariant and a contravariant annotation of type parameter `A` would cause a conflict, therefore in cases where a type is used both in contravariant (method input argument type) and covariant position (method return type), it can only be invariant, so a `Ref[Middle]` would neither be a sub- or super-type of `Ref[Top]` or `Ref[Bottom]`.

A useful property of having variance is the possibility to define a single value serving the purposes of “empty structure”, “default”, or “value absent”. Functions wishing to signalise that a certain operation may or may not succeed, can wrap their return value in the `Option` type:

```
trait Option[+A]
case class Some[A](value: A) extends Option[A]
case object None extends Option[Nothing]
```

For example, *Scala*’s associative array `Map` when queried with a `get(key)`, returns an optional value. If a value associated with the key was present in the dictionary, `Some(value)` is returned, otherwise `None`:

```
// (a Map is constructed with pairs in the form of 'key -> value',
// a syntactic sugar for tuples of arity 2)
val m = Map("scalar" -> 0, "control" -> 1, "audio" -> 2)
val scalarID = m.get("scalar") // result: Some(0)
val demandID = m.get("demand") // result: None
```

Since `Option` is covariant in `A`, a single value `None` suffices to represent any inexistent value. It does not matter what type `A` is, since `Option[Nothing]` is subtype of any other `Option[A]`.

In fact, `Option[A]` has a method `get` which unwraps the value. In the case of `Some[A]`, this is the plain value of type `A`, in the case of `None` this must be “nothing”. It has been said before that `Nothing` does not have a value embodying it, instead it corresponds to an exception being thrown. Thus, `None.get` will result in an exception while still being properly typed. *Scala* uses the same exception handling system as *Java*, but does not adopt the idea of checked exceptions—methods

do not formally declare which exceptions they throw, and callers are not required to (but may) wrap methods in a `try ... catch` block.

#### B.4.4 Type Bounds

Type parameters may be constrained by an upper or lower bound. The upper bound is more common, specifying that the parameter must be equal to the upper bound or more specific (a subtype of it). The declaration of a type `A` with an upper bound is given as `A <: Upper`, and a lower bound as `A >: Lower`. An upper bound was used in one example introducing implicits in Sect. B.3.3:

```
def max[A <: Ordered[A]](x: A, y: A): A = if (x > y) x else y
```

The type parameter `A` is constrained to be a subtype of `Ordered[A]`, so that the function can use the `>` method of the arguments. An example for lower bounds is prepending to *Scala*'s `List` type, an immutable singly linked list. Prepending is done with method `::` (equivalent to `cons` in Lisp languages):

```
abstract class List[+A] {  
  def ::[B >: A](x: B): List[B]  
  ..  
}
```

The list's type parameter `A` is the least upper bound across all the contained elements. It is therefore possible to prepend an element of a less specific type `B`, resulting in a `List[B]`. Since the tail contains elements which are subtypes of `A`, they still satisfy the property of having all subtypes of `B`. For instance:

```
val m = new Middle { val name = "Niklaus"; val age = 78 }  
val t = new Top    { val name = "Martin" }  
val a = m :: Nil  // List[Middle]  
val b = t :: a    // List[Top]
```

Reminiscent of `Option`'s `None`, here `Nil` is a special value signifying the empty `List[Nothing]`. In infix notation *Scala* treats method names ending in a colon specially, such that the receiver is found on the right side and the argument on the left side. The third line is thus equivalent to `val a = Nil :: (m)`. This reversal of receiver and argument spatially reflects that `m` is prepended

to `Nil`. It is more obvious in longer lists such as `1 :: 2 :: 3 :: Nil`, whose Lisp correspondence is `(cons 1 (cons 2 (cons 3 nil)))`.

Since `Nothing` is the bottom type, `Middle` is indeed a super type of it, `[Middle >: Nothing]`. The last line prepends again an element of the less specific type `[Top >: Middle]`, resulting in a `List[Top]`.

#### B.4.5 Abstract Type Members

Similar to values which can be either passed as arguments or be defined as concrete or abstract members of a type, this duality exists for types as well. Type parameters, which have been shown above, closely correspond to *Java*'s generics. The second form in which type declarations may appear is as members of another type. At first sight, the following two definitions are equivalent:

```
// type constructor parameter
trait Ordered[A] {
  def >(that: A): Boolean
}

// abstract type member
trait Ordered {
  type A
  def >(that: A): Boolean
}
```

The second form is as powerful as the first, except lacking the possibility of variance annotation. It appears to be more verbose when requiring the proper type:<sup>14</sup>

```
def max[A1 <: Ordered { type A = A1 }](x: A1, y: A1): A1 = if (x > y) x else y
```

On the other hand, type members are useful when there are plenty of them and when the client side does not require knowledge about the applied types:<sup>15</sup>

<sup>14</sup>We use the term 'proper type' to indicate either plain parameterless types or higher kinded types where all type parameters have been applied, following Adriaan Moors, Frank Piessens and Martin Odersky (2008), 'Generics of a Higher Kind', in: *ACM SIGPLAN Notices*, vol. 43, 10, pp. 423–438

<sup>15</sup>In a survey of generic programming abstractions, the authors stress the clutter produced by type parameters in the *OCaml* language: «Associated types expressed in this manner [i.e. as type parameters] contribute to cluttered syntax because every associated type must be named in every context where the concept is used, *regardless of the associated type's relevance*» (my emphasis) Ronald Garcia et al. (2007), 'An Extended Comparative Study of Language Support for Generic Programming', *Journal of Functional Programming* 17(2), pp. 145–205; a similar argument is made later in this article regarding the capabilities of *Eiffel*, *C#* and *Java*.

```
trait Sys[ID, Tx, Var]
def apply[A, B, C](s: Sys[A, B, C]) = ... // need to carry around all parameters
def apply(s: Sys[_ , _ , _])           // ...unless wildcards are used

// versus:
trait Sys {
  type ID
  type Tx
  type Var
}
def apply(s: Sys) = ... // function does not expect actual types
```

A frequently asked question is when to use which form.<sup>16</sup> We will choose type parameters when the use site of the object carrying that type can only meaningfully deal with the object by knowing the actual parameters, or when variance is desired. This holds for data structures. Also helper structures will use type parameters if their number is small. On the other hand, we will build our main system abstraction with four principal abstract type members. There will be different types of systems (such as ephemeral versus persistent) which must be thus modular and relating to each other, and at the same time the principal types must be associated with each other. The next sections introduce the mechanisms through which this is achieved.

#### B.4.6 Path-Dependent Types and Type Projections

Reference to types which are parameters of other types happens indirectly. Type parameters basically need to be repeated in function calls such as `def max[A <: Ordered[A]](x: A, y: A): A`—here the type parameter of `Ordered` is repeated as type parameter `A` to the `max` function.<sup>17</sup> The advantage of type members is that these can be directly addressed. The first way of addressing is to use *an instance* (value) of the outer type as a receiver and a period to select the type member. The resulting type is called a *path-dependent type*, and shown in Listing B.3.

Path-dependent types constitute very strong constraints. In the example, the transaction opened from `v1.sys.open()` has the same type as the transaction required to update the sink `v1`, but since it cannot be determined that the second sink `v2` is based on the same system, its type

---

<sup>16</sup>Cf. Bill Venners (7th Oct. 2009), *Abstract Type Members versus Generic Type Parameters in Scala*, URL: <http://www.artima.com/weblogs/viewpost.jsp?thread=270195> (visited on 08/01/2013)

<sup>17</sup>See again Garcia et al., ‘An Extended Comparative Study of Language Support for Generic Programming’, §12.2f

```

trait Sys {
  type Tx
  def open(): Tx
}

trait Sink[A] {
  val sys: Sys // having a value allows to use type sys.Tx
  def update(a: A)(implicit tx: sys.Tx): Unit
}

def test(v1: Sink[Int], v2: Sink[Int]): Unit = {
  val s = v1.sys
  implicit val tx: s.Tx = s.open()
  v1() = 2 // ok. note: syntax shortcut for v1.update(2)
  v2() = 3 // does not compile -- 'v1.sys' is a different value than 'v2.sys'
}

```

*Listing B.3: Path-dependent types*

`v2.sys.Tx` is incompatible with `v1.sys.Tx`. In practise, path-dependent types can be useful for hiding implementation details, as the update of `v1` works perfectly, although the transaction type remains completely opaque. On the other hand, building modular systems becomes arduous, as a lot of bookkeeping is needed to ensure compatibility between objects, and more importantly, all participating objects must hold an instance of the system (it must be a value `val sys` to guarantee a stable type).

