Musical Composition with Grid Diagrams of Transformations

Masterarbeit
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Florian Thalmann

2007

Leiter der Arbeit

Prof. Dr. Guerino Mazzola
School of Music, University of Minnesota
Department of Informatics, University of Zürich

Prof. Dr. Oscar Nierstrasz
Institute of Computer Science and Applied Mathematics, University of Bern

The address of the author:

Florian Thalmann
Rodtmattstrasse 73
CH-3014 Bern
thalmann@students.unibe.ch

Software Composition Group
University of Bern
Institute of Computer Science and Applied Mathematics
Neubrückstrasse 10
CH-3012 Bern
http://www.iam.unibe.ch/~scg/

Abstract

In the present thesis, the concepts of OrnaMagic, a module for the generation and application of musical grid structures, which is part of the music composition software *presto* (Atari ST), are generalized, abstracted and adapted for modern functorial mathematical music theory. Furthermore, an new implementation for the present day composition software Rubato Composer (Java) is provided.

Acknowledgements

Many people contributed to this master's thesis and I am grateful to all of them. First and foremost, I would like to thank my primary supervisor, Prof. Dr. Guerino Mazzola. He taught me Mathematical Music Theory and Music Informatics, both very fascinating domains, and during the evolution of this thesis, he guided me with great competence, open mindedness and enthusiasm. It was a great experience to write the two papers related to this thesis with him and to contribute to the forefront of music research.

I wish to thank Prof. Dr. Oscar Nierstrasz, my secondary supervisor, for giving me the opportunity to work in his Software Composition Group (SCG) and for taking on all the administrative complications arising from this external collaboration. Furthermore, I would like to thank him and Orla Greevy from the SCG for the thorough proof-reading of this thesis.

I am grateful to Gérard Milmeister and Julien Junod from the Visualization and MultiMedia Lab at the University of Zürich, as well as Karim Morsy from the Technische Universität München, for the interesting and helpful discussions and the great time we had. Many thanks go to the people working in the SCG student pool in Bern for their good company, especially Pascal Zumkehr, Reto Kohlas, and Stefan Reichhart.

I would like to extend my deepest gratitude to my family and friends for their support, for their belief in me and for their appreciation for what I do.

Florian Thalmann, April 2007

Contents

List of Figures

Chapter 1

Introduction

The use of computers for *music composition* became more and more important by the end of last century. First of all, there are many commercial software products such as sequencing software [Logic] [Cubase], which are indispensable for the production of modern popular music, or notation software [Sibelius] [Finale]. However, already in 1991, Karlheinz Essl, one of today's leading computer music composers, said that for entering unknown musical territory, the widespread commercial products are not suitable [Essl91]. For such uses, since the early days of computer music, composers like Max Mathews, Gottfried Michael Koenig and Iannis Xenakis used to write their own software and this has not changed since then.

Around the same time when Essl made this comment, an innovative software product named *presto* has been released. The goal of this software was to offer the reusability of a flexible and mature software product without limiting the creativity of the composer using it. It was based on cutting-edge mathematical music theory, formulated by Guerino Mazzola [Mazz90] and provided numerous functions for composing and manipulating music geometrically.

Described as one of the most powerful music composition tools to date [Mazz89], a module in *presto*, namely the *OrnaMagic* module, allows to generate and use *periodic grid constructions* in an abstract way never seen since. Many other software products incorporate similar functions, which are in fact special cases of OrnaMagic applications, namely quantizing or tonal alteration.

Today, the Atari ST is outdated and modern *mathematical music theory* has made many steps forward [Mazz02] and now uses more abstract and versatile data types and operations. A new software product has been created according to these new theoretical foundations, namely Rubato Com-

POSER [Milm06a] [RubatoC]. It is a platform independent Java 1.5 application, featuring an extended mathematical framework as well as a modular design.

1.1 Problem Statement

Since *presto*, musical grid structures and their different application possibilities have never been implemented at the same level, not to mention developed further. This leads us to the research question of this thesis:

> **Research Question:**
>
> *How can the concepts of OrnaMagic be adapted to modern functorial mathematical music theory and how can they be implemented for* RUBATO COMPOSER*?*

1.2 Contributions

The contributions of this thesis are:

- Analysis of OrnaMagic's musical grid structures named *ornaments*, as well as their application possibilities.
- Generalization of these concepts for functorial mathematical music theory. Definition of two major mathematical constructs, namely *wallpapers* and *alteration*.
- Definition of necessary additional constructs for practical usage of the major constructs, namely *melodies, rhythms* and *scales*.
- Implementation of the formalized constructs in form of five *rubettes*, i.e. plugins for RUBATO COMPOSER, namely the *Wallpaper, Alteration, Melody, Rhythmize* and *Scale* rubettes.
- Composition of different examples showing the uses of the implemented rubettes.

1.3 Related Work

Speaking of related works, it is important to also illuminate the historical background from music composition. Musical grid structures have been a common composition tool in Western music for centuries. Here are some examples:

- *Real imitation* in counterpoint [Fux65]. A melodic line is imitated note by note, where note values and intervals stay the same. This pattern was used for example by Josquin Desprez (1457/58 - 1521) in his polyphonic music and later refined by Johann Sebastian Bach (1685 - 1750) for the composition of his great fugues.

- Arnold Schönberg (1874 - 1951) tiled melodic patterns and their transformations, e.g. the retrogrades or inversions, in his *twelve-tone technique*.

- In his works 'Harawi' or 'Visions de l'Amen' for example, Olivier Messiaen (1908 - 1992) used a special technique called *rhythmic canon* [Andr02], where just rhythms are imitated by different voices.

- Conlon Nancarrow (1912 - 1997) experimented with melodic motifs and mathematical rhythmic patterns in his work *'Studies for the Player Piano'*.

- Steve Reich (*1936), one of the pioneers of *Minimal Music*, composed music based on temporally shifted tiled patterns, for example in his 'Piano Phase'.

More generally, *rhythm patterns* and *loopings* in many musical styles from around the world can be seen as repetitive grid structures.

Besides the previously mentioned *presto*, just a few attempts have been made to provide a computer program for building such grid structures. All examples stated here have a clearly distinct goal from the contributions of this thesis:

- Moreno Andreatta *et al.* [Andr01] [Andr02] implemented Olivier Messiaen's rhythmic canons for the platform independent composition software *OpenMusic* [Bres05].

- Karlheinz Essl provided several related functions in his *Real-Time Composition Library* for Max/MSP [Essl96].

- Several applications of *Cellular Automata* for MIDI composition have been realized, most commonly producing two-dimensional grid structures for the time-pitch plane [Burr04].

- In the tutorial for jMusic, a music composition framework for Java, Steve Reich's 'Piano Phase' was reconstructed, a typical musical grid structure [JMusic]. Many *musical programming languages*, such as Common Music [Taub91] or Haskore [Huda96], can be used to produce similar patterns, but not on the sophisticated and complex mathematical level of RUBATO COMPOSER.

- The concept of *looping* is widespread in music software and we investigated it in our collaborative music generation software Jam Tomorrow [Thal06].

1.4 Thesis Outline

Chapter 2 (p.5) gives a brief overview of modern *mathematical music theory* and the two related music composition applications *presto* and Rubato Composer.

Chapter 3 (p.21) presents the concepts of *presto*'s *OrnaMagic module* and gives some common examples for its application.

Chapter 4 (p.27) identifies the limitations of OrnaMagic, proposes a structure for a new implementation and *generalizes and formalizes* concepts for functorial mathematical music theory.

Chapter 5 (p.45) describes how the mathematical formalizations are *implemented* for Rubato Composer and gives an overview of the implementation design.

Chapter 6 (p.53) discusses how our new implementation can be used to produce meaningful compositions and gives some *examples* for inspiration.

Chapter 7 (p.63) looks back and compares the new implementation with its predecessor. Furthermore, it provides an outlook on its future uses and further work.

