Luc Döbereiner

Structuring Symbolic Spaces

Programming Languages and Composition Systems in Computer Music:

Thoughts, a System, and Music

Bachelor's Thesis
Institute of Sonology
The Hague, 2008

Preface

The work presented in this text was carried out between 2006 and 2008 at the Institute of Sonology, The Hague and the Institute of Electronic Music and Acoustics, Graz. The main body of the thesis is split in three parts, a general theoretical essayistic reflection on the use of computers in composition, the description of a program I have been developing since 2006[1], and a description of three compositions of mine.

I would like to express my gratitude to Paul Berg for his knowledge, patience, supervision, understanding, and help. I would also like to thank Prof.Dr. Gerhard Eckel of the Institute of Eletronic Music and Acoustics in Graz (IEM), who helped me developing some of the fundamental ideas presented in this text during my Erasmus semester at the IEM. Furthermore, I would like to thank Graham Flett for the scrutiny, care, and perspicacity of his corrections, Raviv Ganchrow for his comments on the first part of this text, and all of my teachers and fellow students, whose comments, music, and ideas have influenced me over the last four years.

Luc Döbereiner
Transvaal, The Hague, May 2008

[1]A part of this description will be published in the upcoming *Sound and Music Computing (SMC'08) Conference Proceedings* [Döb08].

Contents

1. Introduction

Computer music is a diversified field. With the introduction of the computer into the general practice of production in most genres of music it has also become a term which is inherently vague. Therefore, it is necessary to limit the frame of the discussion for this text.

In this text, I will not give a historic survey of the development of computer music systems, as there are already numerous articles and books dealing with this subject[1]. Rather, I will try to connect with two composers, Iannis Xenakis and Gottfried Michael Koenig, whose thoughts and music have had a profound impact on my own music as well as a composition program I have developed and will present in the second part of this discussion. Moreover, I have chosen to focus on these composers because of their approach and method towards dealing with technology. Technology is fundamentally an extension of our own natural faculties, which can facilitate the discovery of new possibilities. Given that the computer is a "mental extension" requiring formal externalization of mental processes it may stipulate abstractions and generalizations of musical concepts. In the diverse and divergent works of Koenig and Xenakis, there is a point of intersection that deals with abstraction, generalization, and externalization of mental processes, which is already inherent in both composers' pre-computer work. Essentially, it is at this point of intersection where it is most interesting to consider the role of the computer, where I believe their work critically embraces technology, not from an engineer's point of view, but from the point of view of a thinking composer.

With regard to the first part of this text, I will refer to other people's opinions. However, I want to clarify that I do not intend to depict or demonstrate the conceptual edifices from which they stem and I will not give a full account of the theories of both composers. Instead, I will try to present a view of my own, while setting thoughts which influenced my work, in a light that will inevitably be shaded by my own biases.

[1]The reader may, for example, refer to [LA85], [Ame87], or [Man81].

In essence, as Goethe said, “es ist äußerst schwer, fremde Meinungen zu referieren, besonders wenn sie sich nachbarlich annähern, kreuzen und decken.”[2][Goe10]

My goal in this discussion will be to trace out some fundamental motivations for using the computer in music composition and while doing so I will at least allude towards some of their consequences. Furthermore, I will attempt to present a view of the role of the computer in composition as a tool of cognition in a an empirical process of inquiry.

Chapter 2 – Composition Processes Dealing with general issues pertaining to the use of the computer in composition processes, especially in regard to the works of Iannis Xenakis and Gottfried Michael Koenig, I argue that the the computer in the realm of composition acts as a tool of discovery facilitating a speculative process of inquiry.

Chapter 3 – Model and Music Discussing two historic works, which have been composed with the help of computer programs: Gottfried Michael Koenig’s *Übung für Klavier* and a series of pieces by Iannis Xenakis, here referred to as the *ST pieces*.

Chapter 4 – The Design of CompScheme Discussing my program *CompScheme*, its general design and primary event producing mechanism, *streams*. Several general design ideas of the system are illustrated by short examples.

Chapter 5 – The Event Model Explaining *CompScheme*’s event type model; starting from simple examples of bundling parameters together in events, moving on to defining event types and concluding with interpreting events as higher-level structural constructs. The event model is illustrated by examples of controlling the *SuperCollider* server in real-time and modeling structure 1 of Koenig’s *Übung für Klavier*.

Chapter 6 – Stochastic Synthesis in CompScheme *CompScheme*’s module for stochastic synthesis is presented as an attempt to find a generalization and possible extensions of the works of Xenakis and Koenig in this particular field of study. I will then try to show that the flexibility and expressiveness of *streams* lends itself well, not only to the description of higher-level compositional processes, but also to the lower-level sound production.

Chapter 7 – Bellavista I,II, and III Using a series of three pieces of mine, which were realized using my program *CompScheme*, for fixed-medium *Piz Palü* and *Piz Zupò*

[2]English translation: “It is most difficult to lecture on other people’s opinions, especially when they converge, cross and coincide.”

and a piece for solo piano called *Piz Bernina*, I will discuss methods of sound synthesis, the pieces' material and their structure.

Part I.

Thoughts

2. Composition Processes

Der Komponist lehrt den Computer verstehen und sprechen; was der Computer sagt, zeigt dem Komponisten, was er selber verstanden hat und aussprechen konnte. [Koe93a]

Gottfried Michael Koenig

2.1. Introduction

Computer music systems mediate between the composer's intentions and the resultant musical work. The interposition of the computer in the composition process entails a transformation of the composer's intentions. As figure 2.1 shows, the composer's task is it to first transform his or her intentions into rules and input data, which the computer system can then process. The computer then can execute these rules and offers the means for analysis and introspection. Consequently, the resulting output influences the composer's intentions once again, causing the composer to re-formulate his intentions in terms of *rules and data*, due to the fact that the computer alters the way in which representation occurs.

Using the computer in the composition of music raises fundamental questions about *musical composition* itself. Specifically towards the activity in which the computer is to be embedded, the *composition process*. However, by their very nature, these questions tend to escape attempts of being definitively answered. The term *composition process* as well as the term *musical composition* are both vague and have manifold definitions. By *Musical composition* one can mean the printed notation of a piece of music, its performance, the resulting sound, or the production of a 'score' intended to be played by acoustic instruments or realized by electronic or digital means. By *composition process* one generally understands the composer's activity to be the production of a

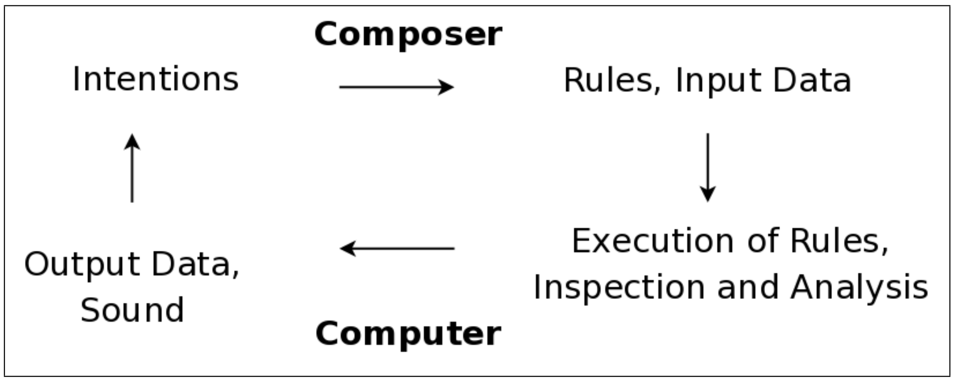


Figure 2.1.:

piece of music, ranging from preparatory work to the actual realization of the piece. The nature of this activity, however, differs from composer to composer; from piece to piece, and it is thus difficult to generalize. As composer Horacio Vaggione states, "music composition processes can be envisioned as complex systems involving a plurality of operating levels." Any closed definition of *music* or *music composition* including the seemingly all-embracing "music is everything we call music" inevitably leads to a reduction. Nevertheless, it is every piece itself and the composition of it which tries to form an answer; it unfolds its own "creation principle." [Vag01]. However, one ca discriminate between *music* and *composition*; as Herbert Brün enunciates, "music is *traces left by composition.*"[Roa85, page 4] In this text, I will adopt this difference and understand the "composition process" as an open and investigative activity that is not geared to a specific predetermined outcome, and whose principles are unfolded in the resulting music.

As Adorno says, a musical work can be seen as a "thesis": "Everything that might appear in music as being immediate and natural [. . .] is, in reality, the result of a 'thesis'; the isolated sound cannot escape this rule." (Adorno qtd. in [Vag01]) If one accepts Adorno's notion of a musical work as a "thesis", a musical *antithesis* is always imaginable, and in fact the history of music has shown that the reversal of assumed *musical universals* has often led to *musically successful* works. Therefore, prescriptive theories of musical composition generally become limitations, rather than benefits.

In this chapter, I will take a look at ideas of composition processes, methods, and the way in which computer can take part in the construction of composition models. In doing so, I will rely heavily on the theoretical works of the composers Gottfried Michael Koenig and Iannis Xenakis. Although situated in different contexts, both composers have approached the computer due to its suitability for constructing and testing models.

However, the computer itself was not the primary attraction for these composers, rather their axiomatic approaches naturally lent themselves well for programmatic implementations.

2.2. Compositional Methods

The computer requires a formal and unambiguous formulation of procedures, which leads the designer of a system or a composer working with the computer to a conscious thinking about his or her work flow and an abstract reflecting on *compositional methods*. The development of a piece of music, whatever its initial intentions, must be broken down into a sequence of describable operations. A computer music system provides the composer with means of representation and relation, with which he or she can form this sequence of operations. The system thus states a certain idea of the compositional process, by the operations, means of combination, and means of abstraction it offers. Thus, the problem arises, how an open and investigative composition process, which is not aimed at realizing predetermined results, can be modeled with means, which demand unambiguity and purposefulness. The question then becomes, how does an algorithm act as a compositional tool. According to Koenig, compositional rules can be derived in three ways: by the analysis of existing music, by introspection (i.e. analysis of the composer's own experience), or by the description of a model that is remote from analysis or introspection and instead emphasizes synthesis. I believe, that analytic models tend to stress a notion of purpose, in that they aim at modeling or deriving rules from existing music, therefore, they are contrary to my understanding of the composition process being a speculative and open-ended activity. This problem can only be solved when rules are used in a speculative way, having an axiomatic role whose consequences are then to be assessed on the basis of an audible result. As Koenig says, "the analytical task–given the music, find the rules–is reversed: given the rules, find the music."

Xenakis sees laws not as an "end in themselves" but as "tools of construction and logical lifelines" [Xen92, page 16]. Similarly, Koenig states that, "rules abstracted from music by means of analysis, introspection or model construction result primarily in the acoustic (or graphic) equivalent of this abstraction; the relation to music has to be created again." [Koe78]. For both composers the application of rules and the unfolding of compositional methods has not been an attempt to *formalize music*, but from my perspective is rather an inquiry into the question: "What is the minimum of logical constraints necessary for the construction of a musical process?" [Xen92, page 16]. This in turn leads to the question: which musical processes can be modelled by "logical constraints"? Both Koenig and Xenakis have formulated generalizations of composition methods, some of which I will discuss in this chapter.

2.2.1. Koenig's "Composition Processes"

In his article "Composition Processes", written in 1978, Gottfried Michael Koenig describes four generalizations of compositional methods. *Interpolation* is a top-down method, which "pushes forwards from the outer limits of the total form into the inner areas," in contrast to *extrapolation*, a bottom-up method, which "would proceed from the interior towards the outside." Despite their contrasting approaches both methods overlap at a central point; the connection of an outer formal shell and its inner detail: "Both methods are concentric; [...] the relation of the detail to the whole is always quite clear to the listener." A third method Koenig calls "chronological-associative" in which every element is "given its irremovable place in time" and the "composing process unfolds along the time-axis." The fourth method is a combination of methods in the "composition of blocks", each of which constitutes an independent section arranged in sequence or in parallel. Koenig states that an approach which comes closest to the "real" processes of composition is an extended form of the "chronological-associative" method, which includes feedback:

> Here the composer supplements individual data and syntactic rules describing only local strategy by objectives with which local events are continually compared. This type of method seems to approach most closely the real process of composition, but it also involves the greatest difficulties of representation in program structures.[Koe78]

The fact that Koenig deems this extended *chronological-associative* method the most difficult to model in program structures is an interesting aspect with regard to the development of computer music systems.

2.2.2. Xenakis's "Fundamental Phases of a Musical Work"

In his book *Formalized Music*, Xenakis described the "Fundamental Phases of a Musical Work", which shows a composition process; generalized by introspection, i.e. derived from observation of his own experience. It is not a discussion of different compositional methods, but rather the description of a work flow; the construction of a musical work as a sequence of operations.

1	**Initial conceptions**	intuitions, data
2	**Definition of the sonic entities**	sounds, instruments
3	**Definition of transformations**	relations, manipulations
4	**Microcomposition**	functional relations of the elements of 2
5	**Sequential programming** of 3,4	the schema and the pattern of the work
6	**Implementation of calculations**	verifications and modifications of 5
7	**Final symbolic result**	e.g. notation, graphs
8	**Sonic realization**	performance, playback

Table 2.1.: Fundamental Phases of a Musical Work (taken from [Xen92])

It is notable that Xenakis mentions intuition at the outset of the list. The mathematical and formal methods with which relations, manipulations, and schemes are constructed are then an "extension of intuition." [Xen60] Concerning the use of the computer, Xenakis says, "this list does establish ideas and allows speculation about the future. In fact, computers can take in hand phases 6. and 7. and even 8." [Xen92, page 22] And indeed, Xenakis developed a computer program, whose general outline follows the list, which is described in section 3.3. It is interesting to notice that Xenakis permits permutations in the order of the list and — as does Koenig with his idea of an extended "chronological-associative" method — includes feedback into the process. Thus, both composers point towards open systems, which enable the composer to dynamically rearrange the work flow as the processes is unfolding.

2.3. The Role of the Computer

Today, in the industrialized part the of the world, the computer is almost ubiquitous in music production and it fulfills different roles in various contexts. In the scope of this discussion, it is not necessary to investigate all of these roles, rather I have chosen to concentrate on the role that the computer has in the algorithmic composition of music. Even within this field, the computer assumes many different roles in the composition of music. Composition programs have been developed to generate entire pieces of music such as Hiller and Issacson's *Illiac Suite* [HI59] or to "solve particular structural problems" [Koe93b] such as Koenig's *Project 1*. Whatever the intention of a composition program is, the user or composer faces certain implications of the integration of the computer into the composition process. It should be notes that the general-purpose computer was not designed to be a musical instrument nor to be a tool in the composition of music. For this reason, the representation of music and the development of music composition systems raise fundamental questions about the role the computer plays in algorithmic music composition.

Despite the vagueness of the terms and the impossibility of clear definitions, *musical composition* always deals with representations and manipulations of some data on a variety of levels of representation. A look at the history of musical notation shows that the development of musical representation is directly linked to the development of musical composition itself. A representation of music is an idea of what music is composed of and consequently a manifestation of the available possibilities of composing music.

Any musical representation performs a selective abstraction; it establishes relations between things. Instead of operating with "real" objects, the representation enables an operation on signs, which abstract and generalize the multifaceted layers of a reality and let certain aspects of it emerge clearly. Operations within symbolic systems are an integral part of musical composition, and in fact are critical to any artistic creation.

2.3.1. Formalizing Music

In the last hundred years, theoretical and compositional aspects of musical thought have been influenced by the growing impact, complexity, and expressiveness of formal ways of thinking. Prior to the advent of the computer, however, there have been no formal languages in music, and the question remains as to what extent music can be formalized. The term *Formalized Music* has become a buzzword for the connection of mathematics and music, since the publication of Iannis Xenakis's book with the identical title. It is, however, a somewhat misleading term. As Xenakis has stated, mathematics is an "extension of intuition." [Xen60] Xenakis has used the term with reference to fundamental research in mathematics. His use of axiomatic methods and the construction of models in composition have not been an attempt to construct a mathematical formalization of music, but rather an effort in trying to apply the principles of building models (*Prinzipien der Modellbildung*) from mathematics in the construction of a fundamental research of music (*musikalische Grundlagenforschung*). Besides the formal nature of some musical abstractions and the fact that historically algorithmic procedures have played a role in the composition of music, the computer and the way of thinking it demands has changed the way we understand music. As composer Herbert

Brün formulates, the computer demands composers to envisage their ideas as systems:

> Composers may think of themselves and their minds and their ideas in any way they please, until they decide to use the computer as an assistant. From that moment on, composers must envisage themselves, their minds and their ideas as systems. [Brü04, page 177]

Provided a composer does not wish to work purely based on some concept of "inspiration", or based on a learned routine, such as in the reproduction of a historic style, he or she has to think consciously about the necessary testing grounds and experiments in order to realize the composer's "artistic intentions". This type of groundwork has to be described in some form, regardless if a composer believes his or her intentions are removed from "rules" and "models". Nevertheless, a composition can emanate from both relations and experiences, which can be expressed formally and those which escape this type of description. Horacio Vaggione writes:

> Abstractions of musical ideas are manifested in myriad ways and degrees, one of which is of course their suitability for implementation as algorithms, enabling musicians to explore possibilities that would otherwise lie out of reach. [Vag01]

Thus, the computer, which demands formalization, helps to reveal which musical abstractions can and which cannot be formalized. The formalization of certain aspects in a musical composition can provide the composer with a world of new possibilities. A purely formal approach, however, is limited and "formal rigor does not always result in musical coherence." [Vag01]. The goal of formalization in music can therefore only be to facilitate the discovery of previously inaccessible possibilities for the composer. Moreover, a musical result may not exhibit a rule-based or formal character, yet the work process from which it emerges is describable, for it can be discriminated between *music* and *composition*. Computer-aided algorithmic composition of music entails a formalization to some degree, simply by the unambiguity and generalization the computer demands. However, this should not be understood as an attempt to formalize music. If it can be said that formalization takes place to a certain degree then it is in composition, not in music.

2.3.2. The Computer as an *Erkenntniswerkzeug*

Given the impossibility (and futility) of a complete formalization of music, what role does the computer play in a composition process and what is gained by its use?

Every art form addresses perception and imagination, just as the objects in our environment do. Nevertheless, artworks differ from "natural objects" in that they (to put it in Hegelian terms) mirror the thinking consciousness by an externalization (*Entäußerung*) of its content. That is, art as an external object presents itself to the mind as a manifestation of mental activities. By a reflection on our consciousness in the object and a reflection on the object itself it creates an elevation of consciousness. Art is thus seen as a mode of cognition (*Erkenntnis*) and has primarily to do with a "revelation of truth" (*Entfaltung der Wahrheit*) (G.W.F. Hegel qtd. in [Ado49]). I do not wish to advocate the idea of a universal, absolute "truth", but would rather like to connect with Hegel's notion of art as a mode of cognition (*Erkenntnis*).