A more relaxed form of addressing type members is known as *type projection*. It uses the outer type (not an instance of it) as receiver and a hash character `#` to select the type member, as shown in Listing B.4.

Of course, for the sink to be capable to actually use the transaction, type member `Tx` in `Sys` must be fixed or have an upper bound. This can be done in subtyping, where type members allow refinement:

```

trait DurableTx {
  def write[A](id: Int, value:A): Unit
}
trait DurableSys extends Sys {
  type Tx <: DurableTx // refined with an upper bound
}

```

```

trait Sys {
  type Tx
  def open(): Tx
}

trait Sink[A] {
  def update(a: A)(implicit tx: Sys#Tx): Unit
}

def test(s: Sys, v1: Sink[Int], v2: Sink[Int]): Unit = {
  implicit val tx: Sys#Tx = s.open()
  v1() = 2 // ok
  v2() = 3 // ok
}

```

Listing B.4: Type projections

Abstraction	Examples
Mixin / intersection	trait A extends B with C type t = A with B with C
Variance	trait A[+B, -C]
Bounds	type t = A <: Upper type t = A >: Lower
F-bounded quantification	type A <: F[A]
Parameters vs. members	trait A[B] trait A { type B }
Path dependent type	a.B
Type projection	A#B
Self-type	trait A { this: B => }

Table B.2: Type system abstractions in *Scala*

The question then is how a group of related structures such as transactions, sinks and sources can be associated with each other to build a coherent system. What we use is a construct called F-bounded type quantification. It is discussed in Sect. 5.3.3 when actually showing our system layout.

The abstractions of the type system are summarised in Table B.2.

## B.5 Concurrency Abstractions

While Van Roy neglects type system abstractions, a substantial part of his survey reviews concurrency abstractions. This corresponds with the focus of the language *Oz* to which he has

contributed. This language is dynamically typed and based on multiple paradigms including functional, OO, and logic programming, and supports multiple concurrency abstractions. In *Oz* these are built into the language, corresponding with Van Roy’s opinion that it «is not enough that libraries have been written in the language to support the paradigm. The language’s kernel language should support the paradigm».<sup>18</sup>

*Scala* takes the opposite approach. While functional and OO constructs are the backbone of the language, it tries to be minimal with respect to other features built directly into the language. Because of its powerful and flexible syntax and type system, it allows concurrency models to be provided as pure library solutions written in ordinary *Scala*. These include low-level mechanisms such as *Java*’s fork-join framework,<sup>19</sup> and higher-level approaches such as actors, software transactional memory, futures and promises and dataflow programming. Furthermore, although *Scala* inherits the JVM’s thread model, it is possible to build for example co-routine based cooperative multitasking—as known from *SuperCollider*—with the help of a compiler plug-in for continuations.<sup>20</sup>

Traditionally, *Scala* was associated with the actors model of concurrency,<sup>21</sup> since a library based implementation *scala-actors*<sup>22</sup> was part of the standard library. Actors are lightweight processes which communicate with each other by sending immutable messages and thereby avoid mutations in shared memory. In version 2.10, *scala-actors* were deprecated in favour of another actors framework called *Akka*.<sup>23</sup> Clearly, exchanging one implementation for another would have been much more difficult if the actors paradigm had been built into the language

---

<sup>18</sup>Van Roy, ‘Programming Paradigms for Dummies: What Every Programmer Should Know’.

<sup>19</sup>Doug Lea (2000), ‘A Java Fork/Join Framework’, in: *Proceedings of the ACM 2000 conference on Java Grande*, pp. 36–43.

<sup>20</sup>Tiark Rompf, Ingo Maier and Martin Odersky (2009), ‘Implementing First-Class Polymorphic Delimited Continuations by a Type-Directed Selective CPS-Transform’, in: *ACM SIGPLAN Notices*, vol. 44, 9, pp. 317–328.

<sup>21</sup>Gul A. Agha (1985), *Actors: A Model Of Concurrent Computation In Distributed Systems*, tech. rep. AITR-844, MIT Artificial Intelligence Laboratory.

<sup>22</sup>Philipp Haller and Martin Odersky (2009), ‘Scala Actors: Unifying thread-based and event-based programming’, *Theoretical Computer Science* **410**, pp. 202–220.

<sup>23</sup>*Akka Documentation* (8th Mar. 2012), Release 2.1.0, Typesafe Inc, URL: <http://doc.akka.io/docs/akka/2.1.0/Akka.pdf> (visited on 14/01/2013).

directly. On the downside, *Scala* cannot prevent the user from actually sending mutable data across actors, so it requires the user to explicitly take care of this issue.<sup>24</sup>

Software transactional memory (STM), which is described in more detail in Sect. 5.3.1, takes a kind of opposite approach. It promotes the use of mutable cells, but supervises their access and update through transactions. Typically transactions are local to a thread, and when multiple threads access cells through their respective transactions, the system automatically detects conflicts and retries the transactions until they succeed.

Futures and promises can be seen as building blocks for a dataflow system. In general, a future is a placeholder for a value that is yet to be determined. As a placeholder it allows to work with the virtual value through function compositions. It is a lazy technique in terms of a call-by-need because the program can do meaningful concurrent things up until the moment that the future's value is actually needed. The terms 'future' and 'promise' are related and have been used slightly differently across the literature. For example B. Liskov and L. Shriram take futures as untyped values running on a single machine, whereas promises are strongly typed, may run distributed in a network, and are capable of propagating errors.<sup>25</sup> In *Scala*'s implementation, all this holds for the `Future` type, while a `Promise` simply provides a write-once access to the value of a future, needed for particular implementations.

The *Akka* framework mentioned earlier provides a futures and promises based dataflow implementation similar to the *Oz* language, shown in Listing B.5.<sup>26</sup>

## B.6 Summary

We have introduced the *Scala* programming language. Its advantages are a sophisticated static type system, broad access to existing technologies and libraries, platform independence by running on the JVM, availability of industry strength development tools, expressiveness, scalability

---

<sup>24</sup>There is ongoing research by the authors of *scala-actors* to equip the language with safety against accidental sharing of mutable state.

<sup>25</sup>Barbara Liskov and Liuba Shriram (1988), 'Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems', in: *ACM SIGPLAN Notices*, vol. 23, 7, pp. 260–267.

<sup>26</sup>The example is taken from the *Akka* online documentation at <http://doc.akka.io/docs/akka/2.2.3/scala/dataflow.html> (visited on 17/11/2013). It uses the continuations compiler plugin to allow the interruption and resuming of the control flow.

```

val v1, v2 = Promise[Int]() // promises represent dataflow variables
val f = flow {             // access and updates involving these must be wrapped in 'flow'.
    // '<<' assigns and '()' evaluates a dataflow variable.
    v1 << v2() + 10 // v1 will evaluate to the sum of v2's value + 10
    v1() + v2()    // the 'flow' block's result is the evaluated sum
}
// 'flow' returns a future. When the result is available, print it:
f onComplete println

// now (later) a value is assigned to v2, making it possible
// to complete the future 'f' with the result of 'v1() + v2()' (prints 20)
flow { v2 << 5 }

```

*Listing B.5: Dataflow programming with Akka*

and high performance. Born out of an academic context, the programming methods laboratory of the EPFL, there is inspiring and ongoing research being carried out around the language, while commercial support was recently added through the company Typesafe Inc. The language is more than ten years old now and adoption is steadily growing,<sup>27</sup> so it is reasonable to assume that it will still be relevant and supported in the next decade or two.

In comparison, *SuperCollider* is specialised for computer music. While we have shown that there are essentially no abstractions which cannot be also found in *Scala*, the latter does not (yet) have a dedicated interest group for computer music. At first, *SuperCollider* might be more accessible to composers because there is a community which can answer music relevant questions, and being dynamically typed, the cognitive load of understanding the type system is much smaller. Furthermore, the concurrency model of *SuperCollider* is simpler, because it is strictly deterministic and only supports coroutines. On the other hand, *SuperCollider* is not suited for building larger applications, because the combination of dynamic typing and lack of a proper development environment renders debugging extremely tedious, and performance critical code must be written in *C*. Every functionality that is not supported by the standard library must be written from scratch, whereas *Scala* can make instant use of the large amount of high quality *Java* libraries and an increasing amount of native *Scala* libraries.

<sup>27</sup>Language adoption and popularity has the same problems as performance benchmarks—the results are highly dependent on the metric chosen. The Wikipedia article on *Scala* lists a number of sources, according to which it is currently the second most used language on the JVM after *Java* itself.