Appendix A (p.67) is an addition to the Rubato Composer *manual* [Milm06b] and serves as a reference to the software implemented for this thesis.

Chapter 2

Mathematical Music Theory and Applications

Mathematical music theory has become an indispensable basis for music informatics and computational musicology. It has been developed by mathematicians or composers like Leonard Euler [Eule39], Milton Babbitt [Babb92] or David Lewin [Lewi87][Lewi93]. The most recent advancement has been made by Guerino Mazzola in his work *The Topos of Music* [Mazz02]. His theorical achievements have been applied in several software products on different platforms, namely [Presto], [RubatoN], [RubatoX] and [RubatoC].

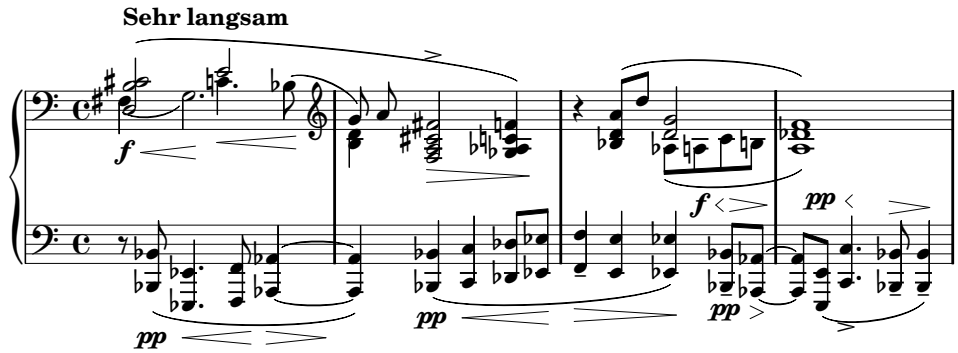
Two of these software products, *presto* and RUBATO COMPOSER, facilitate music composition by providing composers with convenient mathematical tools. We describe both of them in this chapter, including their underlying theories. *presto* uses a geometrical representation of music, whereas RUBATO COMPOSER, its present-day descendant, is based on modern functorial mathematical music theory.

2.1 Music Notation

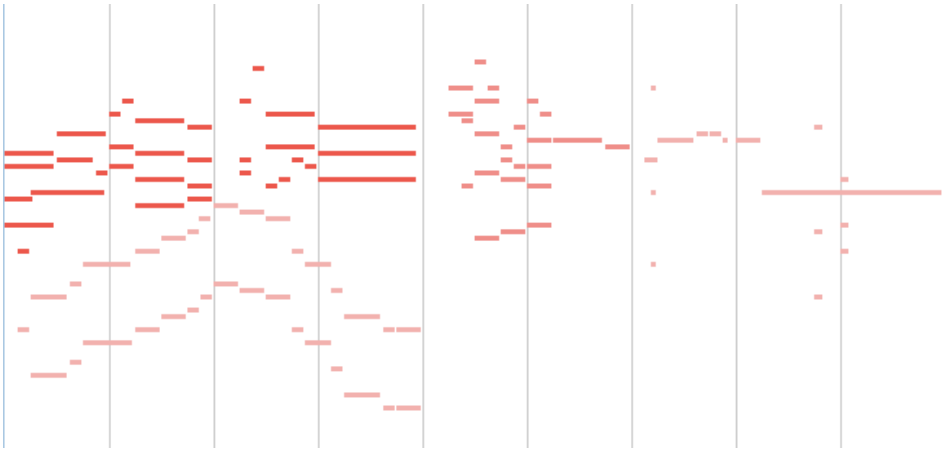
Before we attend to mathematical music theory, we provide a short introduction to the different visual representations of music, used in this thesis.

In the standard *western score notation* [Ston80], note symbols are placed on the staff comprising five lines. Depending on their position on the staff, note symbols represent notes of a different pitch (the higher symbol, the higher the pitch). Duration is symbolized by different note values and additional symbols like dots and ties, whereas loudness (i.e. dynamics) is often specified below the staff. Figure 2.1 (a) shows an example for a piano score, the lower

staff representing the player's left hand and the upper staff the right hand. The examples in this thesis have been typeset using the open source music engraving software LilyPond [Nien03].



(a)



(b)

Figure 2.1: Arnold Schönberg's piano piece Op. 19, No. 3, (a) the first four bars in western score notation and (b) the whole piece in piano roll notation.

In modern music software, the *piano roll notation* system established itself, due to its flexibility, abstractness and simplicity. It is inspired by the medium used to operate mechanical player pianos, hence the name. Notes are represented by horizontal bars in a two-dimensional coordinate system, where the horizontal axis represents time and the vertical axis pitch. The length of such a note bar describes the duration and its opacity often the loudness of the represented note (see Figure 2.1 (b) for an example). Different voices are mostly visualized using different colors. The piano roll notation is very useful for showing complex musical structures, but unsuitable for precise applications, such as sight reading by musicians. However, it is used in Rubato Composer and therefore many of the examples in

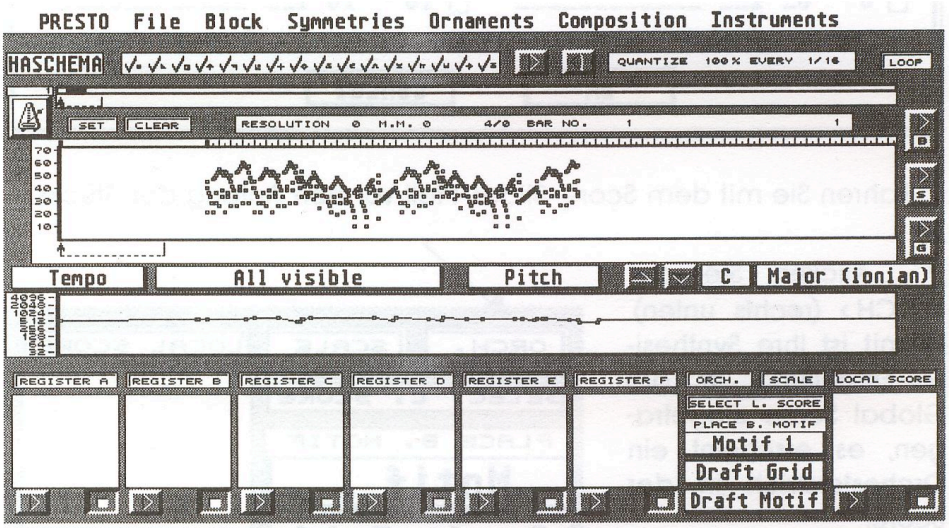


Figure 2.2: The main view of the *presto* software for Atari ST.

this thesis are visualized in this way.

2.2 Geometry of Notes: *presto*

It is interesting to note that the close affiliation of music and geometry has existed for centuries. The western score notation, for example, and its predecessor, the neumatic signs, can be seen as a grid, where notes are drawn as points and melodies as curves. This representation has been abstracted at the end of the last century and this is indispensable for music to be represented and processed by computers.

Some of the major abstractions in this field were made by Guerino Mazzola by the end of the 1980s and described in his book *Geometrie der Töne* [Mazz90]. The theoretical foundations were subsequently implemented in the music composition software *presto*, described in this section.

presto [Mazz89] [Presto] was published in 1989 for Atari ST computers. It provides a very elaborate interface (shown in Figure 2.2) to compose and manipulate music and also supports reading from and saving to standard MIDI files [MIDI96], recording music using a MIDI interface, drawing music using the mouse or printing the composed music.

These features are all wide spread among music software products, but what mainly distinguishes *presto* from other applications is its ability to generate and modify music by performing a variety of geometrical operations on it.

Its unique tools are based on a *geometrical representation* of music, which we describe now.