The American philosopher Nelson Goodman sees aesthetic experience as cognitive experience and the arts, just as the sciences, "as modes of discovery, creation, and enlargement of knowledge,"[Goo68, page 102] i. e. understanding. Goodman claims that the arts, just as the sciences, deal with symbol systems:

> The difference between art and science is not that between feeling and fact, intuition and inference, delight and deliberation, synthesis and analysis, sensation and cerebration, concreteness and abstraction, passion and action, mediacy and immediacy or truth and beauty, but rather a difference in domination of certain specific characteristics of symbols.[Goo76, page 264]

Music has had a unique position among the disciplines of art, in that it naturally escapes reification. Music has not served clear representational purposes, it is difficult to determine its "material", and since music had been bound to performance, it had not been tied to any physical object prior to the invention of the phonograph in the late 19th century. The relationship of matter and thought in music is less clear, thus musical works have always been to greater extend conceived of as mental representations. In his book *Art as Experience*, John Dewey describes the relationship of matter and thought and art and science with regard to the creative act

> The odd notion that an artist does not think and a scientific inquirer does nothing else is the result of converting a difference of tempo and emphasis into a difference in kind. [...] The artist has his problems and thinks as he

> works. But his thought is more immediately embodied in the object. [. . .], the scientific worker operates with symbols, words and mathematical signs. The artist does his thinking in the very qualitative media he works in, and the terms lie so close to the object that he is producing that they merge directly into it. [Dew34, page 14]

Dewey associates the scientists work with "symbols", whereas he sees the artist's thoughts directly linked and embodied in the object. He could not foresee the invention of the computer as an instrument for art based on "symbolic matter", but he stresses that there is no "difference in kind" between the thinking of the artist and the scientific inquirer. In music, where the process of creation takes place largely in a non-material world, in which the "material" is per se symbolic (free of an object carrier), the symbolizing ability of the computer renders it to be a natural extension of the artist's collective resources. The interposition of the computer in the compositional process entails an introduction of a symbolic representation that mediates the construction of the composition. In contrast to earlier methods of representation in music, i.e. notation, the computer shifts interpretation from the product-oriented music notation to process-oriented programs; notation and production method merge in a work process featuring a cyclical interaction.

In his article "Computer-Verwendung in Kompositionsprozessen" Gottfried Michael Koenig describes the relationship between composer and computer:

> Der Komponist kann der Frage nachgehen was eine Regel ist, [. . .] auf welche Weise verschiedene Regeln sich beeinflussen. Der Komponist lehrt den Computer verstehen und sprechen; was der Computer sagt, zeigt dem Komponisten, was er selber verstanden hat und aussprechen konnte.[1] [Koe93a]

This quote contains three important general ideas about what is gained by using computer programs in composition processes: the idea of *rules* as something that can be formulated in terms of a computer program, the idea of composing with inter-dependent rules that influence each other, and the general approach of *pursuing a question.* The latter is of capital importance in that it describes a fundamental motivation for formulating musical ideas in terms of rules; an inquiry into composition.

[1]English translation: "The composer can pursue the question what a rule is, [. . .] in which way different rules influence each other. The composer teaches the computer to understand and speak; what the computer says shows the composer what he himself has understood and could express."

In a similar line of argument, but from the point of view of computer music language designers, Roger B. Dannenberg, Peter Desain, and Henkjan Honing write in their paper "Programming language design for music":

> Implementing ideas on a computer often leads to greater understanding and insights into the underlying domain. Programming languages can be developed specifically for music. These languages strive to support common musical concepts such as time, simultaneous behavior, and expressive control. At the same time, languages try to avoid pre-empting decisions by composers, theorists and performers, who use the language to express very personal concepts. This leads language designers to think of musical problems in very abstract, almost universal, terms. [DDH97]

As a question about rules is pursued, the composer defines such rules and will then see their consequences, and will then gain insight into the nature of the rule and his or her own understanding. By the abstraction the computer demands, the idea of musical problems is augmented and through interaction the computer can become a source of compositional ideas. Rather than being a passive, mechanically executing medium, a programming language can inspire musical ideas; as John Chowning states in an interview with Curtis Roads:

> More and more, the musical idea evolves from a kind of cyclical interaction with the language. One asks something of the language and it yields more than you asked for. That's not surprising since the language represents thousands of years of thought about thought. [Roa85, page 23]

The computer is seen here as an extension of mental activities, a tool for gaining knowledge. The goal is not to derive rules from existing music, i.e. to *formalize* music, but to use the computer in an exploring process of inquiry. Herbert Brün formulates this distinction felicitously with regard to sound synthesis:

> It is one thing to aim for a particular timbre of sound and then to search for the means of making such a sound and timbre audible. It is another thing to provide for a series of events to happen and then to discover the timbre of the sounds so generated. [Brü69]

Only in a real examination of the medium and its possibilities can the computer find a useful place in music. The use of the computer enables the composer to experience "musical form as a process"[Koe93c], not only in listening, but in a work process. Computer music programs require the construction of models on two levels, they form a model of the composer's workplace and a generalized model of composition. The computer is thus not merely a practical tool, used for its computational performance, but becomes an *Erkenntniswerkzeug.*

2.4. Interpretation

The use of algorithmic methods in the composition of music constitutes an externalization of a part of the *composition process*. As stated above, depending on the initial intentions not all parts of this process may be externalized equally well. In computer-assisted composition the computer is not only used for the mere production of sound, but involved in the processing of abstract signs and relationships. The computer is included in a phase of planning and may also be part of the final realization. The output of a computer program may also be translated into performance instructions which are remote from the program itself, specifically in the algorithmic composition of instrumental music. In any way, there is a translation of the output data into an *aesthetic object*, i.e. a musical performance, be it the playback of a tape, a performance by instrumentalists or another form.

Gottfried Michael Koenig terms this process *"aesthetic integration"*[Koe93d]. As Koenig argues, the process of "aesthetic integration" is a consequence of algorithmic composition:

> Die Algorithmen verkörpern die allgemeine Idee eines Stückes, während die musikalischen Daten nur abstrakte Beziehungen unterhalten und die Ausarbeitung der Partitur unterstützen, die diese Beziehungen konkretisiert.[2][Koe93d]

In this way the composer himself becomes an interpreter of the material, with which the externalized process has provided him. The distinction between the "general ideas", "musical data", their interpretation, and their concretization elucidates the various steps in the composition process in which the computer is differently engaged. On the one hand this distinction is often blurred, as can be seen in many recent computer music systems, and on the other hand, the generality and openness of some current systems enables the composer to influence the process at any state, and may therefore help to resolve the conflict between initial intentions and eventual output.

[2]English translation: "The algorithms embody a general idea of a piece, while the musical data only maintain abstract relations and support the realization of the score, which concretizes these relations."

3. Model and Music

Two Musical Examples

3.1. Introduction

In this chapter, I will discuss one piece by Gottfried Michael Koenig called *Übung für Klavier*, which he realized using his program *Project 2* and a family of pieces, here called *ST* pieces, by Iannis Xenakis. Although the initial intentions of *Übung für Klavier* and those of the *ST* pieces, as well as the concrete resulting music, differ greatly, their production shows several points of intersection. Both works are realized using programs, which form a conscious generalization of previous work and can be seen as experimental and speculative models. Their axiomatic approach constitutes a certain simplification of musical processes; a reduction to the substantial essence of music.

3.2. Koenig's Übung für Klavier

Gottfried Michael Koenig's piece *Übung für Klavier*, composed in 1969, is the first piece he realized with his program *Project 2*. *Project 2* was not designed for the realization of a single piece, but as a general composition program. As Koenig writes in the preface of the score, it is therefore, "necessary to generalize individual composing habits; an attempt must be made to formulate a theory – however limited – of composition." [Koe69] *Übung für Klavier* (Study for Piano) is thus a first test of this theory. The word *Übung*, meaning 'study', but also 'practice', does not primarily refer to the player or the instrument, but rather to the composition of the piece; it is a study in writing a piece with the program *Project 2*. The title thus identifies the work as a test object and reveals the critical reflection of the work itself on the model and material from which it is derived.

The piece consists of 12 structures and 3 variants of each of these structures, of which the pianist chooses one variant for each structure to be played. Hence, there are $3^{12} = 531441$ possible combinations of variants, of which one is performed.

Project 2 is based on a number of basic notions: compositional rules, musical quantities (*data*), characteristics of musical sound (*parameters*), combinations of rules and data (*structure formula*), and "combinatorial possibilities" [Koe70a] resulting from a structure formula (*variants*). As Koenig states, "The purpose of PROJECT 2 (PR-2) is to 'calculate musical structure variants'." [Koe70a] Since aleatoric decisions are employed in different phases of the program, it is not necessary to enter additional data for the creation of *variants* from a *structure formula.* The rules and the set of data are fixed for a specific structure and the computer constructs variants.

By following a questionnaire of over 60 questions, the *composer* describes a certain *model* of which *variations* are created. In *Übung für Klavier*, the 12 structures are *structure formulas*, variants are created by the use of aleatoric procedures. There are eight parameters that describe a *structure formula*: instrument, harmony, register, entry delay, duration, rest, dynamics, and mode of performance. Since the piece is only for one instrument, the instrument parameter is ignored.

3.2.1. The List-Table-Ensemble Principle

The basic principle for the construction of musical data in *Project 2* is a three-layered process of entering, grouping, and selecting elements, the so-called *List-Table-Ensemble* principle. The construction of data for almost all parameters follows this principle. In the first instance, the composer enters a list of "allowed" elements; a basic reservoir of the smallest components. In the second layer, the user forms groups of these elements in a table, a list of selections from the list of "allowed" elements. Consequently, an *ensemble* is formed by selecting groups from the table. Thus, there are three layers of selection, i.e. choices of elements from a given supply. The first two selections are done by the composer. The composer chooses the basic elements and determines their grouping in the table. These selections and groupings remain the same for all variants of a structure. The third selection, however, is done with the help of the selection programs *alea*, *series*, and *sequence* (q.v. *Selection Principles* in this section), which chose the number and indices of the table-groups to be inserted into the *ensemble.* The third level differs from the first two levels of selection in that the selection can be changed for each variant and not single elements, but whole groups of elements are selected.

The *List-Table-Ensemble* principle is an extension of the series as a basic building block, which can be permutated in order to derive relationships. Aside from the input of the initial reservoir of “allowed” values, the other levels operate on indices. The concrete values are substituted by pointers to concrete values. The operation on pointers is an abstraction, which constitutes an intermediate meta-level, through which aspects of the musical reality become controllable. In doing so, numbers operated on never refer to themselves, i.e. calculations and the construction of numerical structures are not done for their own sake, but always for the purpose of referring to concrete values. Thus, numerical values always serve the description of musical situations. The translation of concrete values into indices creates a level on which processing is possible. The *List-Table-Ensemble* principle clearly discriminates between the material and its order. The initial input of “allowed” values is an unordered set of possible elements and the operations on indices establish orders and groupings, thereby breaking the series up into material and sequence. Composer Karlheinz Essl argues in his text “Zufall und Notwendigkeit”, that this conception of numerical values as an intermediate compositional level stems from Koenig's experiences in the electronic studio:

> Dort, wo die Ebene der musikalischen Notation überhaupt wegfällt, wird die numerische Übersetzung kompositorischer Daten und Vorgänge zu einer unabdingbaren Forderung bei der Realisierung elektronischer Musik. Die Notwendigkeit der Abstraktion ergibt sich zudem aus den Gegebenheiten des Studios [. . .].[1] [Ess89]

3.2.2. Selection Principles

For the selection of table-groups, for the formation of the score from ensemble data, and a few other purposes, there are selection principles used. These principles typically establish orders of the elements of a given supply. The selection programs are mainly of an aleatoric nature. For Koenig, a described structure, such as the *structure formula*, or a given set of elements, from which a selection is made, has a potential, which can be manifested in many ways. The use of *constrained randomness* here, is quite different, than for example in the work of Iannis Xenakis. Whereas Xenakis works with different random distributions whose specific characteristics are crucial and often audible in the result, i.e. random processes themselves are used for the creation of musical data, Koenig uses randomness mostly for creating variation. Aleatoric decisions are used to

[1]English translation: “Where the level of musical notation in general ceases to apply, the translation of compositional data and processes becomes an indispensable demand in the realization of electronic music. Moreover, the necessity of abstraction arises from the conditions of the studio.”

create variable decisions within fields of defined possibilities, not in order to introduce a character of randomness (*Zufallscharakter*) into the musical output.

> Wir wollen darunter [reduzierter Zufall] eine einmalige aleatorische Entschiedung innerhalb eines Feldes möglicher Entscheidungen verstehen oder doch so weniger Entscheidungen, daß der Zufallscharakter empirisch nicht in Erscheinung tritt; der Zufall lenkt zwar, wird aber nicht als solcher erkannt.[2] (Koenig qtd. in [Boe67])

The *selection principles* used in *Project 2* are show in table 3.1. *Alea* is the only principle which has no memory, i.e. after an element has been selected all elements are still equally likely to be selected next. *Series* creates a random permutation of the given supply. The selection principles *Ratio* and *Group* are results of a discourse in serial music. In order to construct higher-level compositional units (*groups*), and find ways out of the pointillistic style, the basic rule of serial music, the avoidance of repetition, is abolished. Instead, a controlled and serially organized form of repetition is introduced. Karlheinz Essl describes Koenig's approach towards repetition as a generalization of Stockhausen's theory of groups. [Ess89, page 45] In his text "Gruppenkomposition: Klavierstück 1", Karlheinz Stockhausen defines the term "group":

> Mit "Gruppe" ist eine bestimmte Anzahl Tönen gemeint, die durch verwandte Proportionen zu einer übergeordneten Erlebnisqualität verbunden sind, der Gruppe nämlich.[3] [Sto63]

The principle *tendency* has a different nature than the previously described methods. In contrast to the other principles, it does not only select from a supply, but creates a shape in time, it thus introduces directionality. In contrast to the other principles, whose outputs could also be reversed and may still be valid, the *tendency* principle works in time. The connection with time is also the reason why Koenig employs the *tendency* principle especially frequently for the selection of entry delays, in fact he uses the principle in 8 of the 12 structures of *Übung für Klavier*. [4]

[2]English translation: "We want to understand it [reduced randomness] as a unique aleatoric decision within a field of possible decisions, or, however, decisions in which the character of randomness does not appear empirically; randomness guides, although it is not recognized as such."

[3]English translation: "'Group' means a certain number of tones, which form a higher-level quality of experience, the group, through related proportions."

[4]Koenig also mentioned the suitability of the *tendency* principle for the selection of entry delays in a lecture given at the *Technische Universität Berlin* on the 24th of January, 2003.

alea (a,z)	Uniform random choice between **a** and **z**.
series (a,z)	Uniform random choice between **a** and **z** without repetition. If an element was selected, it can not be selected again until all remaining elements have been selected.
ratio (p_1,p_2,...,p_n)	The same as **series**, but the factor **p** determines how often an element can be selected before it is blocked.
group (a,z,type)	The elements are selected by either **alea** or **series** and repeated in groups. The size of the groups is also selected by either **alea** or **series** and the limits are determined by **a** and **z**.
tendency (d_1,$a1_1$,$a2_1$,$z1_1$,$z2_1$,...)	Uniform random choice between changing borders, starting between $a1$ and $a2$ ending between $z1$ and $z2$. There can be any number of "sub-tendencies", d_i determines the relative length of "sub-tendency" **i** within the whole mask.
sequence (i_1,i_2,...,i_n)	The order of the elements is directly defined by the user.

Table 3.1.: Selection Principles

The *selection principles* also represent a perspicuous formulation of a generalized idea of composition processes, namely the idea of *composition as selection.* On different levels the material is clearly defined, firstly a selection is made by the composer, secondly generalized methods of selection are applied to a field of possibilities, consequently leading to the description of structures with certain potentials. The idea of selection and variation, of providing fields of possibilities, which are articulated by specific selections, pervades *Übung für Klavier* on many levels. On an outer level, the performer is involved in creating the final form of the piece. The variations from which the composer selects are themselves possible concretizations of a *structure formulas*, which in turn are aggregates of material, in the first instance from a given supply of data determined by the composer, and secondly selected and ordered with the help of the *selection principles.* G.M. Koenig's *Übung für Klavier*, therefore, shows a unique clarity in dealing with decision-making processes.

3.2.3. Harmony

Although *Project 2* allows the composer to establish a parameter hierarchy and set a "main parameter", which resolves certain constraints stemming from interdependent parameters, most parameters are dealt with in the same way, the *List-Table-Ensemble* principle. There is, however, one important exception, the *harmony* parameter, which determines the pitch organization. There are three different principles used: *chord*, *row*, and *interval.* In the *chord* principle, the user enters a chord table which is then sorted by using a selection principle. Thereafter, the chords are transposed, using either *alea*, *series*, a given row, or no transposition. The *row* principle establishes a sequence of rows derived from a given input row. These transpositions can be produced in four ways: transposition with *alea*, with *series*, chromatically ascending, or by using the same row as a row of transposition intervals. The third principle, *interval*, uses a chord entered by the user and analyzes it according to its interval content. From this interval content a matrix is build, which determines which intervals can follow each other and which cannot. The special treatment of the pitch parameter indicates a supremacy of this parameter over the others. The *harmony* stands out, in that it is the only one which employs methods, which are specific to its domain.

3.2.4. The 12 Structures of Übung für Klavier

In the preface of the score of *Übung für Klavier*, Gottfried Michael Koenig roughly describes the "total plan" according to which the "elementary constellations" were arranged:

> Sequences of tones start rapidly in a narrow range and gradually get slower, at the same time covering a wider pitch range (structure 1). [. . .] Rapid tone sequences are left over; they get quicker and slower whilst the range narrows, widens and narrows again (structure 3).[. . .] A transition to the last group of structures is formed by chords which start regularly and then become irregular [. . .] (structure 7); groups of chords then alternate with groups of tones (structure 8). [. . .] Groups of tones of equal length occur at various speeds, linked by means of the pedal (structure 11), and finally there are two-part configurations, rapid at first, then in a composed ritardando (structure 12). [Koe69]

Although Koenig sees some structures as transitions, such as structures 3 and 7, his description of the general course of the piece shows that *Übung für Klavier* is permanently in transition. Despite its serial and combinatorial nature, the piece continually exhibits directionality on many levels. Entry delays, pitch ranges, and dynamics move throughout the structures, there are hardly any moments of standstill. This is in most parts the effect of using *tendency masks*. The extensive use of this principle might stem from the fact that it was one of the main novelties of *Project 2* in comparison to its precursory models.