## Appendix C

### Record of Activities

#### Publications

- › Hanns Holger Rutz (2010), ‘Rethinking the SuperCollider Client...’, in: *Proceedings of the SuperCollider Symposium*, Berlin
- › Hanns Holger Rutz, Eduardo Miranda and Gerhard Eckel (2010), ‘On the Traceability of the Compositional Process’, in: *Proceedings of the 7th Sound and Music Computing Conference (SMC)*, Barcelona, 38:1–38:7
- › Hanns Holger Rutz (2011), ‘SwingOSC’, in: *The SuperCollider Book*, ed. by Scott Wilson, David Cottle and Nick Collins, Cambridge, MA: MIT Press, pp. 305–338
- › Hanns Holger Rutz (2011), ‘Limits of Control’, in: *Proceedings of the 8th Sound and Music Computing Conference (SMC)*, Padova, 132:1–132:6
- › Hanns Holger Rutz, Eduardo Miranda and Gerhard Eckel (2011), ‘Reproducibility and Random Access in Sound Synthesis’, in: *Proceedings of the 37th International Computer Music Conference*, Huddersfield, pp. 515–522
- › Hanns Holger Rutz (2012a), ‘A Reactive, Confluently Persistent Framework for the Design of Computer Music Systems’, in: *Proceedings of the 9th Sound and Music Computing Conference (SMC)*, Copenhagen, pp. 121–129
- › Hanns Holger Rutz (2012b), ‘Composing Alongside Paradoxes’, *Explore Dream Discover: Journal of the Peninsula Contemporary Music Festival*, p. 5
- › Hanns Holger Rutz (2012c), ‘Sound Similarity as Interface between Human and Machine in Electroacoustic Composition’, in: *Proceedings of the 38th International Computer Music Conference*, Ljubljana, pp. 212–219

#### Conference Presentations

The following presentations were given in addition to the conference papers listed in the previous section:

- › Hanns Holger Rutz and Nayarí Castillo (24th Sept. 2010), *Dissemination and Temporality*, SuperCollider Symposium, Berlin
- › Hanns Holger Rutz (18th Apr. 2012), *ScalaCollider = Scala + Sound Art*, Scala Days, London

## Seminar Presentations

- › University of Plymouth, ICCMR computer music seminar with Eduardo Miranda: 02 Dec 2010 ‘Tracing the Compositional Process’; 27 Jan 2011 ‘Rethinking the SuperCollider Client’; 03 Nov 2011 ‘Writing Machine’.
- › University of Music and Performing Arts Graz, IEM doctoral candidates computer music workshop with Gerhard Eckel: 3/4 Dec 2009, 17/18 Jun 2010, 16/17 Dec 2010
- › MARE 500: 06 May 2010 work-in-progress presentation ‘Towards a fusion of different time layers in electroacoustic composition’.
- › IEM Graz computer music jour fixe: 15 Oct 2013 ‘Requirements for Computer Music Systems’.
- › University of Plymouth Festival of Research: 14 Mar 2011 ‘Computer Music Programming Tutorial’.
- › LISZT SCHOOL of Music Weimar, SeaM, seminar ‘Klangräume’ (sound spaces): 04 May 2011 spatialisation with *Meloncillo*.
- › University of Media, Arts and Design Karlsruhe, seminar ‘Raum–Klang–Bild–Bewegung’ (space–sound–image–motion): 22 May 2013 ‘Raum als Differenz- und Wiederholungsmaschine’ (space as a machine of difference and repetition).
- › University of Music and Performing Arts Graz, Institute for Music Aesthetics, seminar ‘Musik und Raum’ (music and space), 12 Dec 2012.

## Artistic Work

### Compositions

Exhibition ‘Reverberations’, Hanns Holger Rutz & Nayarí Castillo. Gallery ESC im Labor Graz, 20–30 Oct 2010:

- › *Dissemination*. Sound and room installation. Glass plates, transducers, petri dishes, plant seeds, colour gels.
- › *Feuchtigkeit*. Sound and video installation. Glass plates, transducers, video projection.

Dissemination was also exhibited at ‘{ Sounding Code }’, SuperCollider Symposium, .HBC Berlin, 18–26 Sep 2010.

Exhibition ‘Writing Machines’, Hanns Holger Rutz & Nayarí Castillo. Gallery ESC im Labor Graz, 24 Oct–17 Nov 2012:

- › *Unvorhergesehen–Real–Farblos*. Sound installation. Couch, headphones, computer prints.
- › *Dots*. Installation. Matrix print on fanfold paper.
- › *Voice Trap*. Sound and room installation. Piezo speakers, microphone, metal wire, mirrors, glass domes, text, objects.

Unvorhergesehen... and Dots were also exhibited at ‘Framed’, RONDO Graz, 3 Dec 2012 and Gallery G69 Graz, 14–20 Dec 2012.

- › *Fäden Ziehen* (2010). 6-channels tape composition 7'54". Performed: Peninsula Art Contemporary Music Festival, at Crosspoint Plymouth, 27 Feb 2010.
- › *Durchführung* (2010). Stereo tape composition 1'32", created on 18 Nov 2010 for the BBC Radio 3 Mozart Mash-up project.
- › *Inter-Play/Re-Sound* (2011). Live electronic piece for amplified piano. Performed: (a) Peninsula Contemporary Music Festival, at Crosspoint Plymouth, 11 Feb 2011. (b) ‘Making Sense of Sounds’, SCANDLE workshop, at Crosspoint Plymouth, 21 Feb 2012.
- › *Lighthouse* (2011). Stereo tape composition 3'01". Performed: Folkestone Fringe Festival, ‘Smeaton’s Tower: International Lighthouse Relay’, 26 Sep 2011.
- › *Writing Machine* (2011). Sound installation. Piezos, petri dishes, graphite powder. Exhibited: ‘SONICA’ festival, at Kino Šiška Ljubljana, 11–15 Oct 2011.
- › *Leere Null (2)* (2012). 4-channels tape composition 14'. Part 1 (2011, 5'01", stereo) released on DEGEM CD 10 ‘Replace’, curated by Marc Behrens. The quadrophonic part 2 was composed in 2012. Performed: (a) Peninsula Contemporary Music Festival, at RLB Theatre 1 Plymouth, 10 Feb 2012. (b) International Computer Music Conference, Španski Borci Ljubljana, 11 Sep 2012.
- › *Stranded Boat* (2012). Field recording, stereo 15'12". Contribution to the Electronic Music Foundation project ‘100x John: A Global Salute to John Cage’, Jun 2012.
- › *Sliding* (2013). Sound installation. Exhibited: ‘RONDO@ORF’, building of the Austrian public broadcasting service, Graz, 28 Feb–20 Mar 2013.

- › *(Inde)terminus* (2013). Electroacoustic studies, 8-channels. Presented at the ZKM Kubus Karlsruhe, 23 May 2013.
- › *Machinae Coelestis* (2013). 5-channels tape composition and star projection 16'16". Exhibition: Planetarium 'Sternenturm' Judenburg, 27–30 Aug 2013.

### Concerts

Concerts of mostly live improvisation, excluding pieces already listed in the previous section:

- › 22 Oct 2009 'Nocturne 26', Academy of Media Arts Cologne. Quartet HMSS.
- › 06 Dec 2009 Former Stasi Prison Suhl, Thuringia. Sciss, Ludger Hennig, Blazej Dowlasz.
- › 22 Jul 2010 'Freesound Concert', SMC Conference, Barcelona. Piece "Agua (Cero) 2". Sciss, Nayarí Castillo.
- › 02 Oct 2010 'Experimental Music Series', L129 Leipzig. Sciss, Ludger Hennig, Robert Rehnig.
- › 29 Nov 2010 'Café Concrete', Plymouth. Sciss, Ludger Hennig.
- › 10 Dec 2010 'Krachzehn', MEX festival, Domicil Dortmund. Quartet HMSS.
- › 23 Dec 2010 'Living Room Concert', K283 Bremen. HKM+ VB Schulze, Hammerschmidt, Sciss.
- › 07 May 2011 'Experimental Music Series', L129 Leipzig. D'Incise, Ludger Hennig, Jonas Kocher, Sciss. Released on INSUB.records net label.
- › 27 Jun 2011 'Café Concrete', Plymouth. Sciss solo.
- › 21 Dec 2011 Mariannenstr. 89 Leipzig. Ludger Hennig, Björn Lindig, Constantin Popp, Robert Rehnig, Daniel Schulz, Sciss.
- › 03 Dec 2012 'Framed', RONDO Graz. David Pirrò, Peter Venus, Marian Weger, Matthias Kronlachner, Sciss.
- › 16 Apr 2013 'Staircase', Open Cube, IEM Graz. Tape composition.