In *presto*, notes are defined as points in a four-dimensional geometrical space spanned by the following coordinates:

- *onset*, the time when a note is played,
- *pitch* on the chromatic scale,
- *duration* in time and
- *loudness*.

All four coordinates have discrete values, where onset is between 1 and 11360 and the other three between 1 and 71. Additionally, one of 16 instruments can be chosen for every note in order to orchestrate a composition. However, this parameter is not included in the geometrical space and therefore not relevant with regard to the future explanations.

For the manipulation of such note points, two of the four coordinates can be chosen to be displayed in the score view (Figure 2.2). There, any number of notes can be selected and modified by any specified *geometrical transformation* or set operation. These modifications act on the selected two-dimensional level and include for example transformations like inversion, retrograde or parameter exchange, or set operations like union or intersection. An example for such a transformation in *presto* is shown in Figure 2.3. According to theorem 53 in Appendix E, Section 3.6 in [Mazz02], by combining such two-dimensional operations, any transformation in the four-dimensional musical space can be realized.

The second version of *presto* (1993) involves an interesting module called *OrnaMagic*, from which the conceptual basics for this thesis are taken. These concepts will be described in Chapter 3. In this chapter we focus on decribing the issues of mathematical music theory.

2.3 Functorial Mathematical Music Theory

About a decade later, Mazzola developed further the theoretical base of *presto*. The achievements are described in *The Topos of Music* [Mazz02] and are based on *category theory* [Pier91] and *topos theory* [Gold84], used to formalize, manipulate and analyze musical objects. These theoretical foundations have again been implemented. The corresponding software product is called RUBATO COMPOSER and is described later.

This section gives a brief introduction to the data format of functorial mathematical music theory and the different ways of its manipulation, just as

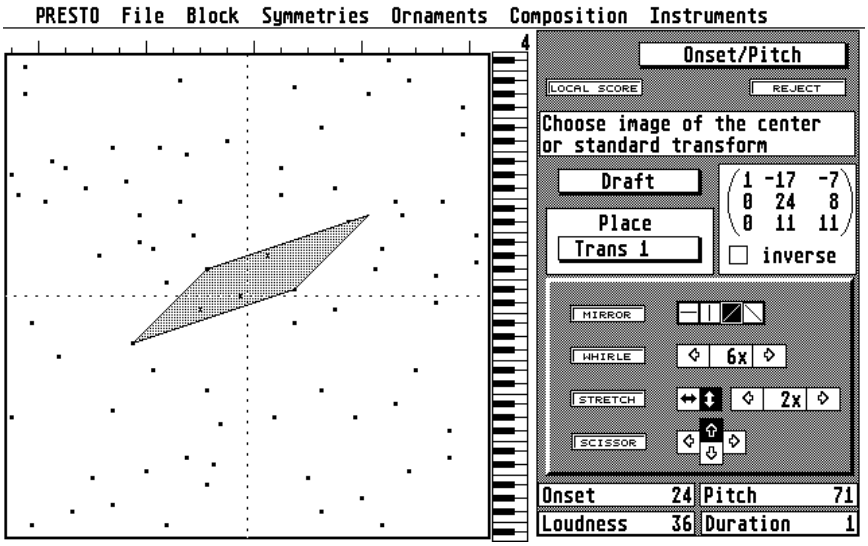


Figure 2.3: The *presto* transformation score.

much as suffices for understanding the later explanations. To understand it fully, a founded mathematical background is required. We refer to Part XVI of [Mazz02] or Chapter 4 of [Milm06a], which both give a straightforward introduction to the mathematical foundations. For the readers who are not familiar enough with the underlying mathematics, we draw analogies to concepts of computer science.

2.3.1 Forms and Denotators

The data format of forms and denotators has many parallels to the concept of object-oriented programming (OOP) in computer science. Denotators can be seen as *points* in a specified form-space, as shown in Figure 2.4. In OOP, one would speak of an object being a point in the space spanned by the attributes of its class. Alternatively, expressed in common OOP language, denotators are instances of forms.

Complex forms are constructed recursively from the primitive **Simple** forms, like in a programming language complex data types are built based on primitive data types like integer numbers or strings. The recursive structure of a *form* F is defined as follows:

$$\mathrm{Name}_F : Id_F.\mathrm{Type}_F(\mathrm{Dia}_F).$$

The form's name Name_F is a a denotator (definition below), its identifier Id_F is a monomorphism $Id_F : \mathrm{Space}_F \to \mathrm{Frame}_F$ of presheaves, which are

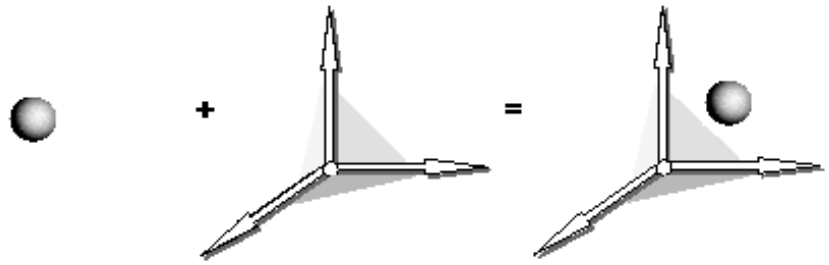


Figure 2.4: A schematic representation of a denotator (left), its form (middle) and the represented object (right).

contravariant functors $P : \mathbf{Mod} \to \mathbf{Sets}$ from the category of modules **Mod** to the category of sets **Sets**. Type_F is one of four form types: **Simple**, **Limit**, **Colimit** or **Power**, and Dia_F is a diagram of forms, defined as follows depending on the specified type:

- **Simple**: Dia_F is a module M with $\text{Frame}_F = \text{Hom}(-, M) = @M$. As previously stated, **Simple** is the basic type and it is comparable to the primitive data types in a programming language. However, the category of modules includes number rings such as $\mathbb{Z}, \mathbb{Z}_n, \mathbb{Q}, \mathbb{C}$ as well as word monoids $\langle \mathit{Unicode} \rangle$ or polynomial rings $R[X]$ among others, which facilitate the construction of more complex mathematical structures than in a programming language.

- **Limit**: the frame space Frame_F is the limit $\mathit{lim}(\text{Dia}_F)$ of the form spaces of the given diagram. If the Dia_F is discrete (i.e. no arrows), Frame_F is the cartesian product, which is a conjunction of attributes. In OOP, the corresponding construction would be a class with several attributes.

- **Colimit**: here, Frame_F is the colimit $\mathit{colim}(\text{Dia}_F)$ of the given diagram's form spaces. In case Dia_F is discrete, we have the disjoint union of a list of cofactors. To realize such a disjunction for a set of classes in OOP, a superclass can be defined for them.

- **Power**: Dia_F has a single vertex form G and no arrow. $\text{Frame}_F = \Omega^{\text{Space}_G}$, Ω being the subobject classifier (see [Mazz02] Appendix G, Section 3.1). This corresponds to an collection of instances of the specified form G. For this, we normally have predefined set classes in OOP.