Koenig defines three classes of basic material for *Übung für Klavier*: rapid tone sequences ("passages"), single tones ("tones"), and chords. Table 3.2 shows the durations in seconds of the three variants, the basic material, the harmonic principle used, and the selection principle used for the generation of the entry delays as well as their minimum and maximum values in seconds for the 12 structures of the piece. Koenig establishes many cross-references between the structures by reusing and transforming material. Figure 3.1 shows the pitch material used in the twelve structures and their relationships. The used "basic material" and the relationships of the pitch material indicate that the piece can be roughly divided into three parts. The first part (structures 1–3) consists mainly of single tones and passages of varying density and movement in pitch range. The *interval* principle is used in all of the three structures, using the same six-tone chord. Structures 4 to 7 feature mainly chords, which are different transpositions of the chord table used in structure 4, which in turn is constructed by selecting several two-, three- and four-note combinations of tones from the chord used in structure 1. The final part of the piece (structures 7–12) returns to the original *interval* principle of the first part.

Considering that the piece clearly stands in a serial tradition, it is remarkable that it remains quasi-monophonic throughout most structures.

> At first the piano was treated as a one-part instrument (structures 1, 3, 9, 11) and then at least as a one dimensional one: chords in succession (structures 5, 7). A concession to wider performance possibilities was made by combining rapid and slow tone sequences (structures 2, 10), and the same was done with tone sequences and chords (structures 4, 6, 8). Not until the last structure (12) is the instrument treated polyphonically. [Koe69]

Despite of the piece's continuous development, created by shapes and masks, the established cross-references, transformations of material, and variant-oriented way of composing show a non-linear aspect of the form. Rather than seeing form as a container to be filled with events or as hierarchical top-down structure, form and material are mutually dependent and created in one process. As in other pieces by Koenig, such as *Streichquartett 1959* or *Terminus*, the form is determined by degrees of relationship, intermediation of contrasts, and transitions. Koenig's approach to form, therefore, concurs with Adorno's notion, that the form is only *substantial* if it emerges from the demands of the material itself:

> Der unreflektierte, in allem Gezeter über Formalismus nachhallende Formbegriff setzt Form dem Gedichteten, Komponierten, Gemalten als davon abhebbare Organisation entgegen. Dadurch erscheint sie dem Gedanken als Auferlegtes, subjektiv Willkürliches, während sie substantiell ist einzig, wo sie dem Geformten keine Gewalt antut, aus ihm aufsteigt. Das Geformte aber, der Inhalt, sind keine der Form äußerlichen Gegenstände sondern die mimetischen Impulse, welche es zu jener Bilderwelt zieht, die Form ist.[5] [Ado73, page 213]

[5]English translation: "The unreflected idea of form [Formbegriff], which re-echoes in all the clamor about formalism, opposes form as a removable organsition to the versed, the composed, the painted. Thereby it [form] appears as imposed on the thought, as subjectively arbitrary, whereas it is only substantial, where it does not violate the formed, but arises from it. The formed [das Geformte], however, the content, are no objects external of the form, but mimetic impulses, which are drawn to this imagery [Bilderwelt], which is form."

struc	**variant durs**	**basic material**	**harmony**	**e.d. selection (range)**
1	35/46/67	passages	interval	tendency (0.1–1.72)
2_1	18/25/39 (total)	tones	row	series (1.11–3.32)
2_2		tones/passages	row	series (1.11–3.32)
2_3		tones/passages	interval	ratio (0.1–2.67)
3	15/24/34	passages	interval	tendency (0.1–0.72)
4_1	35/48/81 (total)	chords	chord	ratio (0.3–2.14)
4_2		chords/passages	row	tendency (0.1–1.38)
4_3		chords	chord	ratio (0.46–3.32)
4_4		chords	chord	tendency (0.46–3.32)
5	19/30/34	chords	chord	tendency (0.24–2.67)
6_1	32/36/55 (total)	chords	chord	group (0.46–1.38)
6_2		tones	row	alea (1.11–2.67)
7	22/28/42	chords	chord	tendency (0.46–1.38)
8_{chords}	41/71/85 (total)	chords	chord	alea (0.24–0.89)
8_{tones}		tones	interval	alea (0.46–1.72)
9	21/34/44	passages/tones	interval	tendency (0.1–1.72)
10_1	18/34/53 (total)	passages/tones	interval	tendency (0.1–1.72)
10_2		passages/tones	interval	tendency (0.1–3.32)
11	32/51/49	passages	interval	series (0.1–0.46)
12_1	42/42/42	passages/tones	interval	alea (0.1–1.11)
12_2	28/27/36	passages/tones	interval	tendency (0.1–0.72)

Table 3.2.: The 12 structures

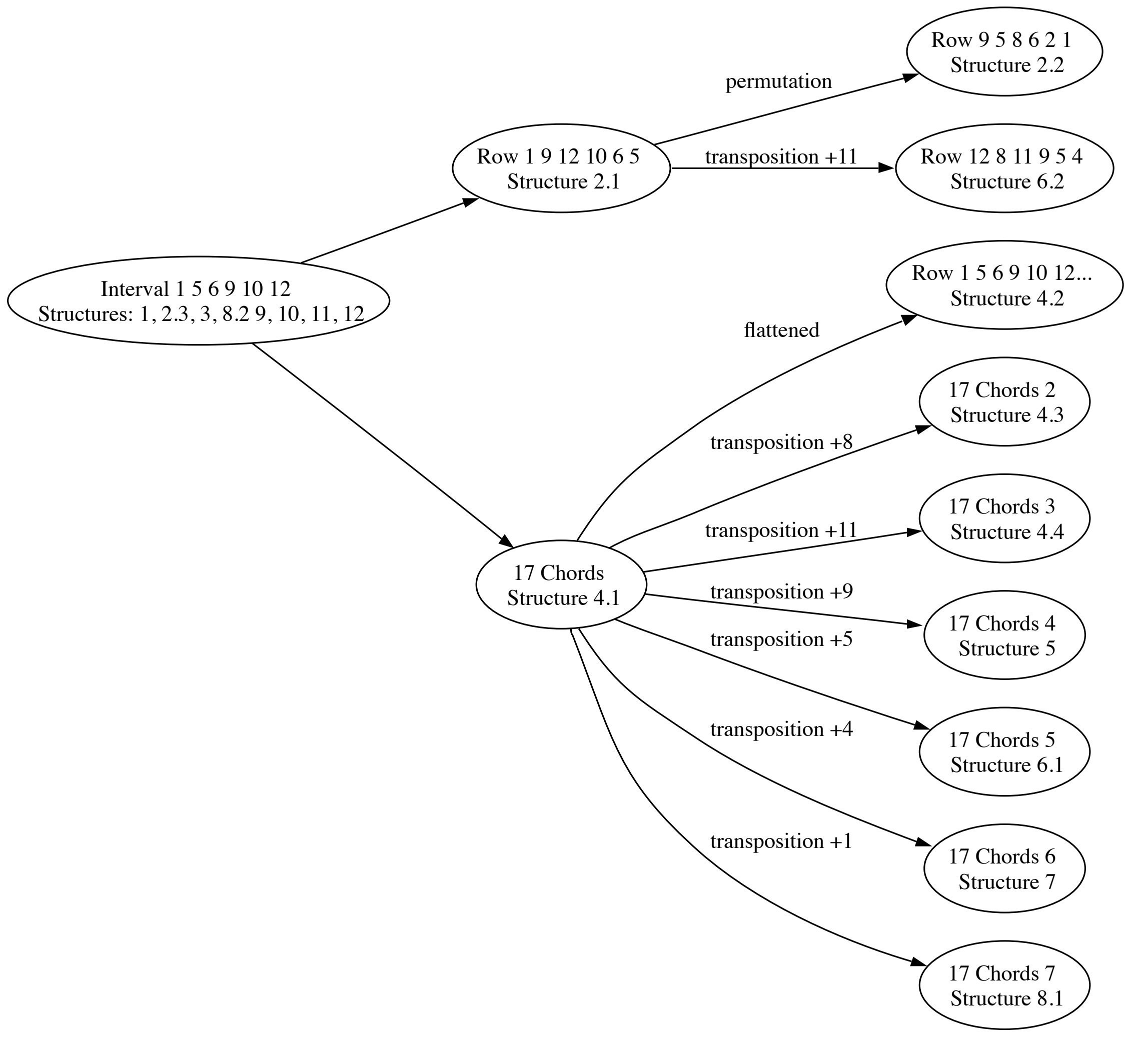


Figure 3.1.: Relationships of the pitch material

3.3. Xenakis's ST pieces

Xenakis's idea of the "fundamental phases of a musical work", described in section 2.2.2, was developed during the composition of his piece *Achorripsis* (jets of sounds) (1957) for chamber orchestra, a piece which is centered around the above mentioned question: "What is the minimum of logical constraints necessary for the construction of a musical process?" [Xen92, page 16] In 1962 Xenakis had the opportunity to work on an IBM 7090 computer and composed a family of pieces (*ST* pieces), which form a continuation of his "fundamental phases of a musical work" in form of an algorithm implemented in a computer program.

3.3.1. Music Ex Nihilo

One can see Xenakis's *ST* pieces in the context of the question above as a fundamental research on musical universals. Xenakis asks the question, "how to create, figuratively speaking, a 'black-box' which has music at the other end, and not just music, but interesting music?" [Var96, page 80] His idea is to create an automaton starting from a zero-point, a point with a "minimum of logical constraints", which can lead to a reconstruction "of the basic ideas of composition." *ex nihilo.* [Xen92, page 207]

> [...] we shall begin by imagining that we are suffering from a sudden amnesia. We shall thus be able to reascend to the fountain-head of the mental operations used in composition and attempt to extricate the general principles that are valid for all sorts of music. [Xen92, page 155]

Xenakis is advocating the search for a generalization that leaves Western art music as a special case.

> Starting from certain premises we should be able to construct the most general musical edifice in which the utterances of Bach, Beethoven, or Schönberg, for example, would be unique realizations of a gigantic virtuality, rendered possible by this axiomatic removal and reconstructions. [Xen92, page 207]

It remains, however, highly questionable whether such a generality can be achieved. Xenakis denies the inherent and irremovable historic quality of musical material, although his own work clearly exhibits a multitude of influences and references. *Achorripsis* and the *ST* pieces constitute a form of simplification, a generalization of musical activity; one might say an "approximation of music". The use of probability functions as an aesthetic phenomenon, and not as an externalization of compositional decision-making processes, may be seen as the construction of a generalized model, with its material and constructions embedded in a historic frame just like any other work of Xenakis, and therefore despite of it speculative, constructivist nature not *ex nihilo*.

3.3.2. The ST Algorithm

1	The work consists of a succession of sequences each a_i seconds long.
2	Definition of the mean density of the sounds during a_i.
3	Composition Q of the orchestra (from r classes of timbres) during the sequence a_i.
4	Definition of the moment of occurrence of the sound N within the sequence a_i.
5	Attribution of the above sound of an instrument belonging to orchestra Q.
6	Attribution of pitch as a function of the instrument.
7	Attribution of a glissando speed if class r is characterized as a glissando.
8	Attribution of a duration x to the sound emitted.
9	Attribution of dynamic forms to the sound emitted.
10	The same operations are begun again for each sound of the cluster N a_i.
11	Recalculations of the same sort are made for the other sequences.

Table 3.3.: Summarized *ST* Algorithm (from [Xen92])

The structure of the algorithm used in the *ST* pieces, which is shown in table 3.3, is essentially the same as can been seen in the flow chart for the creation of *Achorripsis* (see [Xen92, page 135]). Xenakis saw the structure of the *ST* program as an automaton, a "black-box". The "black-box" approach seems surprising, in that it indicates, that Xenakis did not try to "solve particular structural problems" [Koe93b], by externalizing certain aspects in the composition process, but rather tried to find a general model for the automated generation of entire pieces. The *ST* pieces are on the one hand a generalization of his previous work in "free stochastic music", i.e. its "rules" derived at least partly by introspection, but on the other hand it is also a speculative model, which searches for musical universals, for "the general principles", by defining a minimum of rules and then navigating that space of possibilities.

Freed from tedious calculations the composer is able to devote himself to the general problems that the new musical form poses and to explore the nooks

> and crannies of this form. [...] With the aid of the electronic computer the composer becomes a sort of pilot: he [...] supervises the controls of a cosmic vessel sailing in the space of sound [...].

A piece produced with this method is first divided into a number of sections, or sequences, which consist of events with certain attributes. Most decisions are made with the help of differently constrained probability functions, primarily using Poisson's law of rare random events and Gaussian distribution. Chance, however, is not used as a selection mechanism, but rather the phenomenon of randomness itself is translated into sounding matter. Randomness, therefore, becomes an aesthetic principle and a philosophy of composition. With respect to *ST/10-1, 080262*, Xenakis states, "by its passage through the machine, this work made tangible a stochastic method of composition, that of the minimum of constraints and rules." [Xen92, page 134] The Poisson distribution, which is used for the generation of densities, section durations, pitch, and the moment of occurrence of a sound in a section, is generally used to model the number of occurrences of in itself rare events over a fixed period of time.

> It [the Poisson distribution] represents the number of occurrences, per unit time, of an event that can occur at any instant of time. For example, the number of alpha particles emitted by a radioactive substance in a single second has a Poisson distribution. [Knu69, page 132]

Musical events are thus represented as "rare random events" in time. Already with *Achorripsis*, Xenakis extended his application of probability functions to larger sections of the work, thus creating the entire form by the use of probabilities.

This model of the *ST* pieces should not describe a single piece of music, but a class of pieces; Xenakis has compared the model to the idea of a fugue as an abstract automaton, which can be used to create an infinite number of pieces. The structure of the algorithm reveals the supremacy of both timbre and time over other parameters. The most defining differences between the sections of the *ST* pieces are in their density and timbral characteristics. The pitches, which are assigned to the events, are generated using random walks with a Poisson distribution, and have been subject to change by Xenakis during the translation process of the output data into musical notation.

The output of the program required manual transcription and indeed Xenakis interpreted the output in parts rather freely. In *Atrées*, Xenakis even combined generated and differently composed material, as Matossian writes, "he used seventy-five per cent computer material, composing the remainder himself." [Mat05, page 208] Although, Xenakis's model used in the *ST* pieces and *Achorripsis* is speculative and the pieces are testing the validity of a generalized theory of composition, his way of dealing with the output data indicate that on a practical level he did not see the pieces primarily as "test objects", as Koenig did with *Übung für Klavier*, but used the program in parts, contradicting his statements, rather as a generator of musical material. Moreover, he also reused the original data and output, such as in *ST/4* for string quartet, which is based on *ST/10*. This shows that not only the program describes a class a pieces, but even one output of the program can lend itself for the composition of several pieces.

Part II.

System

4. The Design of CompScheme

4.1. Overview

CompScheme is a program for algorithmic music composition and stochastic sound synthesis written in Objective Caml (*OCaml*) [Ler] I have been developing since 2006, *CompScheme* can be used in two ways, as a library for developing applications in *OCaml*, or by accessing its functionality interactively through an interface language[1]. The primary value generating mechanism in the program are *streams*, which are a concept from functional programming, allowing the user to concisely describe networks of dynamic data. *Streams* themselves are rules for generating values. Since streams can be combined and even used to generate new streams, the rules themselves become the object of composition. "Composing with rules" is thus not only interpreted as the mere application of a rule but the actual composition of rules, an idea that is very prominent in functional programming, if one sees rules as functions.

The interface language of *CompScheme* is an implementation of the functional programming language *Scheme*[2] [ASS96]. Instead of having a graphical environment or a fixed work flow, where the user can either visually or by filling out forms or questionnaires construct networks and derive musical data, *CompScheme* requires the user to have a degree of programming proficiency. By using an elegant, popular, small, and powerful general-purpose language such as *Scheme*, the user has all of its expressiveness and means of abstraction at his or her disposal. The user can develop full-range programs, or make small experiments by plugging together built-in streams and output functions.

Internally, *CompScheme* consists of several modules, which contain functions for specific fields of application. Data generated in *CompScheme* can be written out in several

[1]All the code examples in this text are written in the interface language.
[2]The implementation of the interface language is based on *Schoca*: `http://home.arcor.de/chr_bauer/schoca.html`

ways, such as in Wav audio files, Midi files, and binary OSC files for *SuperCollider*. It is also possible to control the *SuperCollider* server in real-time, plot and draw data. Furthermore, *CompScheme* has an event type system, which features built-in event types, and the possibility to create custom event types by bundling named parameters, setting default values, creating transformation and output functions.

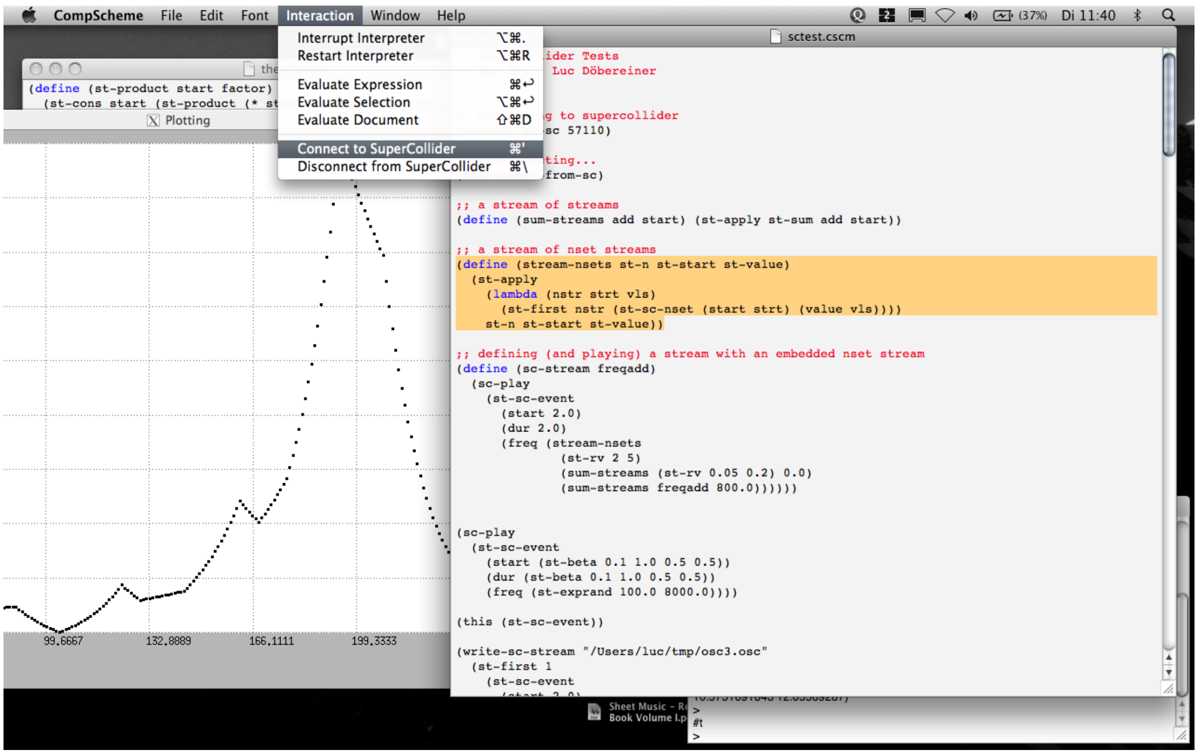
CompScheme runs on *Mac OS X* and *Linux*. The top-level interpreter can run as a command line program, in a *Scheme*-mode in an editor such as *Emacs*, or on *Mac OS X* in a specially developed *Cocoa*-application, which follows the usual editor-and-listener design.