### Artist Talks

- › 'SONICA' festival, at MoTA Museum of Transitory Art Ljubljana, 12 Oct 2011.
- › RONDO Graz, 18 Oct 2012.
- › IMA Workshop Talk, ZKM Kubus Karlsruhe, 23 May 2013.

## Further Training Received

At Plymouth University:

- › Research Dialogue Workshop with Prof. Malcolm Miles, 04 May 2010.
- › Leadership and Management, Feb–May 2011.
- › *LaTeX* Introduction, 17 Mar 2011. Thanks to Martin Coath for the template of this text.
- › An Introduction to Applying for Research Funding, 29 Mar 2011.
- › The Job Interview Workshop for postgraduate researchers, 29 Mar 2011.
- › Introduction to *R*, 12 Jan 2012.
- › General Teaching Associates (GTA) Course, Jan–Mar 2012.
- › Preparing for the Viva, 28 Feb 2012.
- › Careers in Academia, 16 May 2012.
- › Writing up and Completing the Thesis, 13 Jun 2012.

The following external symposia have been attended:

- › ‘On the Choreography of Sound’, compositional practice as research, University of Music and Performing Arts Graz, MUMUTH, 7–8 Sep 2012.
- › ‘Mind the Gap’, symposium on artistic research, University of Music and Performing Arts Graz, MUMUTH, 7–8 Mar 2013.
- › ‘Paths to Artistic Identity: Artistic Experimentation, from Hades to Heaven’, ORCiM Research Festival, Orpheus Institute Ghent, 2–4 Oct 2013.

Furthermore, I completed the open online course ‘Functional Programming Principles in Scala’ by Prof. Martin Odersky, 18 Sep–08 Nov 2012, via coursera.org.



## List of References

- Acar, Umut A., Guy Blelloch and Kanat Tangwongsan (2007), *Non-oblivious Retroactive Data Structures*, tech. rep. CMU-CS-07-169, Carnegie Mellon University, School of Computer Science.
- Agha, Gul A. (1985), *Actors: A Model Of Concurrent Computation In Distributed Systems*, tech. rep. AITR-844, MIT Artificial Intelligence Laboratory.
- Akka Documentation (8th Mar. 2012), Release 2.1.0, Typesafe Inc, URL: <http://doc.akka.io/docs/akka/2.1.0/Akka.pdf> (visited on 14/01/2013).
- Allen, James F. (1983), ‘Maintaining Knowledge about Temporal Intervals’, *Communications of the ACM* **26**(11), pp. 832–843.
- (1984), ‘Towards a General Theory of Action and Time’, *Artificial Intelligence* **23**(2), pp. 123–154.
- Allombert, Antoine et al. (2006), ‘Concurrent constraints models for interactive scores’, in: *Proceedings of the 3rd Sound and Music Computing Conference (SMC)*, Marseille, 14:1–14:8.
- Allombert, Antoine et al. (2008), ‘A System of interactive scores based on qualitative and quantitative temporal constraints’, in: *Proceedings of the 4th International Conference on Digital Arts (ARTECH)*, Porto, pp. 1–8.
- Allombert, Antoine et al. (2010), ‘Virage: Designing an interactive intermedia sequencer from users requirements and theoretical background’, in: *Proceedings of the 36th International Computer Music Conference (ICMC)*, New York.
- Alstrup, Stephen, Thore Husfeldt and Theis Rauhe (1998), ‘Marked Ancestor Problems’, in: *Proceedings of the 39th Annual Symposium on Foundations of Computer Science*, IEEE, pp. 534–543.
- Aluru, Srinivas and Fatih E. Sevilgen (1999), ‘Dynamic Compressed Hyperoctrees with Application to the N-body Problem’, *Lecture Notes in Computer Science* **1738**, pp. 21–33.
- Amelunxen, Hubertus, Dieter Appelt and Peter Weibel, eds. (2008), *Notation: Kalkül und Form in den Künsten (Catalog)*, Berlin: Akademie der Künste.
- Ames, Charles (1987), ‘Automated Composition in Retrospect: 1956–1986’, *Leonardo* **20**(2), pp. 169–185.
- Anders, Torsten (2007), ‘Composing Music by Composing Rules: Design and Usage of a Generic Music Constraint System’, PhD thesis, Belfast: School of Music & Sonic Arts, Queen’s University.

- Andersen, Niels Åkerstrøm (2003), 'The Undecidability of Decision', in: *Autopoietic Organization Theory: Drawing on Niklas Luhmann's Social System Perspective*, ed. by Tore Bakken and Tor Hernes, Oslo: Abstrakt Forlag, pp. 235–258.
- Ashby, W. Ross (1956), *An introduction to Cybernetics*, London: Chapman & Hall.
- Austin, James T. and Jeffrey B. Vancouver (1996), 'Goal Constructs in Psychology: Structure, Process, and Content', *Psychological Bulletin* **120**(3), pp. 338–375.
- Bateson, Gregory et al. (1956), 'Toward a Theory of Schizophrenia', *Behavioral Science* **1**(4), pp. 251–264.
- Becker, Barbara and Gerhard Eckel (1995), *Künstlerische Imagination und Neue Medien: Zur Nutzung von Computersystemen in der Zeitgenössischen Musik*, tech. rep. Arbeitspapiere der GMD No. 960, St. Augustin: German National Research Center for Information Technology (GMD).
- (1996), 'On the Use of Computer Systems in Contemporary Music', in: *Proceedings of the 22nd International Computer Music Conference (ICMC)*, Hong Kong, pp. 118–120.
- Ben-Amram, Amir M. (1995), 'What is a "Pointer Machine"?'', *ACM SIGACT News* **26**(2), pp. 88–95.
- Bender, Michael A., Erik D. Demaine and Martin Farach-Colton (2000), 'Cache-Oblivious B-Trees', in: *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, IEEE, Redondo Beach, pp. 399–409.
- Bender, Michael A. et al. (2002), 'Two Simplified Algorithms for Maintaining Order in a List', *Lecture Notes in Computer Science* **2461**, pp. 152–164.
- Benjamin, Walter (1936/1963), *Das Kunstwerk im Zeitalter seiner technischen Reproduzierbarkeit*, Frankfurt a.M.: Suhrkamp.
- Bentley, Jon Louis (1975), 'Multidimensional Binary Search Trees Used for Associative Searching', *Communications of the ACM* **18**(9), pp. 509–517.
- Bergson, Henri (1910), *Time and Free Will, An essay on the Immediate Data of Consciousness*, trans. by Frank Lubecki Pogson, London: George Allen & Unwin.
- Berkeley DB Java Edition Architecture* (Sept. 2006), An Oracle White Paper, URL: <http://www.oracle.com/technetwork/products/berkeleydb/learnmore/bdb-je-architecture-whitepaper-366830.pdf> (visited on 09/02/2013).
- Biggs, Michael and Daniela Büchler (2011), 'Communities, Values, Conventions and Actions', in: *The Routledge Companion to Research in the Arts*, ed. by Michael Biggs and Henrik Karlsson, Abingdon and New York: Routledge, pp. 82–98.
- Biggs, Michael and Henrik Karlsson, eds. (2011), *The Routledge Companion to Research in the Arts*, Abingdon and New York: Routledge.
- Böhme, Gernot (1993), 'Atmosphere as the fundamental concept of a new aesthetics', *Thesis Eleven* **36**, pp. 113–126.