Starting with **Simple** types and using the other (compound) types, more complex forms can be created. Here are, for example, the definitions of some

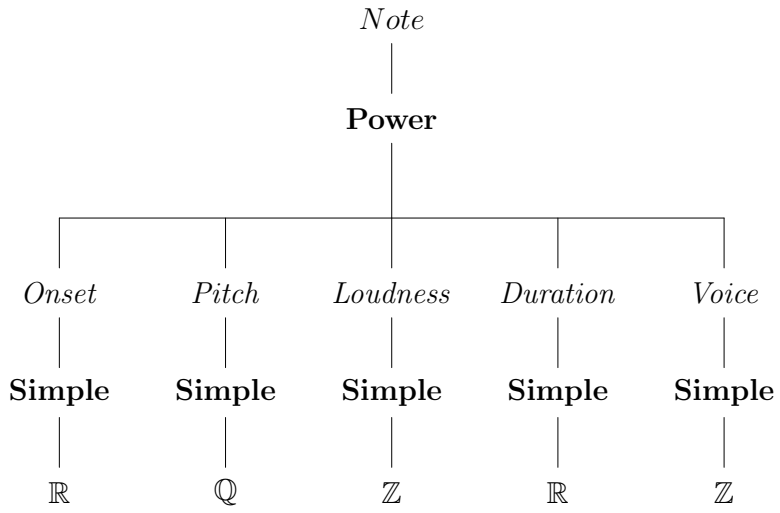


Figure 2.5: The tree representation of the *Note* form.

of the most common forms related to music:

$$\begin{aligned}
\mathit{Score} &: \mathit{Id}.\mathbf{Power}(\mathit{Note}) \\
\mathit{Note} &: \mathit{Id}.\mathbf{Limit}(\mathit{Onset}, \mathit{Pitch}, \mathit{Loudness}, \mathit{Duration}, \mathit{Voice}) \\
\mathit{Onset} &: \mathit{Id}.\mathbf{Simple}(\mathbb{R}) \\
\mathit{Pitch} &: \mathit{Id}.\mathbf{Simple}(\mathbb{Q}) \\
\mathit{Loudness} &: \mathit{Id}.\mathbf{Simple}(\mathbb{Z}) \\
\mathit{Duration} &: \mathit{Id}.\mathbf{Simple}(\mathbb{R}) \\
\mathit{Voice} &: \mathit{Id}.\mathbf{Simple}(\mathbb{Z})
\end{aligned}$$

Such compound forms have a *tree structure*, where the contained **Simple** forms are leaves and all other forms are nodes. They can be visualized as shown in Figure 2.5. In the next section, we discuss some singularities of such trees.

A *denotator* D is an instance of such a space and it is defined as follows:

$$\mathrm{Name}_D : \mathrm{Add}_D @ \mathrm{Form}_D(\mathrm{Coord}_D),$$

where Name_D is again a denotator, the address Add_D is a module A, Form_D is a form F and the coordinate Coord_D is an element of the evaluation $A@\mathit{Space}_F$ of F's form space Space_F.

As an example, let us create a denotator of the previously defined *Score*

form:

$$
\begin{aligned}
o_1 &: 0@Onset(1.0)\\
o_2 &: 0@Onset(1.5)\\
o_3 &: 0@Onset(2)\\
p_1 &: 0@Pitch(\tfrac{60}{1})\\
p_2 &: 0@Pitch(\tfrac{66}{1})\\
p_3 &: 0@Pitch(\tfrac{58}{1})\\
l &: 0@Loudness(120)\\
d &: 0@Duration(1)\\
v &: 0@Voice(0)\\
n_1 &: 0@Note(o_1, p_1, l, d, v)\\
n_2 &: 0@Note(o_2, p_2, l, d, v)\\
n_3 &: 0@Note(o_3, p_3, l, d, v)\\
s &: 0@Score(n_1, n_2, n_3)
\end{aligned}
$$

Before discussing the possibilities to modify such a denotator, we analyze their structure more precisely.

2.3.2 Paths

As we have seen before, compound forms have a tree structure. This is also the case for compound denotators. However, depending on which form types they include, there can be some significant differences between them. Precisely, in a form tree, a form of type **Power** has one branch, whereas in a denotator tree, a denotator of the same form can have an arbitrary number of branches. Also, a form of type **Colimit** in a tree has a branch for each of its coordinators, whereas its denotators only have one branch.

Now to denote a specific form or denotator in such a tree, we use *paths*. A path is defined as a sequence of integers $i. = (i_0, \ldots i_m)$, where each i_k is the index of the branch to follow from the current to the next node, starting at the root. Normally, we use the form paths for a form and its denotators, but because of the above described singularities, we have to deal with two problems.

If the path traverses a **Power** form in the form tree, we take the set of n paths in the denotator tree, one path for each of the corresponding **Power** denotator's elements. Therefore, if a transformation (see next section) is to be applied to a denotator of the form denoted by such a form path, it is applied to all elements of the **Power** denotator lying in between. See [Milm06a] Chapter 11, Section 4.2 for more about this.

A form path traversing a **Colimit** form indicates which of its m coordinators should be reached. If it is not present in the corresponding denotator tree, we obtain an erroneous path. If a transformation involves such a non-present denotator, it is simply not performed. A more precise discussion of these problems in respect of our application field will be given in Chapters 4 and 5.

2.3.3 Denotator Transformations and Set Operations

The functorial approach defines three possible operations for the manipulation of denotators, namely space transformations, set-theoretic constructions and address changes, as concisely described in [Mazz06]. In this section, the first two of them are presented, as we make frequent use of them in Chapter 4. The third is presented in the next section.

Given a denotator $d : A@F(c)$ with the coordinate $c \in A@\text{Space}_F$ and a form morphism $f : F \to G$, we define the *mapping* (space transformation) of d by f as follows:

$$f(d) = d' : A@G(fc),$$

where d' is a new denotator with the mapped coordinate $fc = A@f(c)$. For example, we define a projection morphism $p_d : \mathit{Note} \to \mathit{Duration}$, which projects a note denotator on its duration. Applying this morphism p_d to the above defined note n_2, we obtain:

$$p_d(n_2) = n_2' : 0@\mathit{Duration}(1)$$

The second operation just concerns denotators of forms of type **Power**, but since most constructions in music are based on **Power** forms (sets of notes), we will often use it. We speak of *set-theoretic operations* $d \cup e, d \cap e$ and $d - e$ for two **Power** denotators, which share the same address A. For example, if $d : 0@\mathit{Score}(n_1, n_2)$ and $e : 0@\mathit{Score}(n_2, n_3)$, we obtain:

$$d \cup e : 0@\mathit{Score}(n_1, n_2, n_3) \text{ and } d \cap e : 0@\mathit{Score}(n_2).$$

2.3.4 Non-Zero-Addressed Denotators

The third operation in our functorial approach is the *address change*. For a given denotator $d : A@F(c)$ and an address change $g : B \to A$, we obtain a new denotator $d' : B@F(c \circ g)$, where $c \circ g$ is the image of c under the map $g@\mathit{Space}_F$.

Note that the above defined denotators are all zero-addressed. To give an typical example for an application of address change, we redefine the same

score s as a $\mathbb{Z}^2$-addressed denotator:

$o : \mathbb{Z}^2@Onset(m_o)$ with $m_o : \mathbb{Z}^2 \to \mathbb{R}$ and $m_o(z_1, z_2) = 0.5z_1 + z_2 + 1$
$p : \mathbb{Z}^2@Pitch(m_p)$ with $m_p : \mathbb{Z}^2 \to \mathbb{Q}$ and $m_p(z_1, z_2) = \frac{6}{1}z_1 - \frac{2}{1}z_2 + \frac{60}{1}$
$n : \mathbb{Z}^2@Note(o, p, l, d, v)$
$\hat{s} : \mathbb{Z}^2@Score(n)$

To obtain a denotator equal to the above defined s, we have to evaluate $\hat{s}$ at specific elements of its address $\mathbb{Z}^2$. Precisely, we defined its coordinates so that these elements are the basis vectors $e_1 = (1, 0)$ and $e_2 = (0, 1)$ as well as the zero vector $e_0 = (0, 0)$ of $\mathbb{Z}^2$. To address such an element, we need to define the constant map $g_{e_i} : 0 \to \mathbb{Z}^2$ with $g_{e_i}(0) = e_i$. We hereby obtain the denotators:

$$\begin{aligned} s_{e_i} &: 0@Score(n_{e_i}) \\ n_{e_i} &: 0@Note(o_{e_i}, p_{e_i}, l, d, v) \\ o_{e_i} &: 0@Onset(m_o \circ g_{e_i}) \text{ and} \\ p_{e_i} &: 0@Pitch(m_p \circ g_{e_i}) \end{aligned}$$

for $0 \leq i \leq 1$. Uniting these denotators, we obtain $\hat{s}' = s_{e_0} \cup s_{e_1} \cup s_{e_2}$ which is equal to s.