In this chapter, I first discuss the general motivation for choosing a programming language as an interface of a composition system. Secondly, I will discuss the concept of *streams*, as well as some issues and design ideas of *CompScheme*, which relate to *streams*. In chapter 5, I present *CompScheme*'s event model, which provides a framework for building abstract structural musical units. The discussion of the event model is followed by a concrete example, the modeling of the first structure of Gottfried Michael Koenig's piano piece *Übung für Klavier*. Subsequently, the expressiveness of multi-layered *event streams* is exemplified by presenting *CompScheme*'s facilities for controlling the *SuperCollider* server (SC Server) in real-time. Finally, chapter 6 deals with *CompScheme*'s functions for stochastic synthesis. Starting from finding a generalization of G.M.Koenig's *SSP* and I.Xenakis's *dynamic stochastic synthesis*, I have tried to develop a framework, which facilities experimentation in this field. We will see how the event model can be applied to a lower level, the digital sample itself.

4.1.1. Why text and parentheses?

As H. Abelson and J. Sussman state in their book *Structure and Interpretation of Computer Programs*, "programs must be written for people to read, and only incidentally for machines to execute." Programming languages are primarily tools to express ideas. The formal nature of programming languages stipulates abstraction and generalization. Thus, through programming the structure of an idea may be revealed. Our means of expression shape what we can express. As Ludwig Wittgenstein famously formulated, "die Grenzen meiner Sprache bedeuten die Grenzen meiner Welt."[3][Wit18] Music composition programming languages, therefore, influence our ideas of music. It may, thus, be argued, that the choice of a programming language also has musical consequences.

[3]English translation: "The limits of my language mean the limits of my world."



```
(define (st-product start factor)
  (st-cons start (st-product (* st
```

```
ider Tests
Luc Döbereiner

g to supercollider
sc 57110)

ting...
from-sc)

;; a stream of streams
(define (sum-streams add start) (st-apply st-sum add start))

;; a stream of nset streams
(define (stream-nsets st-n st-start st-value)
  (st-apply
    (lambda (nstr strt vls)
      (st-first nstr (st-sc-nset (start strt) (value vls))))
    st-n st-start st-value))

;; defining (and playing) a stream with an embedded nset stream
(define (sc-stream freqadd)
  (sc-play
    (st-sc-event
      (start 2.0)
      (dur 2.0)
      (freq (stream-nsets
              (st-rv 2 5)
              (sum-streams (st-rv 0.05 0.2) 0.0)
              (sum-streams freqadd 800.0))))))


(sc-play
  (st-sc-event
    (start (st-beta 0.1 1.0 0.5 0.5))
    (dur (st-beta 0.1 1.0 0.5 0.5))
    (freq (st-exprand 100.0 8000.0))))

(this (st-sc-event))

(write-sc-stream "/Users/luc/tmp/osc3.osc"
  (st-first 1
    (st-sc-event
```

```
>
#t
>
```

Figure 4.1.: The *CompScheme* Cocoa-application

In the discussion about the right programming approach for computer music, the advocates of visual programming may say, “a picture says more than a thousands words”, and visual information can be grasped more intuitively. This may be true, especially with respect to ambiguous information. Programs, however, need to be unambiguous, and the strength of programming languages lies in their means of abstraction. With the exception of smaller networks, which can be concisely expressed using visual programming languages, and in which the otherwise so disarraying mélange of visualization and algorithm is not yet so apparent, visual programming languages are generally inferior to text-based languages with regard to means of abstraction. “Flowcharts are *abstraction-hating*. They only contain decision boxes and action boxes.” [GP96]. Moreover, the logic of evaluation is often harder to understand, searching in the code is more difficult, and once a program reaches a sufficiently developed size it is almost impossible to maintain an overview (*Deutsch Limit*).

Computer music systems, however, can be of different types. Some are systems where the user has a sort of dialogue, and others are more general purpose languages, offering the user a set of basic functions, data structures, and means of combination. *CompScheme* falls into the second category and instead of inventing a new language, I have

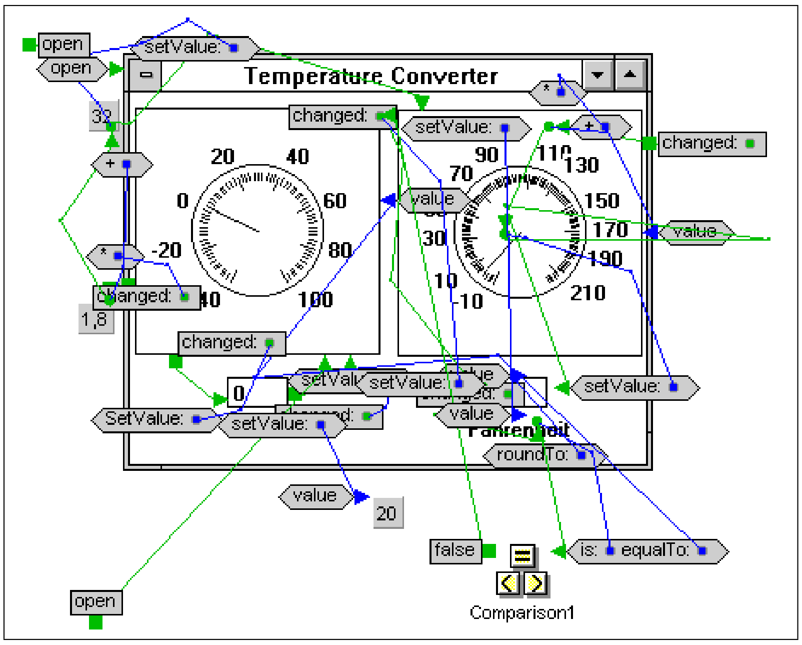


Figure 4.2.: A *PARTS* program converting values in Celsius to Fahrenheit

```
(define (celsius-to-fahrenheit c) (+ (* (/ 9 5) c) 32))
```

Figure 4.3.: A *Scheme* program converting values in Celsius to Fahrenheit

chosen to use a language, which is small, elegant, and powerful and moreover already popular in computer music, *Lisp*.

4.2. Streams

A musical performance or playback is a continuous stream of sound. In any computer representation this continuum, however, is broken up into a discrete sequence of values. The common digital representation of sound in the form of a sampled waveform, common practice musical notation, as well as an event-based higher-level musical abstraction must follow this rule. *CompScheme* is built around the data type of streams, which are an elegant and simple way of dealing with sequences of values, that is widely known, used and "one of the most celebrated features of functional programming"[Pau96]. Whereas, in most imperative and object-oriented systems these sequences are usually created by some iteration that collects the values or a mechanism that involves a change of state, *streams*, however, are persistent.

> Stream processing lets us model systems that have state without ever using assignment or mutable data. This has important implications, both theoretical and practical, because we can build models that avoid the drawbacks inherent in introducing assignment.[ASS96]

This persistence also has advantages in musical applications. The main one, of course, is that no values are lost and everything that has been produced, and therefore everything that *will* be produced, can be referred to, which provides the user with the possibility to look into the "future" of a process and make decisions depending on what is going to happen.

Streams are flexible, they can be combined, can contain values of any kind, such as other streams or functions, and are collections as well as generative mechanisms. In more imperatively oriented systems, values are usually generated with an iterative process and collected in lists. In order to combine several processes one has to generate values, collect them, operate on the collection, collect again and so forth. Streams operate differently in that they generate values on demand. If several *streams* are combined, they are piped into each other, one *stream* generates as much as the next one demands. Hence, infinite processes can easily be expressed. *Streams*, thus, allow the user to concisely describe networks of dynamic data, by plugging simple parts together.

The persistent nature of streams makes the description of interdependent and also parallel processes easy. Many streams may refer to one and demand values without destructively modifying it or having to copy values. Figure 4.4 demonstrates that a stream always returns the same sequence of values.

```
> (define my-stream1 (st-random-value 1 100))
> (for-example my-stream1)
  (87 97 2 72 64 30 22 91 97 76 27 34 64 72 70 91 72 38 58 98)
> (for-example my-stream1)
  (87 97 2 72 64 30 22 91 97 76 27 34 64 72 70 91 72 38 58 98)
```

Figure 4.4.: Persistent streams always return the same sequence of values

Stream returning functions in *CompScheme* have names starting with the prefix *st-*. The function `st-apply`, for example, takes a function and any number of streams, and returns a stream, whose elements are the results of successively applying the function to the elements of the streams.

Figure 4.5 shows a small example of referring to future values. The defined function converts a stream of absolute starting values, into a stream of pairs of starting values and durations. The durations are the successive time differences between the starting points, so that each event would last until the next one starts. Although this is a small problem, this example demonstrates the expressiveness of the stream approach. In a stream-based system we can simply solve the problem by subtracting the next starting time from the current one, and consequently build a list based on the original value (starting point) and the difference (duration). So, we map the function '`-`' over the stream with one value dropped, thus gaining the 'next' value and the stream itself. Evidently, this also allows for a more refined control of durations in relationship to starting positions of adjacent events, than this brief example shows.

```
(define (calc-durations starting-points)
  (st-apply list starting-points
    (st-apply - (st-drop 1 starting-points)
      starting-points)))
```

Figure 4.5.: Referring to the next value

4.2.1. Finite and Infinite Streams

Most stream constructing functions in *CompScheme* return infinite streams, and it is important for the user to keep the distinction between finite and infinite streams in mind.

Some operations on streams, such as plotting a stream, searching for the minimum or maximum element, accessing the last element, or appending streams require the input streams to be finite. Figure 4.6 shows the definition of an infinite stream and the construction of a stream which contains the first three elements of that stream appended onto the stream itself in its infinite form. The function `st-first n stream` limits the `stream` to the first `n` values, it thus converts an infinite stream into a finite one.

```
> (define my-stream1 (st-sum 1 0))
> (for-example (st-append (st-first 3 my-stream1) my-stream1))
(0 1 2 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16)
```

Figure 4.6.: Appending a finite and an infinite stream

"Inherited Ending"

When building networks of streams, in which streams act as supplies for parameter values of other streams, it is not necessary to limit the topmost stream explicitly, if a stream in the network is already finite. In other words, if any stream in a network of streams is finite, the whole network is finite and contains as many elements as the shortest stream in the network does. The stream of random values constructed in figure 4.7 and displayed in figure 4.8 ends after 10000 elements because the parameter for the lower boundary is a linear shape from 1 to 20, which ends after 10000 elements.

```
> (simple-plot (st-random-value (st-line 10000 1 20) 20))
```

Figure 4.7.: Plotting the implicitly finite stream

4.2.2. Higher-Order Functions and Streams

"The most powerful techniques of functional programming are those that treat functions as data."[Pau96, page 171] *Higher-order* functions or *functionals* are functions, which operate on other functions. In functional programming it is common to abstract by defining functions, which take other functions as arguments. This can help revealing the general structure of a method. In a stream based approach many transformations, filterings, and techniques of creating variation can be expressed in terms of higher-order functions.

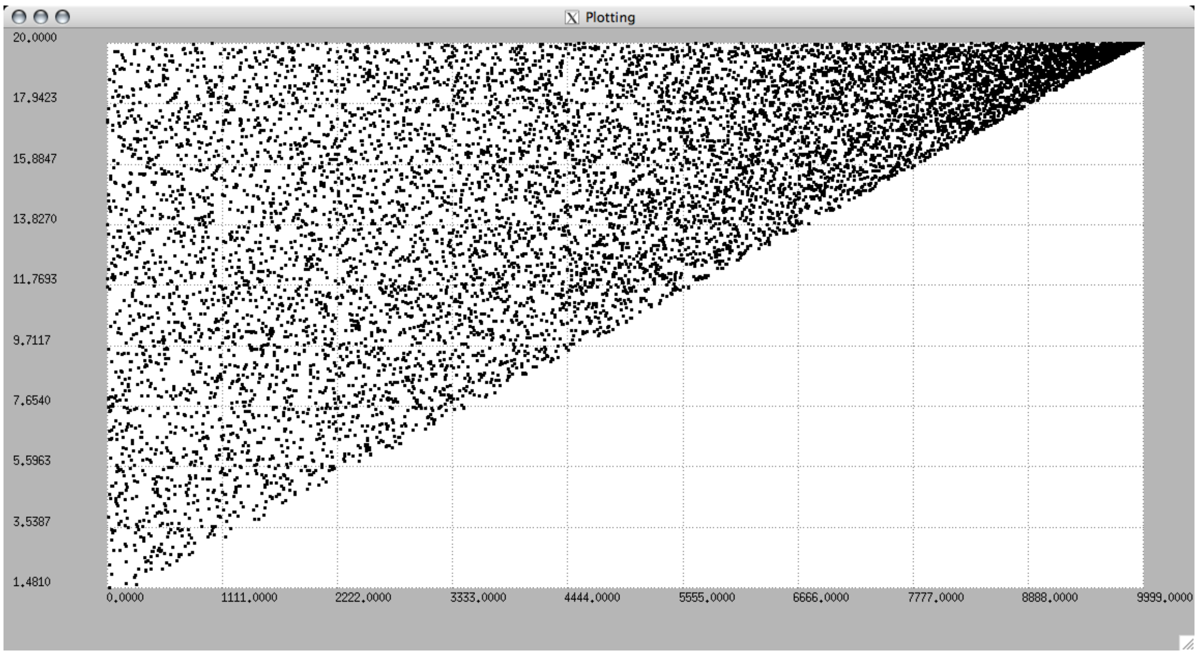


Figure 4.8.: Plotting the implicitly finite stream

Figure 4.9 shows the filtering of a stream of random values. The returned stream only contains those elements of the original stream, which fulfill the predicate $x \; mod \; 12 = 0$. If the numbers were to be interpreted as midi note numbers, the returned stream would only contain the pitch C in different registers. The function `st-filter function stream` returns a stream, whose elements are those of `stream` for which `function` is true.

```
> (for-example (st-filter (lambda (x) (= 0 (modulo x 12))) (st-rv 0 127)))
(48 120 72 96 72 120 60 48 60 72 72 108 24 24 12 24 36 72 36 0)
```

Figure 4.9.: Filtering a stream

Figure 4.10 demonstrates the function `st-apply function` $stream_1 \ldots stream_n$ by creating a triangular random distribution. The function `st-apply` constructs a stream, whose elements are the results of successively applying the function to the streams given. A triangular random distribution can be created by taking the average of two uniformly distributed random values in the same range. In chapter 5, we will see how the function `st-apply` can be used to transform and vary musical structures.

```
(st-apply (lambda (x y) (/ (+ x y) 2)) (st-rv 0.0 1.0) (st-rv 0.0 1.0)))
```

Figure 4.10.: Using `st-apply` to create a triangular random distribution

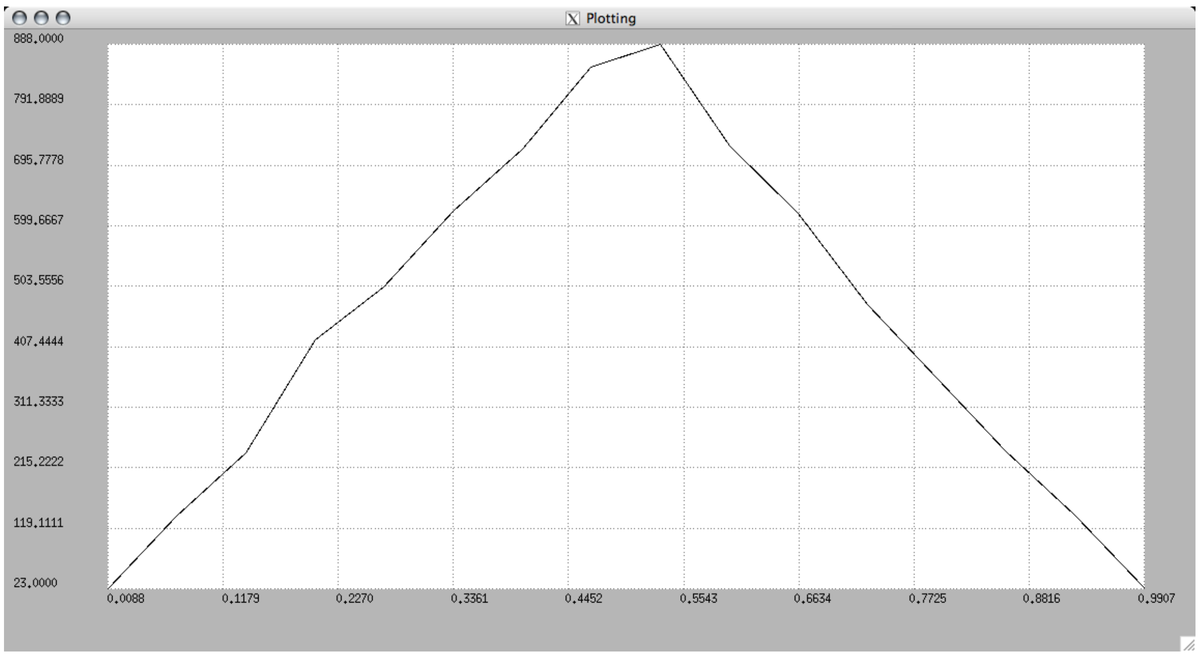


Figure 4.11.: A histogram of the first 10000 elements of the stream from figure 4.10

4.2.3. Defining Streams

Besides using the built-in streams, the user may also define his or her own streams. One way to do this is to define a function for a specific stream network. In this way, we can define a function which returns a stream of triangularly distributed random numbers by naming the stream described above, illustrated in figure 4.12.

```
(define (st-trirnd minium maximum)
  (st-apply (lambda (x y) (/ (+ x y) 2))
    (st-rv minimum maximum) (st-rv minimum maximum)))
```

Figure 4.12.: A simple stream definition

If the desired stream can not be built by combining already available streams, the function `st-cons value continuation` can be used. This function builds a stream, which contains `value` as its first element and `continuation` will be a delayed expression that constitutes the rest of the stream. Figure 4.13 shows the definition of a stream using `st-cons`. The first value is `start` and the continuation will be built by recursively calling `st-product` with the start value multiplied by `factor`, thus creating an exponentially increasing sequence.