- Borgdorff, Henk (2011), 'The Production of Knowledge in Artistic Research', in: *The Routledge Companion to Research in the Arts*, ed. by Michael Biggs and Henrik Karlsson, Abingdon and New York: Routledge, pp. 44–63.
- Bresson, Jean and Carlos Agon (2006), 'Temporal control over sound synthesis processes', in: *Proceedings of the 3rd Sound and Music Computing Conference (SMC)*, Marseille, 9:1–9:10.
- (2007), 'Musical Representation of Sound in Computer-Aided Composition: A Visual Programming Framework', *Journal of New Music Research* **36**(4), pp. 251–266.
- Brodal, Gerth Stølting et al. (2012), 'Fully Persistent B-Trees', in: *Proceedings of the 23rd Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, Kyoto, pp. 602–614.
- Bronson, Nathan G., Hassan Chafi and Kunle Olukotun (2010), 'CCSTM: A library-based STM for Scala', in: *Proceedings of the First Scala Workshop*, Lausanne.
- Burmako, Eugene (2013), 'Scala Macros: Let Our Powers Combine!', in: *Proceedings of the 4th Annual Scala Workshop*, New York.
- Burns, Christopher (2002), 'Tracing Compositional Process: Software synthesis code as documentary evidence', in: *Proceedings of the 28th International Computer Music Conference (ICMC)*, Göteborg, pp. 568–571.
- Burroughs, William S. (1978), 'The Limits of Control', *Semiotext(e): Schizo-Culture* **III**(2), pp. 38–42.
- Canning, P. et al. (1989), 'F-Bounded Polymorphism for Object-Oriented Programming', in: *Proceedings of the fourth international conference on Functional programming languages and computer architecture*, ACM, pp. 273–280.
- Cardle, Marc (2004), *Automated Sound Editing*, tech. rep., Cambridge, UK: Computer Laboratory, University of Cambridge.
- Cattell, Rick (2010), 'Scalable SQL and NoSQL Data Stores', *ACM SIGMOD Record* **39**(4), pp. 12–27.
- Chadabe, Joel (1984), 'Interactive Composing: An Overview', *Computer Music Journal* **8**(1), pp. 22–27.
- Charles, Daniel (1965), 'Entr'acte: "Formal" or "Informal" Music?', *The Musical Quarterly* **51**(1), pp. 144–165.
- Checkland, Peter (2000), 'Soft Systems Methodology: A Thirty Year Retrospective', *Systems Research and Behavioral Science* **17**, S11–S58.
- Coduys, Thierry and Guillaume Ferry (2004), 'IanniX: Aesthetical/Symbolic visualisations for hypermedia composition', in: *Proceedings of the 1st Sound and Music Computing Conference (SMC)*, Paris, 18:1–18:6.
- Collins, David (2005), 'A synthesis process model of creative thinking in music composition', *Psychology of Music* **33**(2), pp. 193–216.

- Collins, David (2007), 'Real-time tracking of the creative music composition process', *Digital Creativity* **18**(4), pp. 239–256.
- Collins, Nick (2008), 'The analysis of generative music programs', *Organised Sound* **13**(3), pp. 237–248.
- (2009), *Introduction to Computer Music*, Chichester, UK: John Wiley & Sons.
- Comer, Douglas (1979), 'The Ubiquitous B-Tree', *ACM Computing Surveys (CSUR)* **11**(2), pp. 121–137.
- Copeland, George and David Maier (1984), 'Making Smalltalk a Database System', *ACM SIGMOD Record* **14**(2), pp. 316–325.
- Cormen, Thomas H., Charles E. Leiserson and Ronald L. Rivest (1990), *Introduction to Algorithms*, Cambridge, MA: The MIT Press.
- (CSAB), Computing Sciences Accreditation Board (1986), *Computer Science as a Profession*, URL: [http://web.archive.org/web/20090117183438/http://www.csab.org/comp\\_sci\\_profession.html](http://web.archive.org/web/20090117183438/http://www.csab.org/comp_sci_profession.html) (visited on 25/05/2012).
- Cushing, Jonathan (29th Mar. 2013), 'Games of a Last Chance: Chris Marker's Olympics', *Los Angeles Review of Books*, URL: <http://lareviewofbooks.org/essay/games-of-a-last-chance-chris-markers-olympics> (visited on 18/11/2013).
- Daubechies, Ingrid (1988), 'Orthonormal Bases of Compactly Supported Wavelets', *Communications on Pure and Applied Mathematics* **41**(7), pp. 909–996.
- Deleuze, Gilles (1968/1994), *Difference and Repetition*, trans. by Paul Patton, New York: Columbia University Press.
- Deleuze, Gilles and Félix Guattari (1987), 'Rhizome', in: *A thousand plateaus: Capitalism and schizophrenia*, trans. by Brian Massumi, Minneapolis: University of Minnesota Press, pp. 3–25.
- Deleuze, Gilles and Claire Parnet (1996), 'L'actuel et le virtuel', in: *Dialogues*, Paris: Flammarion, pp. 179–181.
- Demaine, Erik D., John Iacono and Stefan Langerman (2007), 'Retroactive Data Structures', *ACM Transactions on Algorithms (TALG)* **3**(2), 13:1–13:20.
- Derrida, Jacques (1981), 'The Double Session', in: *Dissemination*, trans. by Barbara Johnson, London: The Athlone Press, pp. 173–286.
- (1972/1988), 'Signature Event Context', in: *Limited Inc*, ed. by Gerald Graff, trans. by Samuel Weber, Evanston, Illinois: Northwestern University Press, pp. 1–23.
- (1967/1997), *Of grammatology*, trans. by Gayatri Chakravorty Spivak, Baltimore: Johns Hopkins University Press.
- Di Scipio, Agostino (1998), 'Compositional Models in Xenakis's Electroacoustic Music', *Perspectives of New Music* **36**(2), pp. 201–243.
- (2003), 'Sound is the interface': from interactive to ecosystemic signal processing', *Organised Sound* **8**(3), pp. 269–277.

- Dietz, Paul F. (1982), 'Maintaining order in a linked list', in: *Proceedings of the fourteenth annual ACM symposium on Theory of computing*, pp. 122–127.
- Dietz, Paul F. and Daniel D. Sleator (1987), 'Two Algorithms for Maintaining Order in a List', in: *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pp. 365–372.
- Douhaire, Samuel and Annick Rivoir (2003), 'Marker Direct: an interview with Chris Marker', trans. by Antoine de Baecque, *Filmcomment* (3), pp. 38–41, URL: <http://www.filmcomment.com/article/marker-direct-an-interview-with-chris-marker> (visited on 31/07/2014).
- Downie, Marc (2008), 'Field—a New Environment for Making Digital Art', *Computers in Entertainment (CIE)* 6(4), 54:1–54:34.
- Driscoll, James R., Daniel D. Sleator and Robert E Tarjan (1994), 'Fully Persistent Lists with Catenation', *Journal of the ACM (JACM)* 41(5), pp. 943–959.
- Driscoll, James R. et al. (1989), 'Making data structures persistent', *Journal of Computer and System Sciences* 38(1), pp. 86–124.
- Eaglestone, Barry et al. (2001), 'Composition Systems Requirements for Creativity: What Research Methodology', in: *Proceedings of Mozart Workshop on Current Research Directions in Computer Music*, Barcelona.
- Eaglestone, Barry et al. (2007), 'Information systems and creativity: an empirical study', *Journal of Documentation* 63(4), pp. 443–464.
- Eddington, Arthur S. (1929), *The Nature of the Physical World*, Cambridge: Cambridge University Press.
- Edelsbrunner, Herbert (1983), 'A New Approach to Rectangle Intersections (Part I+II)', *International Journal of Computer Mathematics* 13(3-4), pp. 209–229.
- Eigenfeldt, Arne and Philippe Pasquier (2011), 'Negotiated Content: Generative Soundscape Composition by Autonomous Musical Agents in *Coming Together: Freesound*', in: *Proceedings of the 2nd International Conference on Computational Creativity*, Mexico City, pp. 27–32.
- Elliot, Andrew J. and James W. Fryer (2008), 'The Goal Construct in Psychology', in: *Handbook of motivation science*, ed. by James Y. Shah and Wendi L. Gardner, New York: The Guilford Press, pp. 235–250.
- Elliott, Conal and Paul Hudak (1997), 'Functional reactive animation', *ACM SIGPLAN Notices* 32(8), pp. 263–273.
- Emmerson, Simon (1986), 'The Relation of Language to Materials', in: *The Language of Electroacoustic Music*, ed. by Simon Emmerson, London: Macmillan, pp. 17–39.
- (1989), 'Composing strategies and pedagogy', *Contemporary Music Review* 3(1), pp. 133–144.