For more precise information about forms, denotators and their operations, see [Mazz02]. We now go on to an introduction to Rubato Composer, the current software product implementing the theories just presented.

2.4 Rubato Composer

Rubato Composer [Milm06a] is an *open source Java 1.5 application* and a *framework* for music composition, that is being developed by Gérard Milmeister. It follows the classical Rubato architecture [Mazz94] and consists of a main application and a number of plugin modules, named *rubettes*. In a graphical user interface (GUI), the rubettes can be connected to form a data exchange *network*, which can then be run by pressing a button. Initially, we delineate the GUI possibilities and the rubette concept, we take a closer look at the mathematical framework of Rubato Composer and the data types currently used.

2.4.1 User Interface

The Rubato Composer GUI is divided in three areas (see Figure 2.6). The main area is a tabbed panel in the center, each of the tabs representing

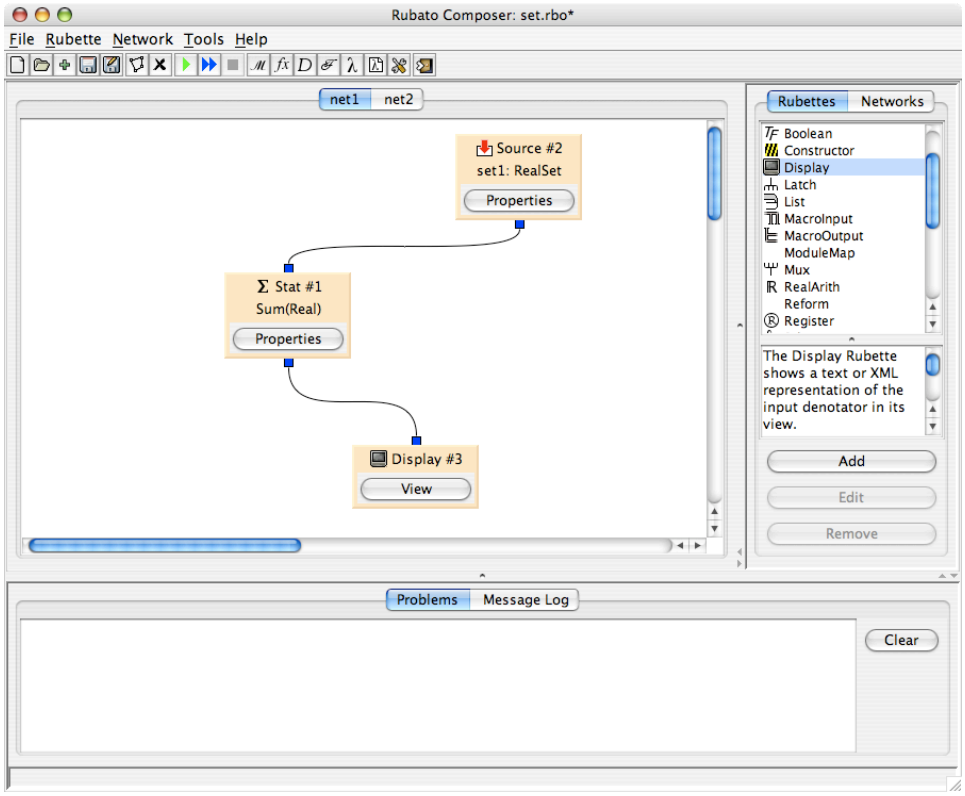


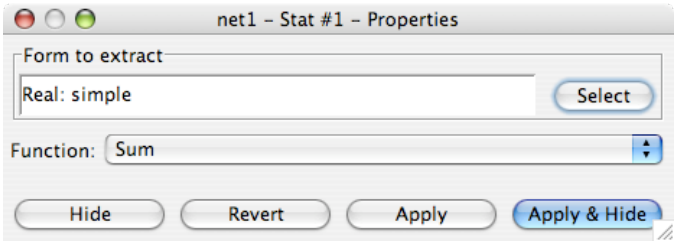
Figure 2.6: The main window of the RUBATO COMPOSER GUI showing a network of three rubettes.

a network of rubettes. To build such a network, rubettes can be dragged from the area at the right border of the window, where there is a list of all present rubettes. The lower area is used to display messages generated during construction or execution of a network.

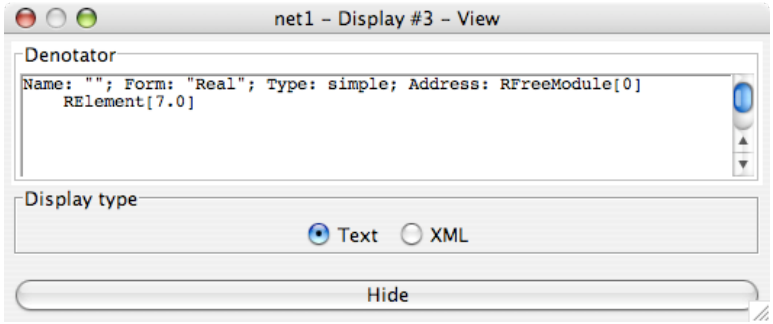
By pressing on the icons in the bar above these areas, several dialog windows can be opened for the definition of morphisms, forms or denotators. To read more about these functions, see [Milm06b].

2.4.2 Rubettes

A rubette is a Rubato software module, represented by a box with input or output connectors or both to receive and send data (Figure 2.6). The data format used for communication between rubettes is the *denotator format* described in Section 2.3. So, a rubette can either receive a number of input denotators to be manipulated or generate denotators itself. Then, denotators can be sent on to other rubettes.



(a)



(b)

Figure 2.7: The Stat rubette's properties window (a) and the Display rubette's view window (b).

A rubette may need specific user parameters to define the characteristics of its task. These are provided by an optional *properties window* (Figure 2.7 (a)), appearing when the properties button is pressed. Furthermore, a *view window* may be generated for each rubette, to show results or data representation, i.e. text, graphs or images (Figure 2.7 (b)).

Rubettes are easy to implement and they can manipulate denotators in any way a programmer can imagine. To create a rubette, the class `AbstractRubette` is subclassed and its abstract methods have to be implemented (details about this are given in Chapter 5). Then, the compiled rubette, packed in a Java archive, has to be put in a specific directory where it is automatically recognized by the main application on the next startup.

2.4.3 Mathematical Framework

Apart from classes for rubette generation and reusable GUI components, the RUBATO COMPOSER framework provides all necessary classes for denotator generation and manipulation. They are located in the package `org.rubato.math`. Here, we take a brief look at the most important classes

in this package.

Forms and denotators are both represented by objects generated at runtime, instantiated from specific subclasses of the classes `Form` and `Denotator`, e.g. `LimitForm` or `PowerDenotator`, depending on their type. As described in Section 2.3.1, there are two major kinds of forms and denotators: simple and compound. First we talk about creating simples.

Following the theoretical model, to generate a **Simple** form F (class `SimpleForm`), we specify a `Module` M as a constructor argument. This module defines the range of the possible values of F. There are many different kinds of modules defined in RUBATO COMPOSER. Besides the common $\mathbb{Z}, \mathbb{Q}, \mathbb{R}, \mathbb{C}$ and $\mathbb{Z}_n$, we also have for example product rings, string rings or polynomial rings. It is one of the major difficulties when programming flexible denotator manipulation processes for Rubato, to always consider all possible modules. We describe this problem later.

A denotator (an instance of `SimpleDenotator`) of the same **Simple** form F is generated by specifying F itself and a `ModuleMorphism` with codomain M. For zero-addressed denotators, one can also simply specify a `ModuleElement` $m \in M$, which then induces the generation of a `ConstantMorphism` (subclass of `ModuleMorphism`) f with $f : 0 \to M$ and $f(0) = m$.