However, the definition shown in figure 4.13 has one limitation, being that the factor is constant and can not be controlled using a stream. Figure 4.14 shows an improved

```
(define (st-product start factor)
  (st-cons start (st-product (* start factor) factor)))
```

Figure 4.13.: A stream definition using `st-cons`

version in which the factor can be dynamically controlled using a stream. In order to conform to the “inherited ending” principle, one has to check first if the factor stream is empty. For each iteration the ‘current’ value of the factor stream is accessed using the function `this` and updated for the next call using the `next` function, which returns the stream without the first (i.e. ‘current’) element. For values which are not streams, the functions `this` and `next` act as identity functions.

```
(define (st-product start factor)
  (if (empty-stream? factor)
    empty-stream
    (st-cons start
      (st-product (* start (this factor)) (next factor)))))
```

Figure 4.14.: A stream definition using `st-cons` and a stream as argument

(**this** stream)	Returns the current value of a stream.
(**next** stream)	Returns the continuation of the stream (a new stream).
(**empty-stream?** stream)	Returns true if the stream is empty, otherwise false.
empty-stream	The empty stream (a constant).
(**st-cons** this next)	Returns a new stream, whose first value is 'this' and whose rest is the result of the delayed application of 'next', which is typically another stream returning function.
(**st-of-list** list)	Returns a stream whose elements are those of 'list'.
(**st-to-list** stream)	Returns a list whose elements are those of 'stream'.
(**for-example** stream)	Returns the first 20 values of 'stream' in a list.
(**st-first** n stream)	Returns a finite stream identical to 'stream', but which ends after 'n' elements.
(**st-append** streams...)	Returns a stream constructed by appending the elements of all the given streams, which are typically finite.
(**st-drop** n stream)	Returns a stream identical to 'stream', but without the first 'n' elements.
(**st-filter** function stream)	Returns a stream, which contains those elements of 'stream' for which 'function' returns true.
(**st-apply** function $stream_1 \ldots stream_n$)	Returns a stream whose elements are the results of successively applying 'function' to the streams.

Table 4.1.: Basic stream functions

5. The Event Model

5.1. General

CompScheme's event model provides a framework for building abstract structural units. Musical events are most commonly represented by bundling values for the description of parameters together. In this way, a musical event can be seen as a list of name-value pairs. A name may, for example, be “freq” and its associated value 440. *CompScheme*'s event model provides means of building name-value pairs, by defining *event types.* Event types are name-value pairs which have a name and default values. *CompScheme* has a number of general functions, with which values and names can be accessed and events transformed. Events, however, do not need to be understood only as lower-level musical events, such as notes, messages to a synthesis processes etc., but may as well be representations of higher-level structural units, such as sections, passages, phrases, blocks, or entire pieces. In that sense, *CompScheme* offers a simple, yet powerful event model, enabling the free construction and aggregation of possibly interdependent parametric control structures on multiple temporal and structural levels.

5.1.1. Simple Event Types

Figure 5.1 shows the definition of a simple event type called `myevent1` with three parameters, `start`, `dur`, and `freq`, and default values associated.

```
(defevent myevent1
  (start 0.0)
  (dur 1.0)
  (freq 440.0))
```

Figure 5.1.: Defining a simple event type

Figure 5.2 demonstrates how a stream of events can be created using the function `event-stream`. In this example, the starting values are a series of integers starting with 0.0 and increasing by 1. The frequency parameter is controlled by an exponentially distributed sequence of random numbers. The duration parameter is not controlled and will thus be the default value from the event type definition, i.e. 1.0. It is also possible for the user to provide more parameters than given in the event type definition, the event will then extend automatically and hold the additionally given values too.

```
(define eventstream1
  (event-stream 'myevent1
    (start (st-sum 1 0.0))
    (freq (st-exprand 100.0 1000.0))))
```

Figure 5.2.: Constructing an event stream

In general, it is the user's responsibility to define in which way an event is to be interpreted. The function shown in figure 5.3 can be used to output an event of the defined type in a *Csound* score file syntax with the fixed instrument number 1.

```
(define (print-myevent1 ev)
  (write "1␣")
  (map (lambda (x) (write x) (write "␣"))
    (list
      (event-get 'start ev)
      (event-get 'dur ev)
      (event-get 'freq ev)))
  (newline))
```

Figure 5.3.: Defining a printing function

5.1.2. Higher-Level Events

CompScheme also has a number of built-in event types, such as different midi and *SuperCollider* events. The functions, which output midi or *SuperCollider* events, require streams which contains events that contain at least all of the parameters, which the respective built-in types have. They may, however, contain additional name-value pairs. An event in *CompScheme* is not only to be created in the last instance, as a bundling of values before the data is written out, but may also be a representation of a higher-level structural element, which requires further interpretation. Thus, a hierarchy of events may be created and top-level or intermediate-level events and their interpretation can be created and changed independently.

The event type defined in figure 5.4 stands for a higher-level construct, a section. In this simple example, a section has five properties: a time offset (`offset`), a duration (`dur`), a minimum frequency (`freqmin`), a maximum frequency (`freqmax`), and starting frequency (`freqstart`).

```
(defevent mysection1
  (offset 0.0)
  (dur 10.0)
  (freqstart 100.0)
  (freqmin 100.0)
  (freqmax 10000.0))
```

Figure 5.4.: Defining a simple higher-level event type

However, an event is only given meaning through interpretation. Figure 5.5 shows the definition of a function, which constructs a stream of events of the above defined type `myevent1` with the parameters from a given event of type `mysection1`. The function `st-until` ends the “section” when the start value of the lower-level event stream is greater than the duration specified in the higher-level event.

```
(define (interpret-mysection1 section)
  (let ((offset (event-get 'offset section)))
    (st-until (lambda (event)
                (> (- (event-get 'start event) offset)
                  (event-get 'dur section)))
      (event-stream 'myevent1
        (start (st-sum 0.1 offset))
        (freq (st-walk
                (event-get 'freqstart section)
                (st-rv -200.0 200.0)
                (event-get 'freqmin section)
                (event-get 'freqmax section)))))))
```

Figure 5.5.: Interpreting the defined event

Figure 5.6 shows the creation of twenty sections by creating the higher-level event stream and mapping the interpretation function, defined in figure 5.5, over it. Figure 5.7 shows the frequencies of the twenty created sections.

This simple example demonstrates the elegance and ease with which a number of sections can be produced from a higher-level description. The possibility to define abstract, higher-level structural units enables the composer to establish long-term relationships among sections, phrases, units, or blocks. It also facilitates the algorithmic organization of form concerning decisions.

```
(st-apply
   interpret-mysection1
   (st-first 20
      (event-stream 'mysection1
      (offset (st-sum 10.0 0.0))
      (frestart (st-rv 1000.0 5000.0))
      (freqmin (st-rv 100.0 8000.0))
      (freqmax (st-rv 100.0 8000.0)))))
```

Figure 5.6.: Controlling the higher-level event stream

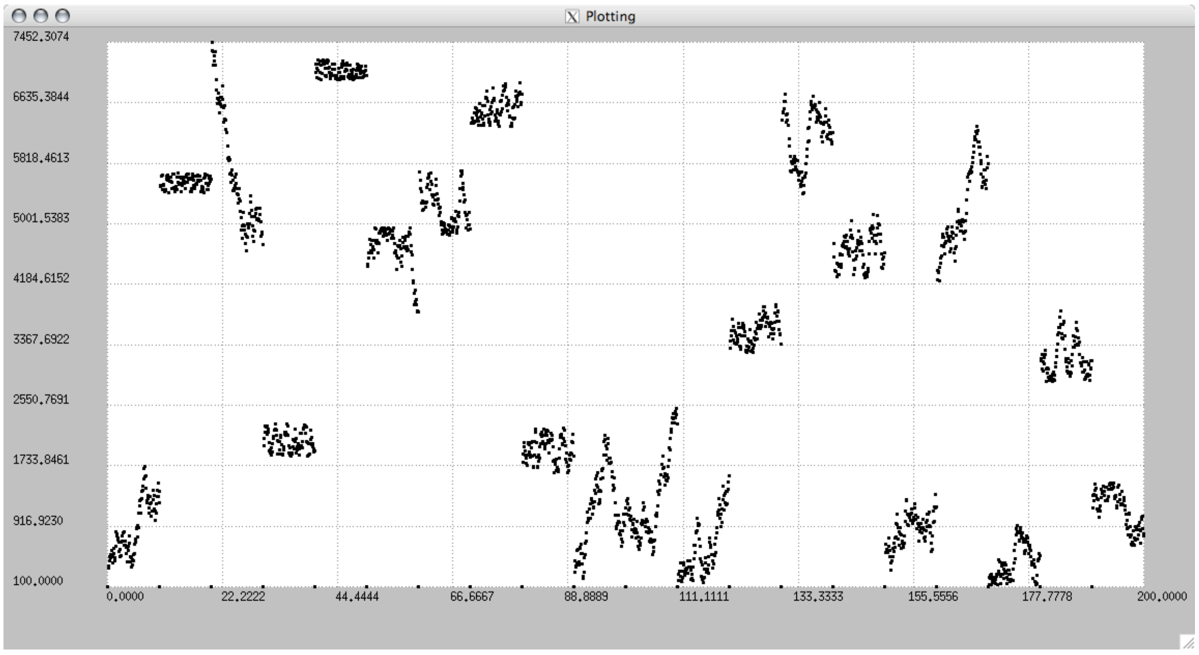


Figure 5.7.: The event's frequencies over their starting points

(**defevent** name (< *par*1 > < *value*1 >) (< *par*2 > < *value*2 >) . . .)	Defines an event type.
(**make-event** type (< *par*1 > < *value*1 >) (< *par*2 > < *value*2 >) . . .)	Creates an event of type 'type'.
(**event-stream** type (< *par*1 > < *value*1 >) (< *par*2 > < *value*2 >) . . .)	Returns a stream of events of type 'type'.
(**event-get** parameter event)	Returns the event's value associated with 'parameter'.
(**event-name** event)	Returns the event's type name as a symbol.
(**event-stream-apply** parameter function stream)	Returns an event stream equal to 'stream', but with the values of 'parameter' replaced by the values returned by mapping 'function' over its values.

Table 5.1.: Basic event functions

5.2. Modeling Structure 1 of Koenig's *Übung für Klavier*

In section 3.2, I have discussed G.M. Koenig's piece *Übung für Klavier*, which was realized using his composition program *Project 2*. In this section, I will show how the first structure of this piece can be modeled in *CompScheme*. Due to the partly incomplete description of the structure it is not possible to regenerate the exact structure, it is, however, possible to come very close to the original, using the available documentation.

As described in section 3.2.1, the basic principle for the construction of musical data in *Project 2* is a three-layered process of entering, grouping, and selecting elements, the so-called *List-Table-Ensemble* principle. In *CompScheme*, I will not directly model this work flow, I will rather divide the construction into three different steps: the definition of user supplied data specific to structure 1 of *Übung für Klavier*, the definition of functions, which model the workings of *Project 2*, especially with regard to entry delay production, and thirdly plugging together the necessary stream functions and the user supplied data to define a function, which returns the structure.

The most characteristic aspect about the first structure of *Übung für Klavier* is the use of masks for the entry delays and dynamic values. Starting very dense, the entry delays gradually get larger towards the end of the structure. As an example, I will show how the entry delays are dealt with in the *CompScheme* model of this structure, the other a parameters are handled similarly. Figure 5.8 shows first the definition of the entry delay list and secondly the definition of two functions, which return the indices of the lower and upper boundaries of the entry delay selecting tendency mask. The parameter `x` is a position within the structure in percent of the total duration. The function `linear-shape x list` linearly interpolates the `list`, which defines line segments in the following format: n_1 $start_1$ end_1 $\ldots n_n$ $start_n$ end_n and returns the interpolated value at position `x` of the specified linear shape.

```
;; list of basic entry delay values
(define *entry-delays*
  '(0.1 0.12 0.15 0.19 0.24 0.30 0.37 0.46 0.58 0.72 0.89 1.11 1.38 1.72))

;; functions, which return the indices of the
;; lower and upper boundaries of
;; the entry delay selecting tendency mask.
;; the parameter 'x' is in percent of the whole structure.
(define (entry-lower x)
  (round (linear-shape x '(20 0 0 27 0 1 20 0 6 33 6 8))))

(define (entry-upper x)
  (round (linear-shape x '(20 0 6 27 4 6 20 7 10 33 10 13))))
```

Figure 5.8.: The user defined masks and basic values for the entry delays

Figure 5.9 shows the definition of the event generating function. As seen above in figure 5.8 the tendency masks are relative, i.e. they do not have a fixed number of elements, but maintain their shape for different specified durations of the structure. The goal is thus to define a function, which takes the durations, i.e. the sum of all entry delays, as an argument, and returns the entry delays, while maintaining the shape of specified tendency masks. In order to do that, the entry delay selecting function needs to know its own output (the "current" time). In *CompScheme*, this can be done by using `st-iterates fun arg`, which returns a streams with the following elements: *arg*, (*fun arg*), (*fun* (*fun arg*)), ...

The duration and the velocity parameter are constructed similarly, but since their values depend on the position in time, i.e. on the entry delays, they do not need `st-iterates`. The generalized selection function for masks in percent is shown in figure 5.10.

Figure 5.11 shows the definition of the final structure generating function. As stated

```
(define (entry-delays duration)
  (st-until (lambda (time) (> time duration))
    (st-iterates (lambda (time)
                   (let ((percent (* 100 (/ time duration))))
                     (+ time
                       (nth *entry-delays*
                         (alea (entry-lower percent) (entry-upper percent)
                            )))))
    0.0)))
```

Figure 5.9.: The entry delay generating function

```
(define (selecting-mask material shape-upper shape-lower start-times
    duration)
  (st-apply
    (lambda (idx) (nth material idx))
    (st-rv
      (st-apply (lambda (time) (shape-lower (* 100 (/ time duration))))
          start-times)
      (st-apply (lambda (time) (shape-upper (* 100 (/ time duration))))
          start-times))))
```

Figure 5.10.: Generalized selection with masks in percent

above, one of the powerful consequences of using relative tendency masks is that the final structure can be stretched and compressed in time. Since the function accepts a duration parameter we can produce structures of any length while maintaining the same development, the same musical gesture. Many other parameters, such as duration, velocity, and register depend on the entry delays, this is why we first define a local variable `start`, which contains the entry delays. As a consequence of the above described persistence of streams, no copying of values is necessary and all streams refer to the same sequence of entry delays, despite the indeterminacy in the process of generation.[1]

5.3. Real-Time Control of the SuperCollider Server

The sound synthesis and music composition programming language *SuperCollider* underwent a major change in its internal design from version 2 to 3. The so-called SC Server, a powerful synthesis engine, and the *SuperCollider* language have been separated into two separate programs and now communicate with the Open Sound Control (OSC) protocol. *SuperCollider*'s system designer James McCartney writes:

[1]Audio examples of different realizations as well as the entire source code can be found on the accompanying CD.

```
(define (structure1 duration)
  (let ((start (entry-delays duration)))
    (st-midi-note
      (start start)
      (duration (selecting-mask *durations*
                  durations-upper durations-lower start duration))
      (velocity (selecting-mask *dynamics*
                  dynamics-upper dynamics-lower start duration))
      (note  (st-apply (lambda (pc mini maxi) (pc-alea pc mini maxi))
               (st-interval-matrix (interval-matrix '(0 4 5 8 9 11))
                 (alea 0 11) (alea 1 11))
               (st-apply (lambda (time)
                           (nth *registers-lower*
                             (registers-lower (* 100 (/ time duration)))))
                 start)
               (st-apply (lambda (time)
                           (nth *registers-upper*
                             (registers-upper (* 100 (/ time duration)))))
                 start))))))
```

Figure 5.11.: The structure generating function

> One goal of separating the synthesis engine and the language in SC Server is to make it possible to explore implementing in other languages the concepts expressed in the SuperCollider language and class library. Some other languages that I think may have interesting potential in the future for computer music are OCaml, Dylan, GOO, and also possibly Ruby[. . .].[McC02]

CompScheme's control possibilities for the SC Server are not designed to be a replacement for the *SuperCollider* language. Synth definitions (SynthDefs), recording, and routing, for example, must still be done in the *SuperCollider* language, but control and instantiation of synths can be done through *CompScheme*. *CompScheme*'s event model for controlling the SC Server is similar to the Pattern classes and the Pbind synth control[2] in the *SuperCollider* language, in that it allows synths to be scheduled with certain parameters. However, ut differs from the Pattern classes in several ways, allowing arbitrarily deep nesting of control streams as the parameters of a synth can be updated within one event, event streams may be directly written into an OSC binary file for non-realtime rendering, and durations and entry delays are always controlled independently. Moreover, the *SuperCollider* event type (SC event) is a regular *CompScheme* event type and can be interpreted, transformed, and written out in numerous ways.

[2]See the *SuperCollider* help files for more information on these classes.

As figure 5.12 shows, *CompScheme*'s *SuperCollider* synth creating event type has three default parameters: the name of the SynthDef, a starting value, which is an entry delay relative to the previous event's starting time, and the duration. It is important, that the synth will free itself, at latest after the time of the duration has passed, because *CompScheme* manages the synth's IDs, in order to be able to update the synth during an event.

```
(st-sc-event
  (synth "sine1")
  (start 1.0)
  (dur 0.5)
  (<parameter4> <value4>)
  (<parameter5> <value5>)
  ...)
```

Figure 5.12.: The `sc_event` stream and its default values

5.3.1. Simple SC Events

Figure 5.13 shows a simple synth definition (synthdef), taken from [Ber07], which is to be evaluated in the *SuperCollider* language. The parameters, which can be controlled with *CompScheme*, are the arguments of the synthdef: freq, amp, dur, attack, decay.

```
(
SynthDef("sine1",{ arg freq = 440, amp = 0.2, dur = 2.0, attack = 0.25,
   decay = 0.25;
  var   ssTime = dur * (1 - attack - decay);
  var    attackTime = dur * attack;
  var    decayTime = dur * decay;
  Out.ar(0,
          SinOsc.ar(freq, 0,amp)
               * EnvGen.kr(Env.linen(attackTime,ssTime, decayTime,1),
                   doneAction: 2)
  )
}).store;
)
```

Figure 5.13.: A simple synth definition (taken from [Ber07])

Figure 5.14 demonstrates how to play a SC event stream in real-time. This example also demonstrates the advantage of persistent streams and the power of higher-order functions. In contrast to midi event streams, SC event streams work with relative entry delays, not absolute starting times. This decision has been made to ensure sensible time values for real-time output. In the example shown in figure 5.14 the starting times are made by a random choice from a list of four values. The value 0.0 stands for simultaneous events (chords). The events last until the next event starts, which is made by using the entry delays of the start parameter and dropping the first value. There is, however, one problem. Due to the chords, events which are followed by simultaneous events will have a duration of 0.0 seconds. In order to ensure that all events last at least 0.1 seconds, a clipping function is applied to the duration stream.

```
(sc-play
  (let ((entry-delays (st-random-choice '(0.0 0.1 0.15 1.7))))
    (st-sc-event
      (synth "sine1")
      (start entry-delays)
      (dur (st-apply (lambda (x) (max x 0.1))
             (st-drop 1 entry-delays)))
      (freq (st-exprand 100.0 4000.0)))))
```

Figure 5.14.: Playing a SC event stream

5.3.2. Sub-Events

As stated above, one of the powers of *CompScheme*'s SC event type system is that events, which instantiate synths, can update the synth during an event. This means that events can not only represent note-like sound events, but also control updates, within such a sound event. In general, this mechanism works by not supplying a static value or a stream of numbers, but by supplying a *stream of streams*. Every stream in this stream of streams is then seen as a development the parameter has during the respective event. However, the streams inside must be of a certain type, namely `sc_nset`. Figure 5.15 shows the definition of two auxiliary functions for the creation of a sub-event stream. The first function returns a stream of `st-sum` streams. The second function returns the stream of `nset`-streams we will use in the final output. The `nset` event type holds two values, `start`, which is a starting value relative to the starting point and duration of the parent event, where 0.0 denotes the starting point and 1.0 the ending of the parent event, and `value` which is the respective value used for the update of the synth's parameter. The defined function `stream-nsets` takes three arguments, which will be streams, the number of elements for each sub-event stream, the starting points and the values themselves, which are assumed to be streams of streams.