- Eppstein, David, Michael T. Goodrich and Jonathan Z. Sun (2005), ‘The Skip Quadtree: A Simple Dynamic Data Structure for Multidimensional Data’, in: *Proceedings of the twenty-first annual symposium on Computational geometry*, ACM, pp. 296–305.
- (2008), ‘Skip Quadtrees: Dynamic Data Structures for Multidimensional Point Sets’, *International Journal of Computational Geometry & Applications* **18**(1 & 2), pp. 131–160.
- Feldman, Morton (1988), ‘Between categories’, *Contemporary Music Review* **2**(2), pp. 1–5.
- (1967/2004), ‘Some Elementary Questions’, in: *Give My Regards to Eighth Street: Collected Writings of Morton Feldman*, ed. by Bernard H. Friedman, Cambridge, MA: Exact Change, pp. 63–66.
- Fiat, Amos and Haim Kaplan (2003), ‘Making data structures confluent persistent’, *Journal of Algorithms* **48**(1), pp. 16–58.
- Flood, Robert L. (1988), ‘Unleashing the “Open System” Metaphor’, *Systemic Practice and Action Research* **1**(3), pp. 313–318.
- Foote, Jonathan (2000), ‘Automatic Audio Segmentation Using A Measure Of Audio Novelty’, in: *Proceedings of the IEEE International Conference on Multimedia and Expo (ICME)*, vol. 1, New York, NY, pp. 452–455.
- Foster, Caxton C. (1965), ‘Information retrieval: information storage and retrieval using AVL trees’, in: *Proceedings of the ACM 20th National Conference*, Cleveland, OH, pp. 192–205.
- Frigg, Roman and Julian Reiss (2009), ‘The Philosophy of Simulation: Hot New Issues or Same Old Stew?’, *Synthese* **169**(3), pp. 593–613.
- Frisson, Christian, Cécile Picard and Damien Tardieu (June 2010), ‘AudioGarden: towards a Usable Tool for Composite Audio Creation’, in: *Quarterly Progress Scientific Reports of the numediart research program*, vol. 3, 2, pp. 33–36.
- Garcia, Ronald et al. (2007), ‘An Extended Comparative Study of Language Support for Generic Programming’, *Journal of Functional Programming* **17**(2), pp. 145–205.
- Gasiunas, Vaidas et al. (2010), *Declarative Events for Object-Oriented Programming*, tech. rep. TUD-CS-2010-0122, Technische Universität Darmstadt.
- (2011), ‘EScala: modular event-driven object interactions in Scala’, in: *Proceedings of the tenth international conference on Aspect-oriented software development*, ACM, pp. 227–240.
- Goodman, Daniel et al. (2011), ‘MUTS: Native Scala Constructs for Software Transactional Memory’, in: *Proceedings of the Second Scala Workshop*, Stanford.
- Goodman, Daniel et al. (2013), ‘Software transactional memories for Scala’, *Journal of Parallel and Distributed Computing* **73**(2), pp. 150–163.
- Goodrich, Michael T. and Joseph A. Simons (2011), ‘Fully Retroactive Approximate Range and Nearest Neighbor Searching’, *Lecture Notes in Computer Science* **7074**, pp. 292–301.
- Gray, Jim (1981), ‘The Transaction Concept: Virtues and Limitations’, in: *Proceedings of the 7th international conference on Very Large Databases*, IEEE, Cannes, pp. 144–154.

- Guba, Egon G. and Yvonna S. Lincoln (1982/2002), 'Epistemological and methodological bases of naturalistic inquiry', in: *Evaluation Models: Viewpoints on Educational and Human Services Evaluation*, ed. by Daniel L. Stufflebeam, George F. Madaus and T. Kellaghan, Second Edition, New York: Kluwer Academic Publishers, pp. 363–381.
- Guibas, Leo J. and Robert Sedgwick (1978), 'A dichromatic framework for balanced trees', in: *19th Annual Symposium on Foundations of Computer Science (FOCS)*, IEEE, Ann Arbor, pp. 8–21.
- Guilford, Joy P. (1967), *The nature of human intelligence*, New York: McGraw-Hill.
- Guttman, Antonin (1984), 'R-trees: A dynamic index structure for spatial searching', in: *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, Boston, pp. 47–57.
- Haller, Philipp and Martin Odersky (2009), 'Scala Actors: Unifying thread-based and event-based programming', *Theoretical Computer Science* **410**, pp. 202–220.
- Hamman, Michael (1999), 'From Symbol to Semiotic: Representation, Signification, and the Composition of Music Interaction', *Journal of New Music Research* **28**(2), pp. 90–104.
- (2002), 'From Technical to Technological: The Imperative of Technology in Experimental Music Composition', *Perspectives of New Music* **40**(1), pp. 92–120.
- Harel, David (2008), 'Can Programming Be Liberated, Period?', *Computer (IEEE)* **41**(1), pp. 28–37.
- Harraway, Donna J. (1991), 'A Cyborg Manifesto: Science, Technology, and Socialist-Feminism in the Late Twentieth Century', in: *Simians, Cyborgs and Women: The Reinvention of Nature*, New York: Routledge, pp. 149–181.
- Harris, Tim et al. (2007), 'Transactional Memory: An Overview', *IEEE Micro* **27**(3), pp. 8–29.
- Heer, Jeffrey, Stuart K. Card and James A. Landay (2005), 'Prefuse: a toolkit for interactive information visualization', in: *Proceedings of the SIGCHI conference on Human factors in computing systems*, ACM, pp. 421–430.
- Heylighen, Francis (2001), 'Bootstrapping knowledge representations: From entailment meshes via semantic nets to learning webs', *Kybernetes* **30**(5/6), pp. 691–722.
- Heylighen, Francis and Cliff Joslyn (2001), 'Cybernetics and Second Order Cybernetics', in: *Encyclopedia of Physical Science and Technology*, ed. by Robert A. Meyers, vol. 4, New York: Academic Press, pp. 155–170.
- Hinze, Ralf and Ross Paterson (2006), 'Finger trees: a simple general-purpose data structure', *Journal of Functional Programming* **16**(2), pp. 197–217.
- Honing, Henkjan (1993), 'Issues on the representation of time and structure in music', *Contemporary Music Review* **9**(1), pp. 221–238.
- Jensen, Christian S. et al. (1992), 'A Glossary of Temporal Database Concepts', *ACM Special Interest Group on Management of Data (SIGMOD) Record* **21**(3), pp. 35–43.

- Jiang, Linan et al. (2000), 'The BT-Tree: A Branched and Temporal Access Method', in: *Proceedings of the 26th International Conference on Very Large Data Bases (VLDB)*, Cairo, pp. 451–460.
- Kane, Brian (2007), 'L'Objet Sonore Maintenant: Pierre Schaeffer, sound objects and the phenomenological reduction', *Organised Sound* **12**(1), pp. 15–24.
- Kaplan, Haim and Robert E. Tarjan (1996), 'Purely Functional Representations of Catenable Sorted Lists', in: *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pp. 202–211.
- Knuth, Donald E. (Mar. 1977), 'Notes on the van Emde Boas construction of priority deques: An instructive use of recursion', *Classroom notes Stanford University*.
- (1973/1998), *The Art of Computer Programming*, 2nd Edition, vol. 3, Reading, MA: Addison-Wesley.
- Koenig, Gottfried Michael (1986/1993a), 'Genesis der Form unter technischen Bedingungen', in: *Ästhetische Praxis*, vol. 3, Texte zur Musik, Saarbrücken: PFAU Verlag, pp. 277–288.
- (1978/1993b), 'Kompositionsprozesse', in: *Ästhetische Praxis*, vol. 3, Texte zur Musik, Saarbrücken: PFAU Verlag, pp. 191–210.
- Krasner, Glenn E. and Steven T. Pope (1988), 'A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80', *Journal of Object Oriented Programming* **1**(3), pp. 26–49.
- Krippendorff, Klaus (1990), 'Models and Metaphors of Communication', *Annenberg School for Communication Departmental Papers (ASC)* (276), URL: [http://repository.upenn.edu/asc\\_papers/276/](http://repository.upenn.edu/asc_papers/276/) (visited on 13/08/2014).
- (1993), 'Major Metaphors of Communication and Some Constructivist Reflections on their Use', *Cybernetics & Human Knowing* **2**(1), pp. 3–25.
- (1994), 'Der Verschwundene Bote; Metaphern und Modelle der Kommunikation', in: *Die Wirklichkeit der Medien; Eine Einführung in die Kommunikationswissenschaft*, ed. by Klaus Merten, Siegfried J. Schmidt and Siegfried Weischenberg, Opladen: Westdeutscher Verlag, pp. 79–113.
- Kruger, Daniel J. (2002), 'The Deconstruction of Constructivism', *American Psychologist* **57**(6–7), pp. 456–457.
- Kumar, Anil, Vassilis J. Tsotras and Christos Faloutsos (1998), 'Designing Access Methods for Bitemporal Databases', *IEEE Transactions on Knowledge and Data Engineering* **10**(1), pp. 1–20.
- Lacan, Jacques (1992), *The Ethics of Psychoanalysis, 1959–1960*, ed. by Marc E. Carvallo, trans. by Dennis Porter, vol. VII, The Seminar of Jacques Lacan, London: Routledge.
- Latour, Bruno (1987), 'Laboratories', in: *Science in action: How to follow scientists and engineers through society*, Cambridge, MA: Harvard University Press, pp. 63–100.