Starting from the generated simples, more complex *compound* forms or denotators can be built. For this, their coordinators or coordinates, respectively, need to be defined. To instantiate class `ColimitForm`, for example, we have to specify a `List<Form>` of forms as coordinators, whereas for a `ColimitDenotator`, the coordinate is just one denotator being of one of the coordinator forms.

For *denotator manipulation*, we mainly perform transformations using instances of subclasses of the class `ModuleMorphism`. To transform a denotator, its value (a module morphism) is combined with other module morphisms using operator methods like `sum()`, `compose()` and `difference()`. This can be also done by calling the `map(ModuleMorphism m)` method of `SimpleDenotator`. For example, assume we have a denotator d of form F for module M. If we wish to realize a translation of d by $m \in M$, we create a translation morphism t with `TranslationMorphism.make(`M`,` m`)` and then perform d`.map(`t`)`.

To realize more complex transformations we often map some of a denotators morphisms into a more-dimensional space using injection morphisms, transform them in this space and finally map the back into the individual spaces using projection morphisms. This procedure is described mathematically in more detail in Chapter 4. A method often used in this context, is the `cast()` method of the classes `Module` and `ModuleElement`.

2.4.4 Common Musical Forms in Rubato Composer

In RUBATO COMPOSER, there are no restrictions to data types, as long they are expressed in the denotator format. Users can freely extend the form library at runtime. The application may even be used for generating and processing data types other than musical. Nevertheless, for musical applications, it is advantageous to use uniform data types. In this section, we would like to present the forms currently used in RUBATO COMPOSER, which are inspired by mathematical music theory.

Most of the current basic musical forms were already defined above, in Section 2.3.1. They are namely *Score*, *Note*, *Onset*, *Pitch*, *Loudness*, *Duration* and *Voice*. These forms are already sufficient to represent a simple musical piece, e.g. one taken from a MIDI file without controller information. However, for building more complex musical structures, it is inevitable to extend these definitions.

One extension that is very useful in combination with the work of this thesis, is the *MacroScore* form, first described in [Mazz02]. The rubettes for handling macro scores have been implemented by Karim Morsy [Mors]. We will give the definition of *MacroScores* here and briefly discuss their use.

MacroScores

A music score may contain *hierarchical structures*, for example notes embellished with a trill. We can see such a trill as a set of satellite notes defined relatively and hierarchically subordinated to a parent note. Now let us imagine the parent note is transformed mathematically, say rotated by 90°. If the same rotation is applied to the satellite notes, they result in two chords and lose their functionality. With the macro score concept, we apply transformations just on the top level of a hierarchical structure. Like this, every subordinated trill note in our example maintains its relative position in time after its base note. Here is the definition of the recursive *MacroScore* form:

$$\begin{aligned} \mathit{MacroScore} &: \mathit{Id}.\mathbf{Power}(\mathit{Node}) \\ \mathit{Node} &: \mathit{Id}.\mathbf{Limit}(\mathit{Note}, \mathit{MacroScore}) \end{aligned}$$

With this recursive definition, it is possible to define structures with any hierarchical depth, i.e. trill notes can be embellished with trills themselves and so on. To describe the transformation of macro scores, given a morphism $f : \mathit{Note} \to \mathit{Note}$ and a denotator $d : A@\mathit{MacroScore}(c_1, \ldots c_n)$ with coordinates $c_i : A@\mathit{Node}(\mathit{Note}_{c_i}, M_{c_i})$, we define a morphism Macro_f :

MacroScore → *MacroScore* as follows:

$$Macro_f(d) = \{(f(Note_{c_i}), M_{c_i}) \mid c_i \in d\},$$

The morphism f is just applied to the top level notes $Note_{c_i}$ of the *MacroScore*, leaving their satellites in M_{c_i} unchanged. If we wish to transform satellites on a specific level, we simply apply $Macro_f$ to the *MacroScore* of their direct parent note.

To process macro scores further, such as playing the music they contain, it is often necessary to convert them into a *Score* denotator. Related to this, a special method called *flattening* is used. It creates a *MacroScore* denotator from another one by collapsing a number of hierarchical levels. For example, if we perform a one-level flattening, the second level satellite notes are integrated into the top-level *MacroScore* denotator, which leads to a hierarchy with one level less.

The conversion from *MacroScore* to *Score* is then executed as a *lossy conversion*, where a *Score* denotator is created just containing the top-level notes, i.e. cutting off all satellites.

RUBATO COMPOSER features rubettes for converting scores to macro scores and vice versa, for selecting macro substructures and for flattening structures. For more information about the conversion methods and the macro score rubettes, we refer the reader to the work of [Mors].

All of the musical examples (see Chapter 6) in this thesis are realized with scores or macro scores. However, in the near future RUBATO COMPOSER's musical form library will certainly be extended over and over and it is one of the major tasks for programmers to design their rubettes in such a way that they are able to handle denotators of preferably any possible forms.

Chapter 3

The Concepts of *presto* OrnaMagic

presto provides the facility to build grid constructions of musical elements and to use them in different application contexts, which is the subject of this thesis. However, *presto*'s musical operations are restricted to two-dimensional geometry, as we will see in the next chapter. The module performing these tasks is called OrnaMagic. Here, we will describe its underlying concepts.

3.1 Ornaments

The functionality of the OrnaMagic module is based on the creation and application of ornaments. An *ornament* is defined to be a periodic musical construction, as shown in Figure 3.1. It is characterized by three attributes:

- a *motif*, the recurring set of notes,
- a *grid* of parallelogram-shaped cells, defined by two vectors v, w in the score view plane
- and a *range*, defined by two integer intervals r_v and r_w, one for each of the two vectors, where $r_i = [a_i, b_i] \subset \mathbb{Z}$ and $a_i \leq b_i$.

Given the range, we obtain a set of displacement vectors V used to build the ornament as follows:

$$V = \{n_v \cdot v + n_w \cdot w \mid a_v \leq n_v \leq b_v, a_w \leq n_w \leq b_w\}$$

We then create the ornament by gathering all the copies of the motif generated by displacing the motif along the translation vectors in V. The

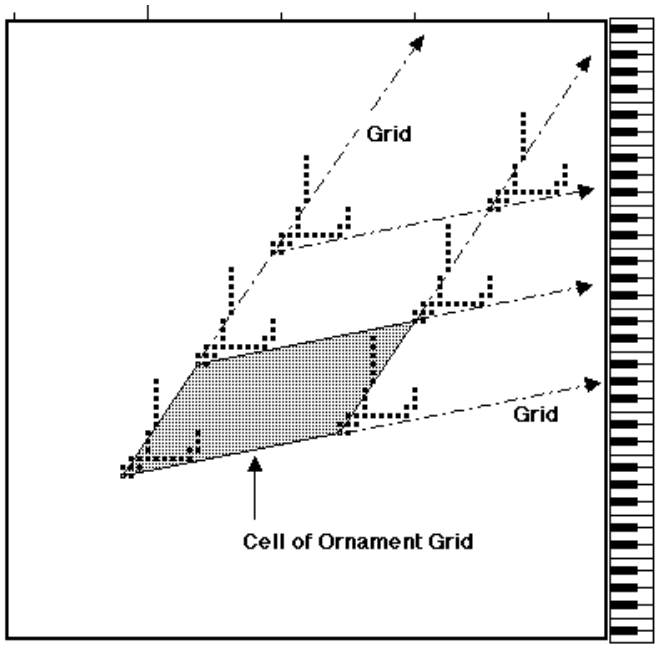


Figure 3.1: A *presto* ornament with ranges $[0, 1]$ and $[0, 2]$.

ornament therefore contains $(b_v - a_v + 1) \cdot (b_w - a_w + 1)$ copies of the motif.