Figure 5.16 finally shows how the an `nset`-stream can be embedded. In the example,

```
 ;; a stream of streams
(define (sum-streams add start) (st-apply st-sum add start))

;; a stream of nset streams
(define (stream-nsets st-n st-start st-value)
  (st-apply
    (lambda (nstr strt vls)
      (st-first nstr (st-sc-nset (start strt) (value vls))))
    st-n st-start st-value))
```

Figure 5.15.: Defining auxiliary functions for sub-events

we create a simple SC event stream, but use the above defined function for the creating a stream of `nset`-streams to control the frequency parameter. The duration of the update streams will be randomly selected between 2 and 5, the starting points of the updates are generated by streams of streams, which all start at 0.0 and increment by a constant addition of a randomly generated value for each event between 0.05 and 0.2. The frequency of each event will thus always start at 800 Hz. The defined function takes the frequency increment per sub-event as an argument, here we call the function with a constant of 100 Hz.

```
(define (sc-nset-stream1 freqadd)
  (st-sc-event
    (start 2.0)
    (dur 2.0)
    (freq (stream-nsets
           (st-rv 2 5)
           (sum-streams (st-rv 0.05 0.2) 0.0)
           (sum-streams freqadd 800.0))))))

(sc-play (sc-nset-stream1 100.0))
```

Figure 5.16.: Defining and playing a stream with an embedded nset stream

5.3.3. Scheduling Event Streams

It is not only possible to extend the event model to lower levels, as described in the previous section, but also to extend it to higher levels. SC event streams themselves can also be scheduled. There is another type of event called `sc_stream_event`, which contains SC event streams and starting times as relative entry delays. In the example in figure 5.17 a stream of SC event streams is build by mapping the SC event stream returning function `sc-nset-stream1` defined in figure 5.16 over a stream of random values, which will be interpreted as frequency increments for the sub-event (see previous section). The function `st-sc-stream` schedules the stream of SC event streams, the entry delays are given by the `start` argument. The function `st-sc-stream` can also take further `st-sc-stream`'s. Therefore, there is no built-in limit and scheduled event streams can be scheduled again.

```
(sc-play-stream
  (st-sc-stream
    (start (st-rv 0.0 2.0))
    (sc-stream (st-apply sc-nset-stream1 (st-rv 10 100)))))
```

Figure 5.17.: Playing a stream of event streams

(**connect-to-sc** port)	Opens a connection to the *SuperCollider* server over the specified port (usually 57110).
(**disconnect-from-sc**)	Closes the connection to the *SuperCollider* server.
(**st-sc-event** (synth $< value1 >$) (start $< value2 >$) (dur $< value3 >$) ($< par4 > < value4 >$) ($< par5 > < value5 >$) ...)	Creates a SC synth event stream.
(**st-sc-nset** (start $< value1 >$) (value $< value2 >$)	Returns a stream of update values to be used inside of **st-sc-event**. The starting values should range from 0.0 to 1.0, where 0.0 stands for the beginning of the event and 1.0 for its end.
(**sc-play** sc_event-stream)	Plays an sc_event-stream in real-time.

Table 5.2.: Basic event functions

6. Stochastic Synthesis in CompScheme

6.1. Introduction

The idea to synthesize sound directly by using musical procedures has been employed by composers of electronic music at least since the early 1950s. Extending the compositional control down to the micro-level, and thus being able to actually compose the sound itself, has not only been part of the basic postulate of the Köln electronic music school, but has also been a general thought in many approaches to computer generated sound until today.

In the 1970s the composers Gottfried Michael Koenig, Iannis Xenakis, Herbert Brün, and others developed systems that abandoned existing acoustic models, and tried to derive sound synthesis methods directly from compositional activities. Rather than trying to compose with sounds created on the basis of given analytical models, the sound is supposed to be the result of the compositional process itself. In 1970 G.M. Koenig described his program SSP, which was not yet implemented at that time:

> As opposed to programmes based on stationary spectra or familiar types of sounds, the composer will be able to construct the waveform from amplitude and time-values. The sound will thus be the result of a compositional process, as is otherwise the structure made up of sounds. [Koe70b]

With SSP, Koenig extended the principles used in his earlier programs Project 1 and specifically Project 2 from the level of the note down to the level of the digital sample. As basic elements amplitude and time values were specified and grouped in segments, in which they were linearly interpolated. For the selection of the basic elements, aleatoric and serial principles were used. SSP may be seen as an attempt to overcome traditional ways of representation that stem from instrumental music, and substitute them with more general descriptions, such as similarity, transition, and variation that are to be applied to the macro-structure of the form as well as to the micro-structure of the sound in one process. This is derived from the axiomatic assumption, that "musical sounds may be described as a function of amplitude over time."[Koe70b]

Iannis Xenakis's idea of dynamic stochastic synthesis differs from Koenig's SSP in its initial intentions. The notion of an evolutionary process is central to Xenakis's idea of dynamic stochastic synthesis. In dynamic stochastic synthesis, breakpoints are grouped – here in cycles of a waveform – and linearly interpolated to form an integration of macro- and micro-levels of musical time. Both approaches to stochastic sound synthesis are primarily rooted in music composition, derived from compositional activities and not in the analysis of sound.

> Any theory or solution given on one level can be assigned to the solution of problems of another level. Thus the solutions in macro-composition (programmed stochastic mechanisms) can engender simpler and more powerful new perspectives in the shaping of micro-sounds. [Xen92]

Xenakis, Koenig, and Herbert Brün were motivated by finding ways of producing sound that are idiomatic to the means of production, the computer. Instead of emulating an instrumental or electronic paradigm, the idea of the sample as the basic musical element is inherently digital. Xenakis, Koenig, and Brün used the sample as the basic musical element in a search for "sounds that had never before existed"[Xen92]. Instead of the novelty of sound, the strength of this non-standard approach to sound synthesis lies in its unification of the sound production and compositional processes. It is therefore really one of representation.

In the following, I present a program that is not aimed at reimplementing, but rather an attempt to generalize from Xenakis's and Koenig's systems for stochastic sound synthesis and thus providing the possibilities for extensions. I try to show, that the flexibility and expressiveness of streams lends itself well not only to the description of higher-level compositional processes, but as well to the lower-level sound production. Stochastic sound synthesis is an area of application in which a basic motivation of electronic music, namely *composing sound*, demands a unified representation. This unification of the sound production and the composition process requires a pervious relationship between sound and control data. However, most current sound synthesis systems and computer music languages establish a strict separation of synthesis and control data. There are, therefore, hardly any platforms today, that enable experimentation in this area.

6.2. Synthesis in CompScheme

In *CompScheme*, rather than considering sound synthesis and composition as two different domains, the same mechanisms are used to describe sound as well as higher-level control. There is no separating wall between sound and control built into the system and no limit to the level of abstraction.

6.2.1. A Generalization of Stochastic Synthesis

As described in section 6.1, both SSP as well as Xenakis's systems group amplitude and time points together, form sequences of these groups, and linearly interpolate the breakpoints. In the case of SSP, these groups – called segments – contain elements selected from initial amplitude and time lists by using Koenig's *selection principles*. In Xenakis's systems, these groups are cycles of one waveform, whose elements are a deviation from the previous cycle's elements, using stochastic processes.

In *CompScheme*, the basic sound synthesis element is the sample, which contains both a time and an amplitude value. A sample is considered an event, just like any other musical event, and can be built and transformed with the same mechanism. Figure 6.2 shows the function `st-sample` which uses the event type syntax shown in figure 6.1.

```
(make-event <name>
  (<parameter1> <value>)
  (<parameter2> <value>)
  etc...)
```

Figure 6.1.: Event stream creation

The example in figure 6.2 creates a stream of sample events from two streams, one determining the positions of the breakpoints and one that determines their amplitude. The positions in this example are taken from a list of four integers and the amplitudes are chosen randomly between -1.0 and 1.0.

```
(st-sample
  (pos (st-of-list '(0 1 2 5)))
  (value (st-random-value -1.0 1.0)))
```

Figure 6.2.: Sample stream

Based on SSP, we may call the sample stream of figure 6.2 a *segment*. A sample's time value denotes its position within the segment to which it belongs. Segments are then collected in a stream – a stream of sample streams – which can be interpolated with an interpolation function and written out into an audio file. A segment can thus be seen as cycle in a process of dynamic stochastic synthesis, or as segment in a collection from which we can select, using a *selection principle*.

6.2.2. Example 1: Dynamic Stochastic Synthesis

For a concrete example, we turn towards implementing a simple process close to Xenakis's GENDY. Generally speaking, in GENDY several breakpoints are defined and interpolated in what could be called one cycle of a waveform. The next cycle is a deviation of the previous one. Each breakpoint and time distance follows a random walk and the total length of each cycle is also controlled.

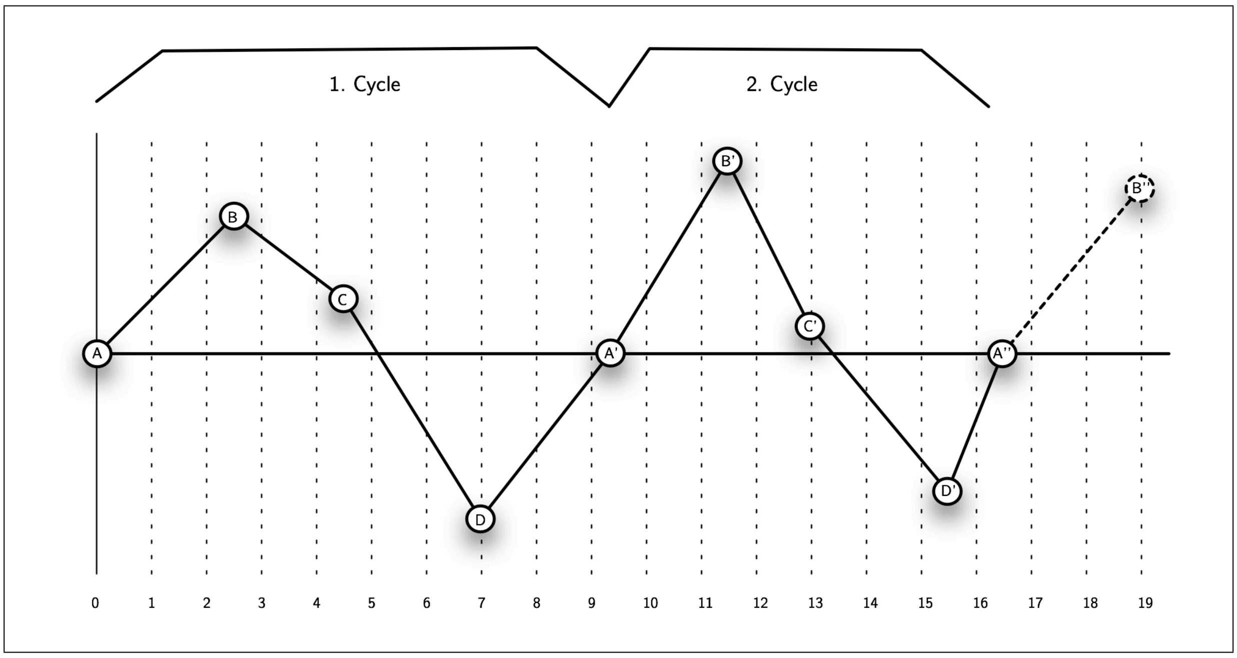


Figure 6.3.: Two cycles

This process can easily be described in *CompScheme* using the function `segments-with-length`, which takes three arguments: the length of the cycle and two lists of numbers, the first one containing a value for each position and the latter one for each amplitude value. The time values are then scaled to fit inside of the specified length. It then returns a stream of samples; here to be considered one cycle.

Figure 6.4 shows the definition of a function called `gendy1`, that describes a GENDY-like process, which is kept simple for the sake of brevity. The function `segments-with-length` is successively applied to the elements of the three argument

```
(define (gendy1)
  (st-apply segments-with-length
    (st-walk 80.0 (st-rv -10.0 10.0) 50.0 250.0)
    (st-apply list
      (st-walk 15.0 (st-rv -2.0 2.0) 5.0 20.0)
      (st-walk 15.0 (st-rv -2.0 2.0) 5.0 20.0)
      (st-walk 15.0 (st-rv -2.0 2.0) 5.0 20.0)
      (st-walk 15.0 (st-rv -2.0 2.0) 5.0 20.0))
    (st-apply list
      (st-walk 0.0 (st-rv -0.1 0.1) -1.0 1.0)
      (st-walk 0.0 (st-rv -0.1 0.1) -1.0 1.0)
      (st-walk 0.0 (st-rv -0.1 0.1) -1.0 1.0)
      (st-walk 0.0 (st-rv -0.1 0.1) -1.0 1.0))))
```

Figure 6.4.: Definition of `gendy1`

streams. The first argument controls the lengths of the cycles, by means of a random walk (`st-walk`) starting with 80, successively adding the elements of the inner stream of random values between -10 and 10 onto its current value, and limited in borders ranging from 50 to 250. The second and third arguments determine the breakpoints's positions within the cycle and are also controlled by random walks. For the sake of brevity, only four breakpoints are made. Before `segments-with-length` is applied, the time points, as well as the amplitude values, are collected in lists. It is to be mentioned, that the returned stream of waveform cycles is infinite. Since the output is a stream, we have not left the high-level description and can easily transform and reuse the created cycles.

Figure 6.5 shows how to write out the first 10000 cycles of a sample stream into an audio file, using a sample rate of 44100 samples per second.

```
(write-sample-stream "gendy1.wav" 44100
   (st-first 10000 (gendy1)))
```

Figure 6.5.: Writing out an interpolation into an audio file

One possible extension of GENDY that composer Sergio Luque proposed [Luq06] is the concatenation of several independent GENDYs. Similar to SSP's *permutation* function in which segments are concatenated by using *selection principles*, Luque concatenates waveform cycles from several independent GENDYs. We could easily concatenate several `gendy1`s with the function `st-interleave`, which interleaves the output of any number of streams and forms a new stream as shown in figure 6.6.

Usually stochastic synthesis is implemented in a form, that makes the positions of breakpoints depending on the sample grid. That means, that a breakpoint can only be

```
(st-interleave (gendy1) (gendy1) (gendy1))
```

Figure 6.6.: Concatenating three independent cycle streams

set at a sample point. A consequence of restricting the positioning of the breakpoints to sample points is that one can only express cycles of durations, which are an integer multiple of the duration of one sample in the chosen sample rate. This limitation imposes a strong frequency grid, which is especially audible with high frequencies. A restriction like this would be considered intolerable in the case of standard oscillators, but it has been often neglected in the discussion of dynamic stochastic synthesis, in favor of reimplementing truthful adaptations of its historic original, including all of its idiosyncrasies. As can be seen in figure 6.4, the breakpoints's locations are expressed in floats. That means, they can be located in between samples, the resulting wave is then 'sampled' again during the interpolation process.

6.2.3. Example 2: A Variation on SSP

The following example demonstrates the use of higher-order functions to create variations of streams. In SSP, one defines segments and then selects an order of the defined segments with a function called *permutation*. In *CompScheme* there is a function called `segment` that takes three arguments: the length of the segment, a stream of relative time distances, and a stream of amplitude values and returns a stream of samples. Figure 6.7 shows the creation of a stream of variations of segments. The described function `segment` is mapped over three other streams, the first one producing the lengths, the second one a stream of streams produced by varying a stream, and the last argument is a stream of amplitude value streams. This last stream of streams is again produced by a mapping of a function, namely `st-repeat`, which takes two streams one containing the number of repetitions and the other containing the values to be repeated. Thus the variable `segments` contains an infinite stream of segments. Whereas in SSP every segment has to be created 'by hand', here we can easily employ SSP's principles on a higher level and create possibly infinite streams of segments.

In order to create a *permutation*, we can select segments from the above defined stream by using another stream. Figure 6.8 shows a possible permutation. Three thousand segments are selected from the above defined `segments` with a tendency mask going from between 0 and 0 to 40 and 60 and an indexing function. Since the deviation among the first elements is smaller then that among the later ones, the output develops from a

```
(define segments
  (st-apply segment
    (st-random-value 5 50)
    (st-apply st-random-value 1 15)
    (st-apply st-repeat
      (st-apply st-random-value 1 10)
      (st-apply st-random-value -1.0 1.0))))
```

Figure 6.7.: segments

rather pitched sound to something more noisy.[1]

```
(st-nth segments (st-tendency 3000 0 0 40 60))
```

Figure 6.8.: a permutation

[1]Sound examples and source code can be found on the accompanying CD.

Part III.

Music

7. *Bellavista I, II, and III*

Figure 7.1.: *Piz Palü*

7.1. Overview

The three pieces, *Piz Palü*, *Piz Zupò* and *Piz Bernina*, which I discuss in this chapter, all belong to a series. *Piz Palü* and *Piz Zupò* are pieces for fixed-medium and *Piz Bernina* is a piece for solo piano. All three pieces have been realized using my program *CompScheme*, which has been described in the previous part of this text.

7.2. *Piz Palü* and *Piz Zupò*

7.2.1. General

Both *Piz Palü* and *Piz Zupò* are produced using one and the same sound synthesis method, which is based on Iannis Xenakis's dynamic stochastic synthesis. Furthermore, in both pieces spectra of musical sounds serve as basic material. But rather than using them to recreate a recognizable effigy of the original or to establish a semantic relation to their sources, they serve as points of departure, arrival and reference. During the course of the pieces these spectral materials are juxtaposed and transformed. The spectra thus serve as frequency an amplitude collections of specific characters. The pieces have been realized using *SuperCollider* as a synthesis engine, using primarily one plugin unit generator, which I developed, whose general workings are described in section 7.2.2. The higher-level control has been realized with the help of my program *CompScheme*.