- Lea, Doug (2000), 'A Java Fork/Join Framework', in: *Proceedings of the ACM 2000 conference on Java Grande*, pp. 36–43.
- Levine, Edward M. (1971), 'Abstract Expressionism: The Mystical Experience', *Art Journal* **31**(1), pp. 22–25.
- Lewis, George E. (2000), 'Too Many Notes: Computers, Complexity and Culture in "Voyager"', *Leonardo Music Journal* **10**, pp. 33–39.
- Liljedahl, Jonatan (12th Nov. 2008), *AlgoScore user guide*, URL: <http://download.gna.org/algoscore/Help/algoscore-manual.pdf> (visited on 10/05/2012).
- Liskov, Barbara and Liuba Shrira (1988), 'Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems', in: *ACM SIGPLAN Notices*, vol. 23, 7, pp. 260–267.
- Loeb, Arthur L. (1991), 'On Behaviorism, Causality and Cybernetics', *Leonardo* **24**(3), pp. 299–302.
- Löfgren, Lars (1968), 'An axiomatic explanation of complete self-reproduction', *Bulletin of Mathematical Biophysics* **30**, pp. 415–425.
- (1992), 'Complementarity in language; toward a general understanding', in: *Nature, Cognition and System II*, ed. by Marc E. Carvallo, Dordrecht: Kluwer, pp. 113–153.
- Loy, Gareth and Curtis Abbott (1985), 'Programming Languages for Computer Music Synthesis, Performance, and Composition', *ACM Computing Surveys (CSUR)* **17**(2), pp. 235–265.
- Luhmann, Niklas (1989), 'Law as a Social System', *Northwestern University Law Review* **83**(1 & 2), pp. 136–150.
- (1993), 'Deconstruction as Second-Order Observing', *New Literary History* **24**(4), pp. 763–782.
- (1995), 'The Paradox of Observing Systems', *Cultural Critique* **31**, pp. 37–55.
- (1997), *Die Kunst der Gesellschaft*, Frankfurt a.M.: Suhrkamp Verlag.
- (1999), 'The Paradox of Form', in: *Problems of Form*, ed. by Dirk Baecker, Stanford: Stanford University Press, pp. 15–26.
- (2000), *Art as a Social System*, trans. by Eva M. Knock, Stanford: Stanford University Press.
- Liotard, Jean-François (1988/1993), 'Oikos', in: *Political writings*, trans. by Bill Readings, Minneapolis: University of Minnesota Press, pp. 96–107.
- Maier, Ingo (Nov. 2009), *The scala.swing package*, Scala Improvement Process (SID) #8, URL: <http://www.scala-lang.org/sid/8> (visited on 29/06/2013).
- Maier, Ingo and Martin Odersky (2012), 'Deprecating the Observer Pattern with Scala.React', *Technical Report EPFL-REPORT-176887*. Ecole Polytechnique Fédérale de Lausanne.
- Marsden, Alan (2000), *Representing musical time: a temporal-logic approach*, Lisse: Swets & Zeitlinger Publishers.
- McCartney, James (2002), 'Rethinking the Computer Music Language: SuperCollider', *Computer Music Journal* **26**(4), pp. 61–68.

- McDermid, Sean (2007), ‘Living it up with a Live Programming Language’, in: *ACM SIGPLAN Notices*, vol. 42, 10, pp. 623–638.
- Meyer, Leonard B. (1963), ‘The End of the Renaissance?’, *The Hudson Review* **16**(2), pp. 169–186.
- Miller, George A., Eugene Galanter and Karl H. Pribram (1960), *Plans and the Structure of Behavior*, New York: Holt, Rinehart and Winston, Inc.
- Mingers, John (1995), *Self-Producing Systems: Implications and Applications of Autopoiesis*, New York: Plenum Press.
- Minsky, Marvin L. and Otto Laske (1992), ‘A Conversation with Marvin Minsky’, *AI Magazine* **13**(3), pp. 31–45.
- Miranda, Eduardo R. (2009), ‘Lovely Algorithms, Hot Weather and Uninspiring Solfeggio’, *Contemporary Music Review* **28**(1), pp. 120–121.
- Mittelstraß, Jürgen (2011), ‘On transdisciplinarity’, *Trames: A Journal of the Humanities and Social Sciences* **15**(4), pp. 329–338.
- Moors, Adriaan, Frank Piessens and Martin Odersky (2008), ‘Generics of a Higher Kind’, in: *ACM SIGPLAN Notices*, vol. 43, 10, pp. 423–438.
- Moss, J. Eliot B. and Antony L. Hosking (1996), ‘Approaches to Adding Persistence to Java’, in: *Proceedings of the First International Workshop on Persistence and Java*, Drymen, Scotland.
- Nathan, Ran, Uriel N. Safriel and Imanuel Noy-Meir (2001), ‘Field validation and sensitivity analysis of a mechanistic model for tree seed dispersal by wind’, *Ecology* **82**(2), pp. 374–388.
- Odersky, Martin (9th June 2006), *A Brief History of Scala*, URL: <http://www.artima.com/weblogs/viewpost.jsp?thread=163733> (visited on 04/01/2013).
- Odersky, Martin, Lex Spoon and Bill Venners (2008), *Programming in Scala: a comprehensive step-by-step guide*, Mountain View, CA: Artima Inc.
- Odersky, Martin and Matthias Zenger (2005), ‘Scalable Component Abstractions’, in: *ACM SIGPLAN Notices*, vol. 40, 10, pp. 41–57.
- Odersky, Martin et al. (2006), *An Overview of the Scala Programming Language*, tech. rep. LAMP-REPORT-2006-001, École Polytechnique Fédérale de Lausanne (EPFL).
- Okasaki, Chris (1998), *Purely Functional Data Structures*, Cambridge: Cambridge University Press.
- Oliveira, Bruno C. d. S., Adriaan Moors and Martin Odersky (2010), ‘Type Classes as Objects and Implicits’, in: *ACM Sigplan Notices – OOPSLA ’10*, vol. 45, 10, pp. 341–360.
- Opyrchal, Lukasz and Atul Prakash (1999), ‘Efficient Object Serialization in Java’, in: *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems*, pp. 96–101.
- Palamidessi, Catuscia and Frank D. Valencia (2001), *A Temporal Concurrent Constraint Programming Calculus*, tech. rep. RS-01-20, BRICS Basic Research in Computer Science.

- Papadakis, Thomas (1993), ‘Skip Lists and Probabilistic Analysis of Algorithms’, PhD thesis, Waterloo, Ontario: University of Waterloo.
- Pask, Gordon (1969), ‘The meaning of cybernetics in the behavioural sciences (The cybernetics of behaviour and cognition; extending the meaning of ‘goal’)', *Progress of Cybernetics* **1**, pp. 15–44.
- Pearce, Marcus (2009), ‘To Beep or Not to Beep’, *Contemporary Music Review* **28**(1), pp. 125–126.
- Pearce, Marcus, David Meredith and Geraint Wiggins (2002), ‘Motivations and Methodologies for Automation of the Compositional Process’, *Musicae Scientiae* **6**(2), pp. 119–147.
- Pluquet, Frédéric (2012), ‘Efficient Object Versioning for Object-Oriented Languages from Model to Language Integration’, PhD thesis, Brussels: Université Libre de Bruxelles.
- Pluquet, Frédéric, Stefan Langerman and Roel Wuyts (2009), ‘Executing code in the past: efficient in-memory object graph versioning’, in: *ACM SIGPLAN Notices*, vol. 44, 10, pp. 391–408.
- Puckette, Miller (1998), ‘The Patcher’, in: *Proceedings of the 24th International Computer Music Conference (ICMC)*, Cologne, pp. 420–429.
- Pugh, William (1990), ‘Skip Lists: A Probabilistic Alternative to Balanced Trees’, *Communications of the ACM* **33**(6), pp. 668–676.
- Rescher, Nicholas (2006), *Process Philosophical Deliberations*, Heusenstamm: ontos verlag.
- Rheinberger, Hans-Jörg (1992), *Experiment–Differenz–Schrift: zur Geschichte epistemischer Dinge*, Marburg: Basiliken-Press.
- (1994), ‘Experimental Systems: Historiality, Narration, and Deconstruction’, *Science in Context* **7**(1), pp. 65–81.
- (1997), *Toward a History of Epistemic Things: Synthesizing Proteins in the Test Tube*, Palo Alto: Stanford University Press.
- (1994/2005), ‘Alles, was überhaupt zu einer Inskription führen kann’, in: *Iterationen*, Berlin: Merve Verlag, pp. 9–29.
- (5th May 2007), ‘Man weiss nicht genau, was man nicht weiss: Über die Kunst, das Unbekannte zu erforschen’, *Neue Züricher Zeitung*.
- (2nd July 2008), ‘Epistemische Dinge—Technische Dinge’, *Bochumer Kolloquium Medienwissenschaft*, URL: <http://vimeo.com/2351486> (visited on 28/08/2012).
- Rompf, Tiark, Ingo Maier and Martin Odersky (2009), ‘Implementing First-Class Polymorphic Delimited Continuations by a Type-Directed Selective CPS-Transform’, in: *ACM SIGPLAN Notices*, vol. 44, 9, pp. 317–328.
- Rompf, Tiark and Martin Odersky (2010), ‘Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs’, in: *ACM SIGPLAN Notices*, vol. 46, 2, pp. 127–136.