In OrnaMagic, ornaments can be defined for the notes in a local score view, which is a selection of notes in the global score. The onset coordinate is given to be one of the two visible dimensions, so ornaments can be created for the coordinate pairs onset/pitch, onset/duration or onset/loudness.

3.2 Applications of Ornaments

After the creation of an ornament using OrnaMagic, it has to be applied to the global score. For this, there are three different ways. The first two are inspired by set theory. If *union* is selected, the ornaments' notes are added entirely to the global score. With *intersection*, all notes that are not present in the ornament are removed from the global score. The third way, *alteration* is a bit more sophisticated and will be described in the next section.

3.2.1 Alteration

Alteration (also *tonal alteration*) is a common expression in music, used to describe a small change in a notes' pitch (♭, ♯). However, in *presto* it is

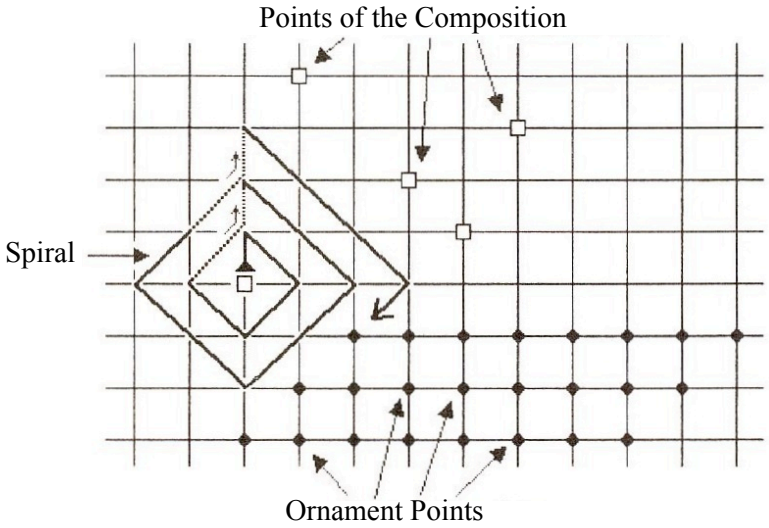


Figure 3.2: *presto* OrnaMagic's nearest neighbor algorithm.

used in a more general way, describing a two-dimensional process. During alteration, the notes in the global score are altered towards the notes of a specified ornament, like in a force field. In fact, each note of the composition is attracted or repulsed by its nearest neighbor in the ornament.

The algorithm used to detect this nearest neighbor works as shown in Figure 3.2. It starts at the position of the compositions note and iterates in a *spiral movement* around it, until an ornament note is found. If no neighbor is found after 32 rotations, the compositions' note remains unchanged. Otherwise, it is *moved towards the neighbor* by a specified percentage of their difference. The starting direction of the spiral can be chosen and therefore the direction in which notes move if there are several neighbors at minimal distance.

If the *percentage* of the difference is 100%, the note is moved exactly to the position of its nearest neighbor. If it is negative, the note is moved in the opposite direction. Additionally, different percentages can be defined for the beginning and the end of the global score. For every position in time inbetween, it is then interpolated linearly. So virtually, the strength of the alteration field may change continuously. For example, if it is chosen to be 45% to -65%, it will be -10% in the middle of the area.

There is also the possibility to define a *rail* to control the direction in which notes move: vertically, horizontally or diagonally. A rail is a straight line that passes through the original note. As soon as the moved note is determined, it is projected on this rail. If for example we decide to perform an alteration on the onset/pitch plane and just the pitch of the notes should

be changed, we use the vertical rail.

3.3 Examples

The OrnaMagic module can be used in many interesting ways. The examples we present in this section, were altogether featured by a simplified *presto* module named *Easy Ornaments*, which made the rather complex use of ornaments accessible in a easy way. For simplicity, for all examples we select the onset/pitch plane.

3.3.1 Looping

Our first example is very simple and does not use alteration. We define a motif with length l in time and define $v = (l, 0), w = (0, 0), r_v = [0, 4]$ and $r_w = [0, 0]$. Then, we execute the ornament in union mode and obtain five copies of the ornament aligned successively in time. This is equal to looping the given motif five times.

3.3.2 Tonal Alteration

As previously mentioned, OrnaMagic's alteration concept is a generalization of tonal alteration. Of course, it is also possible to use the module simply for that. Assume we would like to transform a melody with length l in C major to a melody in C melodic minor. For this, the ornament has to be defined as follows:

We take one octave of the C melodic minor scale as the motif. The grid's vectors are $v = (1, 0)$ and $w = (0, 12)$, to cover all onsets and octaves. We finally define the ranges: $r_v = [0, l]$ to reach all possible onsets of the motif and $r_w = [-6, 6]$ to reach all possible octaves in *presto*. Figure 3.3 shows how the created ornament looks.

Then, we apply the ornament to the melody using alteration of 100%. Here, it makes a difference, which initial direction of the spiral algorithm is selected, because the distances between the ornaments' notes are very short. We have to select 'down' to cause all notes to move downwards to the next note in the minor scale (in fact, just the third, which would eventually be moved the fourth, otherwise). It is also important to use the vertical rail, so that the melody's onsets are not changed.

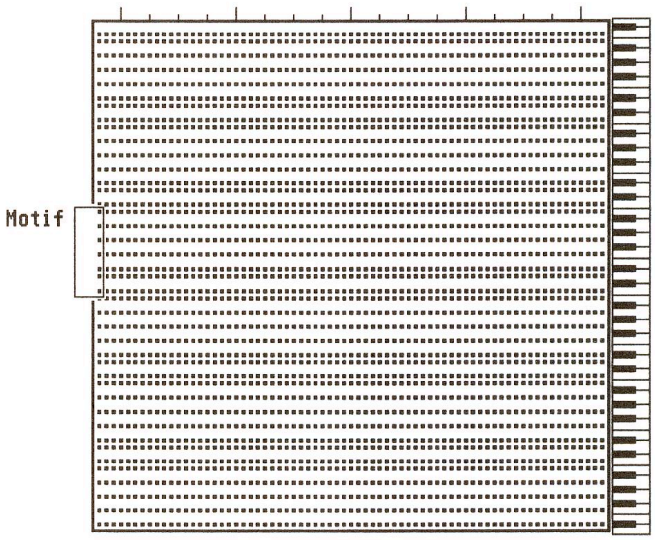


Figure 3.3: An ornament with the C melodic minor scale as a motif.

3.3.3 Quantizing

The third example is very well known in music production. When for example imprecisely recorded MIDI notes are aligned in a raster of note values, e.g. quarter triplets, one speaks of quantizing. Music software products like Logic [Logic] or Cubase [Cubase] offer such quantize functions. However, with OrnaMagic, quantizing can be customized more precisely.

Our raster can be any ornament, for example a grid of sixteenth notes. For this, we define one single note with onset 1 and duration $\frac{1}{16}$ (actually, this may be 2 or 4 in *presto* values) as our motif. Also the ornament's grid vectors are then given the length of $\frac{1}{16}$, thus $v = (\frac{1}{16}, 0)$ and $w = (0, \frac{1}{16})$. The ranges are chosen to be as large as necessary to build an ornament with the size of the composition in the global score that has to be quantized.

Finally, we execute the ornament with alteration using the horizontal rail, for the pitch must not be modified. If we choose a smaller percentage than 100% for alteration, this causes our composition's notes to be more precise, but still not exactly in the raster. For a negative percentage, the notes get even more imprecise.

Chapter 4

Generalization of OrnaMagic's Concepts

Before the concepts of OrnaMagic can be implemented in Rubato Composer, they need to be formalized for functorial mathematical music theory, i.e. described in the form/denotator representation. Due to the higher level of abstraction associated with this new theoretical background, they must also be generalized. The advantages of this generalization will be a broader field of application, i.e. more complex musical objects or even non-musical ones.