7.2.2. Synthesis

The sound synthesis method used in *Piz Palü* and *Piz Zupò* is based on Iannis Xenakis's dynamic stochastic synthesis. In this section, I explain the differences of my implementation to Xenakis's model, for an introduction to dynamic stochastic synthesis the reader may refer to [Hof00] or [Luq06]. Rather than trying to reimplement one of Xenakis's models, I have tried to find a generalization and extend the model, most significantly in two points:

- the positioning of breakpoints independent of the sample grid
- the possibility to allow any signal to control the main parameters

Figure 7.2 shows two cycles of a stochastic synthesis process as it is commonly implemented. In this example a cycle consists of four linearly interpolated breakpoints,

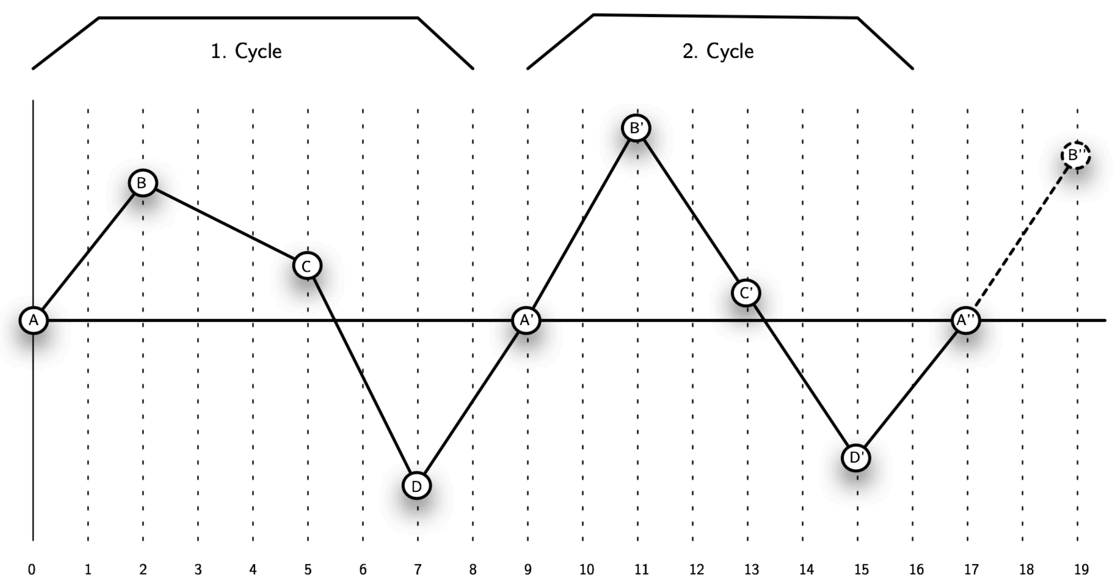


Figure 7.2.: Stochastic Synthesis as commonly implemented

denoted by the letters A,B,C and D. Each of these breakpoints follows its own, independent path. In the next cycle the vertical and horizontal positions are the result of deviations of their previous states. The breakpoints in the next cycle are denoted by the letters A',B',C' and D'.

As can be seen in Figure 7.2, the first cycle has a length of exactly 9 samples and the second cycle one of exactly 8 samples. In this model the breakpoints are always located on a sample, i.e. they are positioned inside the sample grid, indicated here by the vertical dotted lines. A consequence of restricting the positions of the breakpoints to coincide with sample points, is that one can only express cycles having durations which are integer multiples of the duration of one sample in the chosen sample rate. With a sample rate of 44100, this would mean all frequencies that can be expressed are $44100/i$, where i is a positive integer greater or equal to 2.

Figure 7.3 illustrates my implementation of stochastic synthesis with respect to the positioning of breakpoints in time. As can be seen, the breakpoints can be located in between samples, the resulting wave is then 'sampled' again. It is thus possible to create cycles whose lengths correspond to any value (within the usual constraints of digital oscillators) and may change quasi-continuously in length.

My implementation of stochastic synthesis is a generalization of the usual implementation. Instead of trying to model the original truthfully, I have tried to leave as many

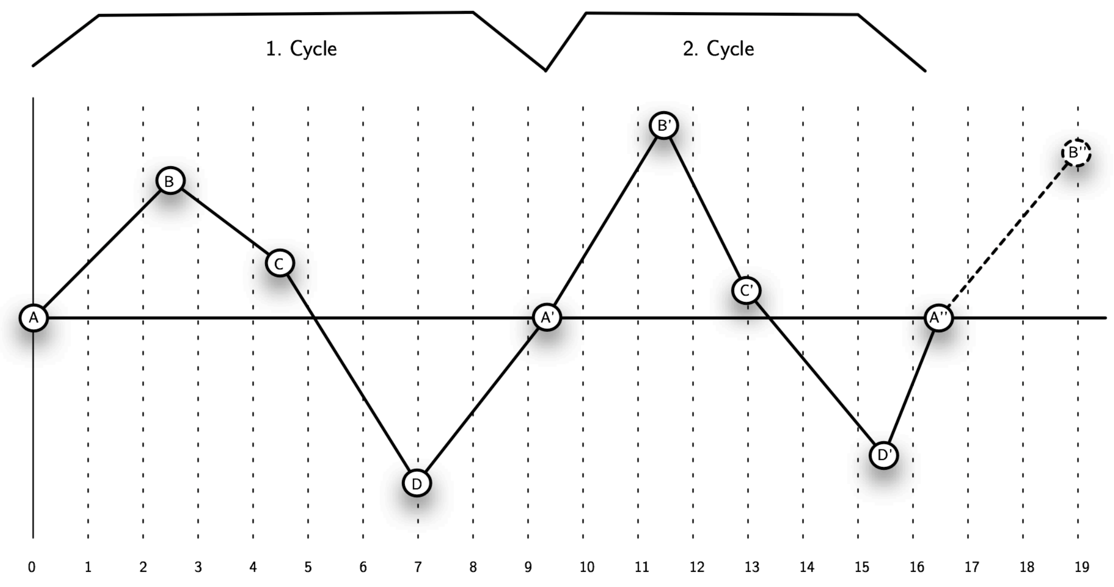


Figure 7.3.: Positioning of breakpoints in between samples

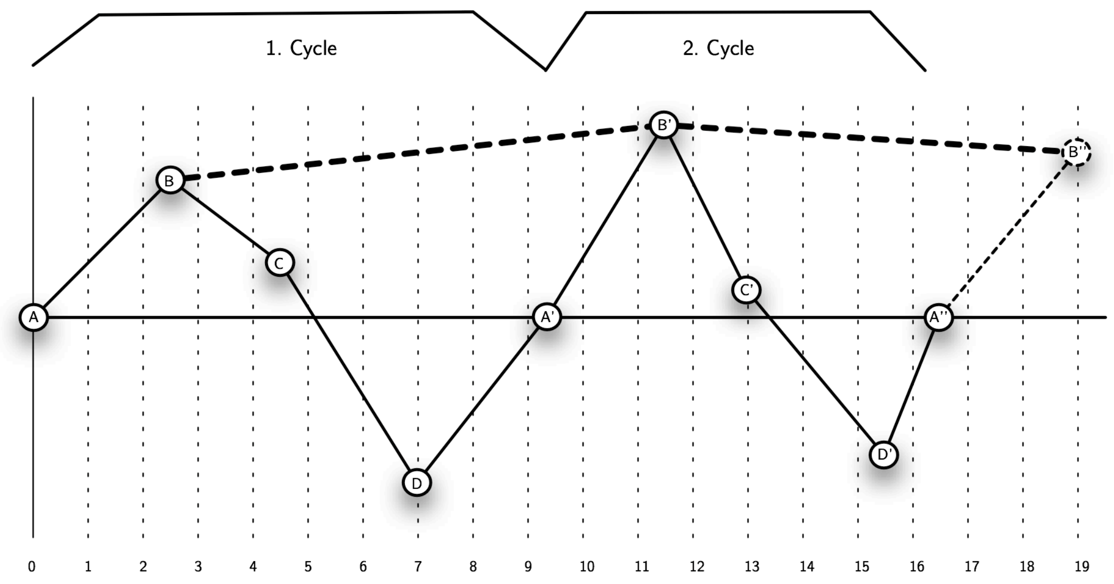


Figure 7.4.: Highlighting the course of B's amplitude development

points as possible open to the user. The course, which the amplitude of each of the breakpoints follow, can be controlled with any input signal. In figure 7.4, the course of one of the breakpoints, B, is highlighted by the bold dotted line.

7.2.3. *Piz Palü*

Piz Palü is a 4-channel composition for fixed medium (tape). The piece consists of streams of events which unfold in a dendritic structure. There are 4 distinct spectra of musical sounds used as basic material. The nature of the spectra itself is here less important than the fact and ways in which they differ from each. The spectra, which are treated as frequency sets create reference points for the organization of the piece. It is impossible for the listener to recognize the original source of a spectrum, it is, however, possible to distinguish the sets from each other. The four spectra used in *Piz Paü* and the ten spectra used in *Piz Zupó* are thus not random in nature, as they transport characteristics of the original sources from which they stem without ever representing them in a recognizable manner.

Synthesis in *Piz Palü*

In *Piz Palü* I have chosen not to use – as commonly done – random walks or second order random walks to control the positioning of breakpoints. Instead, the positioning is controlled by other 'higher-order' stochastic synthesis processes. Thus, there are three levels in the sound producing process:

1. tendency masks, static values, linear and exponential shapes controlling the parameters of
2. multiple stochastic synthesis processes controlling the parameters of
3. one audible stochastic synthesis process

At the core of the concept of stochastic synthesis there is the idea of unifying the macro-structure and the micro-structure in composition and to use procedures which have been successfully employed on a higher level for the synthesis of sound. *Piz Palü* can be seen as an attempt to reverse this application and use signals produced by a stochastic synthesis algorithm – originally intended to be audio signals – as parametric control signals. Stochastic synthesis processes thus occur on two levels in *Piz Palü*, as control signals and as audio signals. Moreover, the spatial distribution of the sounds is again controlled by a stochastic synthesis process, which is then interpreted as a moving position among the four speakers.

Structure and Event Streams

Piz Palü can be thought of as a number of streams of events, which carry certain parameters, creating collections of sounds of different densities, spectral characteristics, rhythmic coherence and dynamic movement. From event to event a stream can either produce a directed variation of its previous event (state), branch out, that is create a 'sibling', die out or merge with another stream. Figure 7.5 shows the four stream operations:

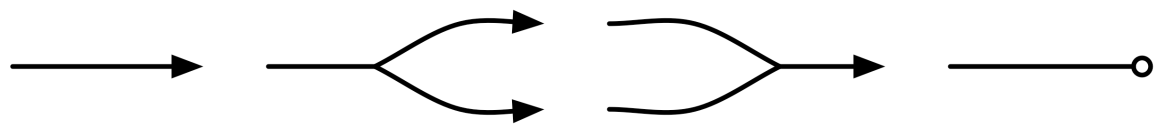


Figure 7.5.: continue, split, merge, or die

Before the actual structure of the event streams was created, a number of fixed points were determined. These eight points, or constellations, are targets to be reached, found by an empirical investigation and by a phase of exploring the possibilities of the model. After the targets were determined, decision points were fixed in time. These are points in which the above mentioned operations take place. The selections of the operations are done by constrained probabilities changing over time. *Piz Palü* can be seen as a number of multiply self-referential processes enfolded on different time levels, creating variants of a single event which unfold in the course of the piece.

Figure 7.6 shows the arborescent event stream structure of the first 110 seconds of the piece.

7.2.4. *Piz Zupò*

Piz Zupò is a 2-channel composition for fixed-medium. In contrast to *Piz Palü*, where the structure of the piece is determined by streams of events, where a continuous evolution is formed, the structure of *Piz Zupò* is built by dividing the piece into sections. There is a common "structure formula" (G.M. Koenig) to all of the ten sections, i.e. a higher-level description of what constitutes a section is.

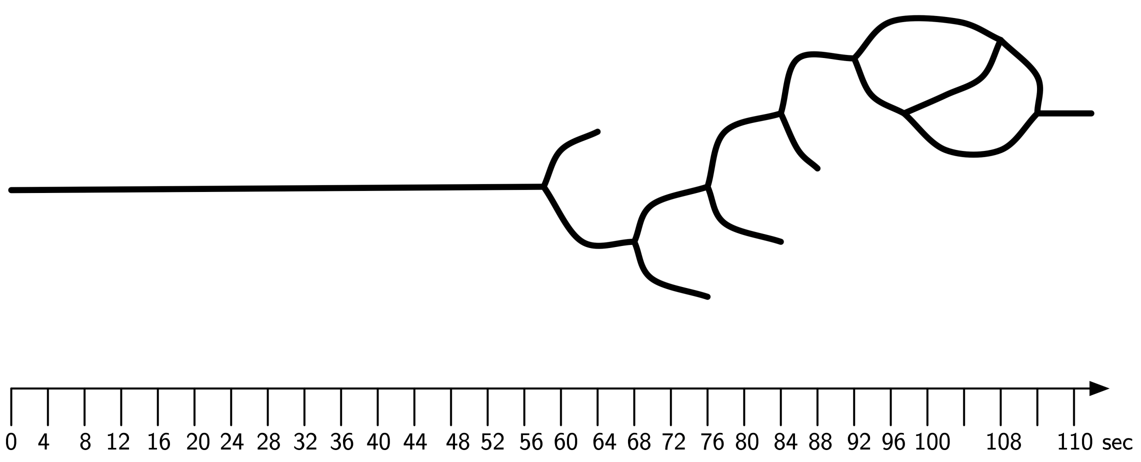

0 4 8 12 16 20 24 28 32 36 40 44 48 52 56 60 64 68 72 76 80 84 88 92 96 100 108 110 sec

Figure 7.6.: Event streams structure of the first 110 seconds of *Piz Palü*

The composition of music, as well as most other structured thought processes, starts by making fundamental distinctions, by discrimination. In *Piz Zupò*, I have made three fundamental distinctions: directionality versus non-directionality, pitch versus noise, and a hierarchy of distinct *levels of description.*

The basic element is an *event*, which, besides several synthesis parameters, has a start time and a duration. Secondly, there are *sections*, which are directional and non-directional streams of events. Thirdly there are directional and non-directional *streams of sections*, constituting the highest level of description. Using the *level of description* as a parameter was central to the production of *Piz Zupò.* Table 7.7 shows the ten sections of *Piz Zupò.* The first column shows the number of the section, the second its duration in seconds, and the third column shows if it is a directed or a non-directed section, "directed" means that an event can only occur at a specific moment in the section, allowing for a change of state where the determining parameters alter throughout the course of the section. The fourth column shows the 'type' of the section, which means the level of description. 'A' stands for streams of events, 'B' stands for streams of sections, and 'C' stands for variations of single events. The fifth column shows the rhythm generating method used. There are two methods which can be seen in figure 7.8 and 7.9. Rhythm type 'T' are rhythmic values generated by a transition table and rhythm type 'S' are rhythmic values generated by exponential shapes. Type 'T' tends to generate more combinatorial structures, whereas type 'S' has a more long term state and generates gestures of speed alterations. The sixth column shows the basic frequency sets used, which can be seen in figure 7.7. The seventh column indicates the presence of dynamically changing frequency values ("glissandi").

The 8th column shows the average "freeze" value. In my implementation of stochastic synthesis, the process can be frozen, enabling direct repetition of cycles and leading to clearly pitched sounds. Thus, the values in the seventh indicate the average "pitchedness" of the sounds. However, for some sections, such as section 4 and 7 the deviations are large, and even section 3 and 5 with high "freeze" values, contain single events with low "freeze" values. Finally, the last column shows the number of simultaneous streams, or "voices", present in each of the sections.

The piece can be roughly divided into three parts, section 1, 2, and 3 form the first part, which is characterized by distinct events of pitched quality, section 4, 5, and 6 form the middle part, which contains more complex higher-level gestures and introduces non-directionality, and section 7, 8, 9, and 10 contain blocks of noisier textures.

The two channels are two renderings of the same event description, the difference only occurs due to the indeterminacy of the synthesis process.

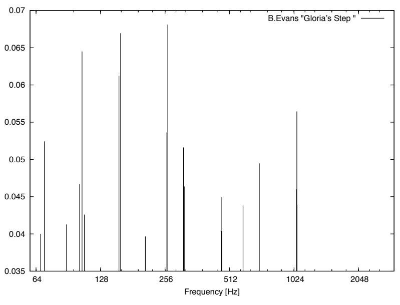
Nr	**dur**	**dir**	**type**	**rhythm**	**Freq. sets**	**Gliss.**	**aver. freeze**	**N Str.**
1	60	Y	A	T	5,6,7	N	$0.999 \rightarrow 0.15$	3
2	115	Y	A	S	5,6,7	N	0.999	2
3	30	Y	A	T	5,6,7	N	$0.9 \rightarrow 0.97$	3
4	60	N	B	S	1,2,3,5,6,7	Y	0.67	4
5	50	N	B	T	1,2,3	Y	0.995	3
6	120	Y	C	T	4,9,10	N	0.48	3
7	20	Y	C	-	6	Y	0.63	1
8	45	N	C	-	6	N	0.0	1
9	45	N	C	-	6	N	0.0	2
10	25	N	C	-	6	Y	0.0	3

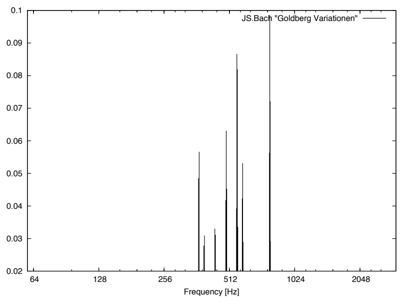
Table 7.1.: The 10 sections of *Piz Zupò*

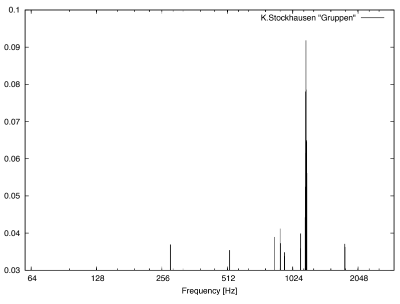
7.3. *Piz Bernina*

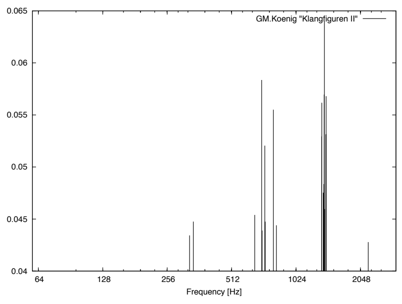
Piz Bernina is, on a general level, an application of the structure of *Piz Palü* onto a different material. Despite the difference of the material, *Piz Bernina* is also composed of "branching out" event streams, as was described in section 7.2.3.