- Rosenberg, Daniel and Anthony Grafton (2010), *Cartographies of Time: A History of the Timeline*, New York: Princeton Architectural Press.
- Rosenblueth, Arturo, Norbert Wiener and Julian Bigelow (1943), ‘Behavior, Purpose and Teleology’, *Philosophy of Science* **10**(1), pp. 18–24.
- Röttgers, Kurt (1983), ‘Der Ursprung der Prozessidee aus dem Geiste der Chemie’, *Archiv für Begriffsgeschichte* **27**, pp. 93–157.
- Rutz, Hannis Holger (2010), ‘Rethinking the SuperCollider Client...’, in: *Proceedings of the SuperCollider Symposium*, Berlin.
- (2011), ‘Limits of Control’, in: *Proceedings of the 8th Sound and Music Computing Conference (SMC)*, Padova, 132:1–132:6.
- (2012a), ‘A Reactive, Confluently Persistent Framework for the Design of Computer Music Systems’, in: *Proceedings of the 9th Sound and Music Computing Conference (SMC)*, Copenhagen, pp. 121–129.
- (2012b), ‘Composing Alongside Paradoxes’, *Explore Dream Discover: Journal of the Peninsula Contemporary Music Festival*, p. 5.
- (2012c), ‘Sound Similarity as Interface between Human and Machine in Electroacoustic Composition’, in: *Proceedings of the 38th International Computer Music Conference*, Ljubljana, pp. 212–219.
- Rutz, Hannis Holger, Eduardo Miranda and Gerhard Eckel (2010), ‘On the Traceability of the Compositional Process’, in: *Proceedings of the 7th Sound and Music Computing Conference (SMC)*, Barcelona, 38:1–38:7.
- (2011), ‘Reproducibility and Random Access in Sound Synthesis’, in: *Proceedings of the 37th International Computer Music Conference*, Huddersfield, pp. 515–522.
- Ryan, Alex (2008), ‘What is a Systems Approach?’, *Arxiv preprint arXiv:0809.1698*.
- Salvaneschi, Guido, Gerold Hintz and Mira Mezini (2012), *REScala: Bridging Between Object-oriented and Functional Style in Reactive Applications*, tech. rep., Darmstadt: Technische Universität Darmstadt.
- Salzberg, Betty and David Lomet (1995), *Branched and Temporal Index Structures*, tech. rep. NU-CCS-95-17, Boston: Northeastern University.
- Salzberg, Betty and Vassilis J. Tsotras (1999), ‘Comparison of Access Methods for Time-Evolving Data’, *ACM Computing Surveys* **31**(2), pp. 158–221.
- Samet, Hanan (1988), ‘An overview of quadtrees, octrees, and related hierarchical data structures’, *NATO ASI Series* **F40**, pp. 51–68.
- (1995), ‘Spatial Data Structures’, in: *Modern Database Systems: The Object Model, Interoperability and Beyond*, ed. by Won Kim, New York: ACM Press and Addison-Wesley, pp. 361–385.
- Sarnak, Neil and Robert E. Tarjan (1986), ‘Planar Point Location Using Persistent Search Trees’, *Communications of the ACM* **29**(7), pp. 669–679.

- Schaeffer, Pierre (1966/1977), *Traité des objets musicaux, essai interdisciplines*, Paris: Editions du Seuil.
- Schottstaedt, Bill (1994), ‘Machine Tongues XVII: CLM: Music V Meets Common Lisp’, *Computer Music Journal* **18**(2), pp. 30–37.
- Schwab, Michael, ed. (2013a), *Experimental Systems. Future Knowledge in Artistic Research*, Leuven: Leuven University Press.
- ‘Forming and Being Informed: Hans-Jörg Rheinberger in conversation with Michael Schwab’ (2013b), in: *Experimental Systems. Future Knowledge in Artistic Research*, ed. by Michael Schwab, Leuven: Leuven University Press, pp. 198–219.
- Schwarz, Diemo (2006), ‘Concatenative sound synthesis: The early years’, *Journal of New Music Research* **35**(1), pp. 3–22.
- Shannon, Claude E. (1948), ‘A Mathematical Theory of Communication’, *Bell System Technical Journal* **27**(3), pp. 379–423.
- Sleator, Daniel D. and Robert E. Tarjan (1985), ‘Self-Adjusting Binary Search Trees’, *Journal of the ACM (JACM)* **32**(3), pp. 652–686.
- Snodgrass, Richard T. and Ilsoo Ahn (1985), ‘A Taxonomy of Time in Databases’, *ACM Special Interest Group on Management of Data (SIGMOD) Record* **14**(4), pp. 236–246.
- Spencer-Brown, George (1969/1979), *Laws of Form*, New York: E.P. Dutton.
- Taube, Heinrich (1991), ‘Common Music: A Music Composition Language in Common Lisp and CLOS’, *Computer Music Journal* **15**(2), pp. 21–32.
- Tichy, Walter F. (1982), ‘Design, Implementation, and Evaluation of a Revision Control System’, in: *Proceedings of the 6th international conference on Software engineering ICSE*, IEEE, Tokyo, pp. 58–67.
- Truax, Barry (1976), ‘A Communicational Approach to Computer Sound Programs’, *Journal of Music Theory* **20**(2), pp. 227–300.
- Tsang, Edward (1993), *Foundations of Constraint Satisfaction*, London: Academic Press.
- Van Roy, Peter (2009), ‘Programming Paradigms for Dummies: What Every Programmer Should Know’, in: *New Computational Paradigms for Computer Music*, ed. by Gérard Assayag and Andrew Gerzso, Paris/Sampzon: IRCAM/Éditions Delatour France, pp. 9–47.
- Varela, Francisco J. (1975), ‘A Calculus for Self-Reference’, *International Journal of General Systems* **2**(1), pp. 5–24.
- (1981), ‘Autonomy and Autopoiesis’, in: *Self-organizing Systems: An Interdisciplinary Approach*, ed. by Gerhard Roth and Helmut Schwegler, Frankfurt and New York: Campus Verlag, pp. 14–23.
- Venners, Bill (7th Oct. 2009), *Abstract Type Members versus Generic Type Parameters in Scala*, URL: <http://www.artima.com/weblogs/viewpost.jsp?thread=270195> (visited on 08/01/2013).

- Von Bertalanffy, Ludwig (1950), 'An outline of General System Theory', *The British Journal for the Philosophy of Science* **1**(2), pp. 134–165.
- (1972), 'The History and Status of General Systems Theory', *The Academy of Management Journal* **15**(4), pp. 407–426.
- Von Foerster, Heinz (1979), 'Cybernetics of Cybernetics', in: *Communication and Control in Society*, ed. by Klaus Krippendorff, New York: Gordon and Breach, pp. 5–8.
- (1991/2003), 'Ethics and Second-Order Cybernetics', in: *Understanding Understanding: Essays on Cybernetics and Cognition*, New York: Springer, pp. 5–8.
- Wadler, Philip and Stephen Blott (1989), 'How to make *ad-hoc* polymorphism less *ad hoc*', in: *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Austin, TX, pp. 60–76.
- Wiener, Norbert (1948), *Cybernetics, or Control and Communication in the Animal and the Machine*, New York: John Wiley & Sons.
- Williams, James (2003), *Gilles Deleuze's Difference and Repetition: a Critical Introduction and Guide*, Edinburgh: Edinburgh University Press.
- Xenakis, Iannis (1992), *Formalized Music, Thought and mathematics in composition*, Revised Edition, Stuyvesant, NY: Pendragon Press.