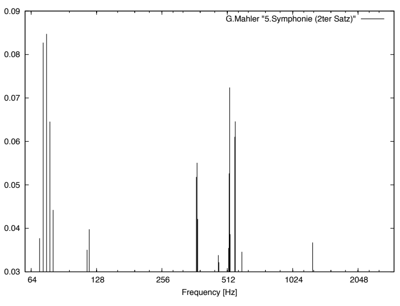
The pitch material is composed of four different pitch sets. These pitch sets were generated by a *Prolog* program, which performed a search according to a number of

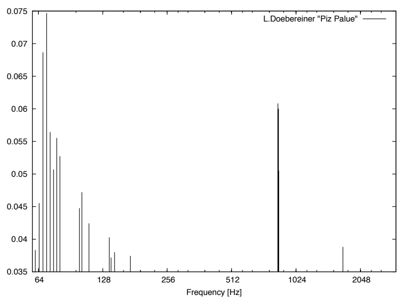
1)


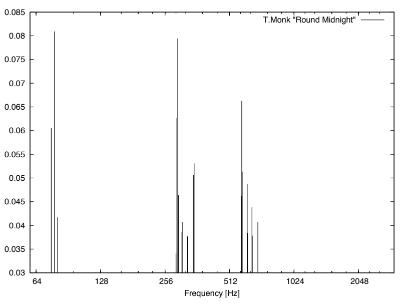
2)


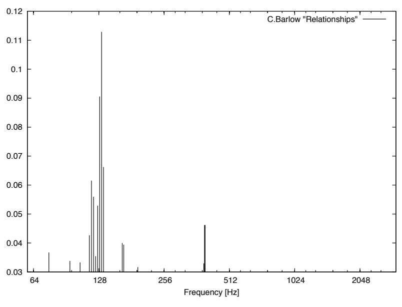
3)


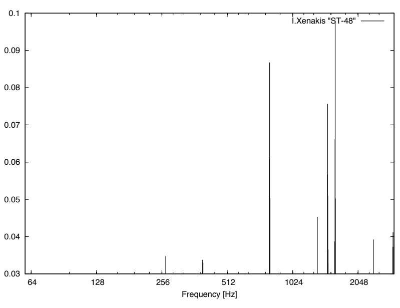
4)


5)


6)


7)


8)


9)


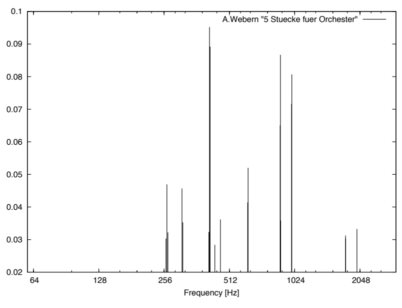
10)


Figure 7.7.: The ten frequency sets used in *Piz Zupò*

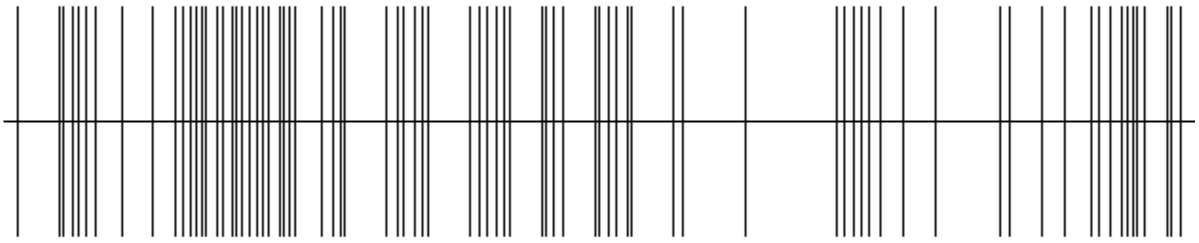


Figure 7.8.: Rhythmic values generated by a transition table (type T)

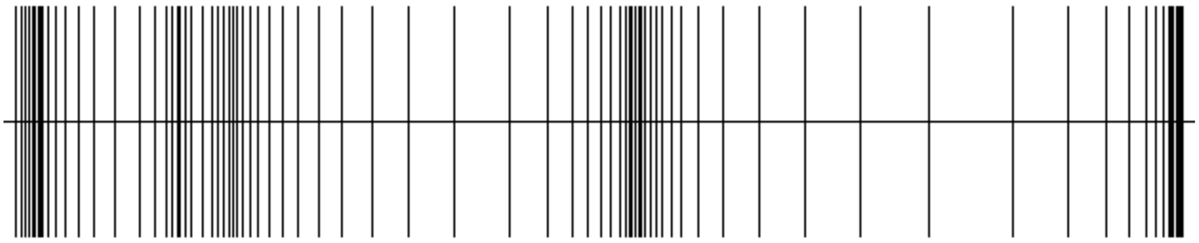


Figure 7.9.: Rhythmic values generated by exponential shapes (type S)

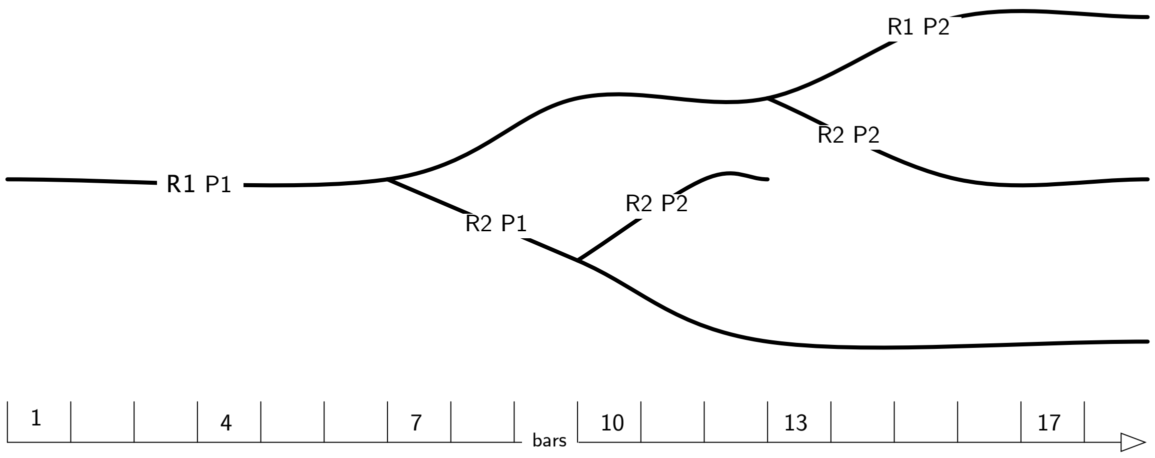


Figure 7.10.: Event stream structure of the first 18 bars of *Piz Bernina*

constraints. The goal was to find scales, which do not repeat in an octave, exclude a number of pitch classes in a specified range, and only contain a specified set of intervals. Table 7.2 shows the scales and the excluded pitch classes of the four pitch sets and figure 7.11 shows the four pitch sets in the ranges used, in musical notation. A goal of this approach was to create sets of pitch material which are identifiably different from each other. The exclusion of certain pitch classes, and the restriction to certain intervals creates complimentary sets of different characters.

Nr	**Scale in intervals (semitones)**	**Excluded pitch classes**
1	3, 5, 1, 1, 4, 1, 3	1, 5, 7, 11
2	1, 3, 6, 1, 2, 3, 2	2, 3, 8, 9
3	5, 2, 4, 2, 5, 5, 2, 5	0, 4, 5, 6, 8, 10, 11
4	1, 7, 1, 3, 6, 2, 1	2, 7

Table 7.2.: The pitch sets of *Piz Bernina*

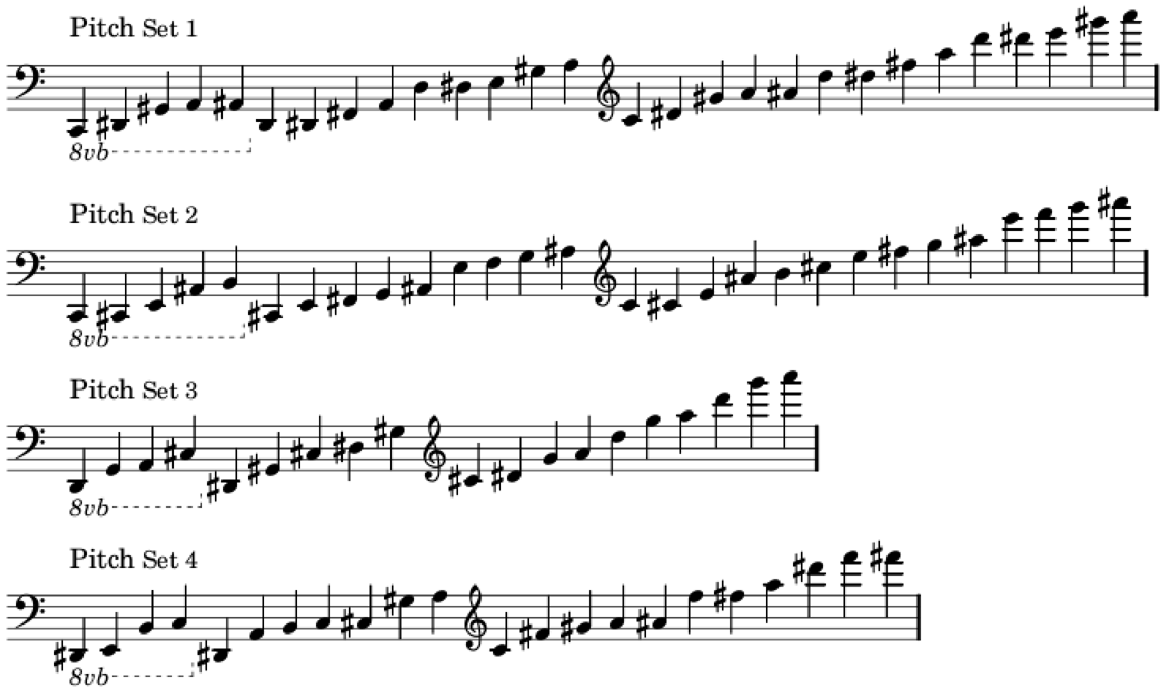


Figure 7.11.: The four pitch sets of *Piz Bernina*

The rhythmic organization of *Piz Bernina* shows some similarity to Clarence Barlow's probabilistic approach to distributing event attacks according to hierarchical divisions of meters (see [Bar80]). In *Piz Bernina*, any rhythmic unit, starting from the bar as the longest unit, can be subdivided into either 2, 3, or 5 subunits, which in turn can be subdivided again. A subdivision carries a list of weights, known as factors, and these factors will be inherited by further generations. A division by two assigns the weight 1 to the first unit and a weight of 0.5 to the second unit. A division by three assigns a weight of 1 to the first, 0.4 to the second, and 0.7 to the third unit. A division by five assigns a weight of 1 to the first, 0.3 to the second, 0.7 to the third, 0.4 to the fourth, and 0.5 to the fifth. A metric division of a bar is built by consecutively dividing the units. For example, one bar may be divided by 2 and the resulting units will be divided by 3. This division (notated 2x3) contains 6 units with the following weights: 1 (1x1), 0.4 (1x0.4), 0.7 (1x0.7), 0.5 (0.5x1), 0.2 (0.5x0.4), and 0.35 (0.5x0.7). Of course, more layers than two can be used, as figure 7.12 shows. These weights operate as probabilities, which reflect the hierarchical structure of the meter. In *Piz Bernina* there are three divisions used: 2x2x2x2, 2x3x2x2, and 5x2x2.

As describe above, *Piz Bernina* is structurally similar to *Piz Palü* and in the course of the piece event streams continue, split, merge, and die, forming an arborescent structure. The lowest-level event here is a note, the event streams, however, produce bars. Furthermore, the state of an event stream contains several settings used for parametric organization. Higher-level changes, thus, mostly occur at bar changes. The state which is carried from event to event, is thus carried from bar to bar. These settings are as follows, the employed pitch set, the current pitch range, the metric division, and finally the density value.

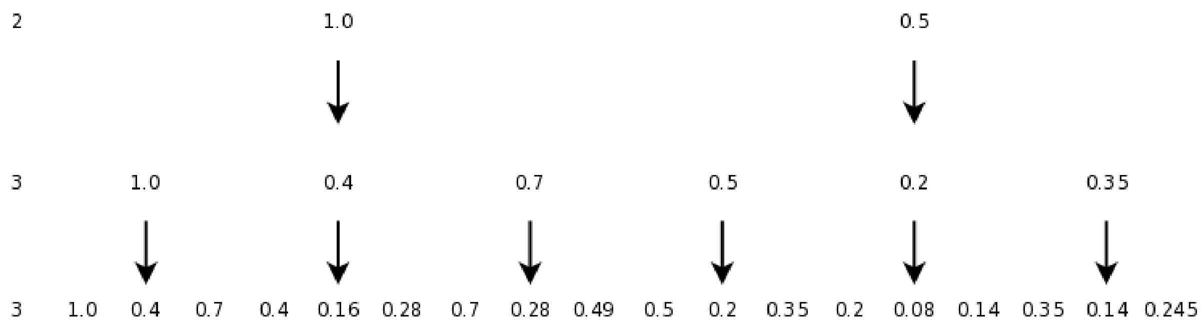


Figure 7.12.: Attack weights for a division of 2x3x3

Part IV.

Appendix

Contents of the CD

The accompanying data CD contains the described program *CompScheme*, my composition discussed in the third part of this text, as well as this text itself in PDF and HTML form. The reader may open a file called *index.html* on the root level of the CD in a browser, which will guide him or her through the contents.

Directory	**Content**
/compscheme/bin	Contains the *CompScheme Cocoa* binary application for *MAC OS X* 10.5 on Mac Intel
/compscheme/examples	Contains all the *CompScheme* examples of this text and additional examples
/compscheme/sources	Contains the source code of *CompScheme* and compilation instructions (see README file)
/music	Contains *Piz Palü*, *Piz Zupò*, and *Piz Bernina* in WAV file format, as well as the score of *Piz Bernina*
/thesis/pdf	Contains this document as a PDF file
/thesis/html	Contains this document in HTML file format

Table 7.3.: Contents of the CD

Bibliography

[Ado49] Theodor W. Adorno. *Philosophie der neuen Musik*. Suhrkamp, 1949.

[Ado73] Theodor W. Adorno. *Ästhetische Theorie*. Suhrkamp, 1973.

[Ame87] Charles Ames. Automated composition in retrospect: 1956-1986. *Leonardo*, 20(2):169–185, 1987.

[ASS96] H. Abelson, G. Sussman, and J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, 1996.

[Bar80] Clarence Barlow. Bus journey to parametron. *Feedback Papers*, (21–23), 1980.

[Ber07] Paul Berg. *Using the AC Toolbox*. Institute of Sonology, 2007.

[Boe67] Konrad Boehmer. *Zur Theorie der offenen Form in der neuen Musik*. Edition Tonos, Darmstadt, 1967.

[Brü69] Herbert Brün. Infraudibles. In Heinz von Foerster and James W. Beauchamp, editors, *Music by Computers*, pages 171–121. John Wiley and Sons, Inc., 1969.

[Brü04] Herbert Brün. *When Music Resists Meaning*. Wesleyan University Press, 2004.

[DDH97] Roger B Dannenberg, Peter Desain, and H. Honing. *Musical Signal Processing*, chapter Programming Language Design for Music, pages 271–315. Swets & Zeitlinger, 1997.

[Dew34] John Dewey. *Art as Experience*. Perigee, 1934.

[Döb08] Luc Döbereiner. Compscheme: A language for composition and stochastic synthesis. In *Sound and Music Computing (SMC'08) Conference Proceedings*, Berlin, 2008.

[Ess89] Karlheinz Essl. *Musik-Konzepte 66: Gottfried Michael Koenig*, chapter Zufall und Notwendigkeit. edition text + kritik, 1989.

[Goe10] J.W.v. Goethe. *Geschichte der Farbenlehre*. 1810.

[Goo68] Nelson Goodman. *Ways of World Making*. Hackett, 1968.

[Goo76] Nelson Goodman. *Languages of Art*. Hackett, 1976.

[GP96] T. R. G. Green and Marian Petre. Usability analysis of visual programming environments: A 'cognitive dimensions' framework. *Journal of Visual Languages and Computing*, 7(2):131–174, 1996.

[HI59] Lejaren Hiller and Leonard Issacson. *Experimental music : composition with an electronic computer*. McGraw-Hill, 1959.

[Hof00] Peter Hoffmann. The new gendyn program. *Computer Music Journal*, 24(2):31–38, 2000.

[Knu69] Donald E. Knuth. *The Art of Computer Programming*, volume 2. Addison Wesley, 1969.

[Koe69] Gottfried Michael Koenig. *Übung für Klavier*. TONOS Musikverlags GmbH, 1969.

[Koe70a] Gottfried Michael Koenig. Project 2, a programme for muscial composition. *Electronic Music Reports*, 1970.

[Koe70b] Gottfried Michael Koenig. The use of computer programmes in creating music. *La Revue Musicale*, 1970.

[Koe78] Gottfried Michael Koenig. Composition processes. *Computer Music Reports on an International Project, UNESCO Rapport*, 1978.

[Koe93a] Gottfried Michael Koenig. *Ästhetische Praxis*, volume 3, chapter Computer-Verwendung in Kompositionsprozessen (1969), pages 27–38. PFAU-Verlag, 1993.

[Koe93b] Gottfried Michael Koenig. *Ästhetische Praxis*, volume 3, chapter Der Computer in der Musik (1970), pages 41–56. PFAU-Verlag, 1993.

[Koe93c] Gottfried Michael Koenig. *Ästhetische Praxis*, volume 3, chapter Genesis der Form unter technischen Bedingungen (1986), pages 277–288. PFAU-Verlag, 1993.

[Koe93d] Gottfried Michael Koenig. *Ästhetische Praxis*, volume 3, chapter Ästhetische Integration mit einem Computer komponierter Partituren (1983), pages 263–271. PFAU-Verlag, 1993.

[LA85] Gareth Loy and Curtis Abbott. Programming languages for computer music synthesis, performance and composition. *Computing Surveys*, 17(2), 1985.

[Ler] Xavier Leroy. The objective caml system: Documentation and user's manual.

[Luq06] Sergio Luque. Stochastic synthesis: Origins and extensions. Master's thesis, Institute of Sonology, 2006.

[Man81] Peter Manning. Computers and music composition. *Proceedings of the Royal Musical Association*, 107:119–131, 1981.

[Mat05] Nouritza Matossian. *Xenakis*. Moufflon Publications, 2005.

[McC02] James McCartney. Rethinking the computer music language: Supercollider. *Computer Music Journal*, 26(4):61–68, 2002.

[Pau96] L.C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1996.

[Roa85] Curtis Roads. *Composers and the Computer*. William Kaufman Inc, 1985.

[Sto63] Karlheinz Stockhausen. *Texte zur elektronischen und instrumentalen Musik*, volume 1 (1952-1962), chapter Gruppenkomposition: Klavierstück 1. Verlag M.DuMont Schauberg, 1963.

[Vag01] Horacio Vaggione. Some ontological remarks about composition processes. *Computer Music Journal*, 25(1):54–61, 2001.

[Var96] Báltint András Varga. *Conversations with Iannis Xenakis*. Faber and Faber, 1996.

[Wit18] Ludwig Wittgenstein. *Tractatus logico-philosophicus*. Suhrkamp, Frankfurt am Main, 1960 (1918).

[Xen60] Iannis Xenakis. Grundlagen einer stochastischen musik. *Gravesaner Blätter*, (18), 1960.

[Xen92] Iannis Xenakis. *Formalized Music.* Pendragon Press, 1992.