In this chapter, we first identify the limitations of OrnaMagic and outline the improvements, that our new implementation offers. Subsequently, we provide mathematical definitions of the generalized concepts and functionality.

4.1 OrnaMagic's Limitations

The possibilities of *presto* and OrnaMagic are limited in several aspects. First of all, OrnaMagic is a module combining several independent operations. Alteration for example, is a self-contained function, which does not have to be applied in combination with ornaments, as in OrnaMagic. With Rubato Composer we are able to split up these functional entities into different rubettes, which we do in the next section.

Second, in *presto* everything is restricted to *two-dimensional* geometry. As mentioned before, every thinkable higher-dimensional transformation or ornament can be represented as combination of such operations, but in practice it can be very complicated to think in such two-dimensional factor transfor-

mations and factor ornaments. However, for our Rubato Composer implementation, we will have to define the OrnaMagic operations to work with any higher-dimensional transformations, which will simplify the definition of complex actions and allow non-geometrical uses.

A third limitation of *presto* is the use of integer numbers to define a note's value, that is for example, users have to do without microtonality, since pitch only can have integer values. In Rubato Composer, data types are not fixed, they can be changed even by runtime. Furthermore, their values can be elements of any possible module, such as $\mathbb{R}$ or $\mathbb{Q}$.

The last important restriction is that for the definition of the grid of an OrnaMagic ornament, only *translation* vectors can be used, i.e. a motif cannot appear rotated or sheared within an ornament. Our new definition, named 'wallpaper' to avoid confounding with ornaments as musical embellishments, will feature grids of any number of arbitrary transformations.

In these premises, we now first analyze OrnaMagic's functionality and define a number of rubettes taking on the identified tasks and then give the necessary mathematical definitions for the new Rubato Composer implementation.

4.2 New Distribution of Tasks

The first step in our generalization process is the identification of *independent subtasks* in the overall functionality of the OrnaMagic module and the distribution of the corresponding responsibilities on a number of new modules. First of all, it is indisputable that the creation of ornaments and their application can be performed by different modules. With a separation of these tasks, the application possibilities can as well be used separately and in other contexts and ornaments can be produced without needing to be applied to a score immediately.

With this abstraction, we obtain four tasks, one for ornament generation and three for the different application tasks, namely union, intersection and alteration. In Rubato Composer, a convenient module named *Set rubette* has already been implemented, which is able to perform all kinds of set operations on **Power** denotators. Thus, the responsibilities of union and intersection can be delegated to this rubette.

The remaining two tasks of ornament generation and alteration are both self-contained and not yet implemented. They are assigned to two new rubettes, the *Wallpaper rubette* and the *Alteration rubette*, the main contribution of this thesis.

Additionally, for the recreation of the OrnaMagic examples described in section 3.3, three simple auxiliary rubettes have been defined, namely the *Melody rubette*, the *Rhythmize rubette* and the *Scale rubette*. They complement the functionality of the Wallpaper and Alteration rubettes in such a way that creating compositions similar to the ones that can be created using OrnaMagic becomes possible. Their mathematical background is also described in this chapter.

4.3 Wallpapers

A wallpaper is a general structure similar to OrnaMagic's musical structure of an ornament described in Chapter 3, Section 3.1. In music, wallpapers are usually built from *Score* and *MacroScore* forms (see Chapter 2, Section 2.4.4). However, as we will see in the examples of Chapter 6, it makes sense to define a wallpaper for any form of type **Power**.

First, we provide a definition for *MacroScore* wallpapers, which we already described in our previous work [Mazz06]. Then after defining some prerequisites, its generalization for **Power** forms is presented in Section 4.3.4.

4.3.1 MacroScore Wallpapers

A *MacroScore* wallpaper is characterized by:

- a *motif* $d : A@MacroScore(c_1, \ldots c_n)$ with n *Node* coordinates $c_1, \ldots c_n$,
- a *grid* spanned by m morphisms $f. = f_1, \ldots f_m : Note \rightarrow Note$ on the *Note* form,
- and a sequence $r.$ of *ranges* $r_i = [a_i, b_i] \subset \mathbb{Z}$, $a_i \leq b_i$, for $i = 1, \ldots m$.

We then define the corresponding wallpaper as

$$\hat{w}(d, f., r.) = \bigcup_{k_i \in r_i} Macro_{f_1^{k_1} \circ \ldots f_m^{k_m}}(d). \tag{4.1}$$

A wallpaper according to this definition is more general than a *presto* ornament in several points and fulfills many of the expectations described in the previous section:

1. the involved denotators can have general adresses,
2. the grid is composed of functions instead of translations,
3. these functions can transform any of the *Note* coordinates, not just *Onset* and a second,

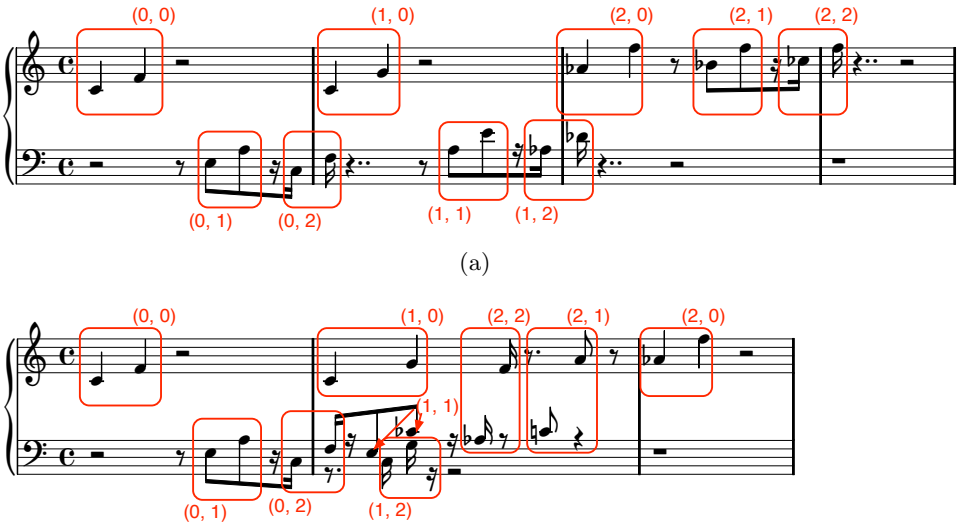


Figure 4.1: Two simple musical wallpapers generated from the two-note motif c'-f', by commuting two morphisms f_1 and f_2, $r_1 = [0, 2], r_2 = [0, 2]$. (a) $f_1 \circ f_2$ (b) $f_2 \circ f_1$. (k_1, k_2) with $k_i \in r_i$ are the grid coordinates of the marked note groups.

4. any number of functions can be used for the grid, not just two.

Now note that it is important to be aware of the order in which the morphisms $f_1 \dots f_n$ are composed, since their commutations generally result in different wallpapers. Figure 4.1 shows an example of two musical wallpapers generated from the same motif with the same pair of affine morphisms $f_1, f_2 : \mathit{Note} \to \mathit{Note}$, where f_1 adds the double of onset to pitch and adds 4 to onset and f_2 halves onset and duration, subtracts the eight-fold duration from pitch and finally adds 2.5 to onset.

For each of the wallpapers, the two morphisms were composed in a different order for each, that is $\mathit{Macro}_{f_1 \circ f_2}$ was used for the top wallpaper and $\mathit{Macro}_{f_2 \circ f_1}$ for the bottom wallpaper. A schematic representation of the two diagrams corresponding to these two wallpapers is shown in Figure 4.2. They are grid diagrams on the *Note* space, spanned by the arrows f_1, f_2.

Before generalizing wallpapers for **Power** forms, we describe in detail two problems arising when denotators of arbitrary **Power** forms are mapped and present our solutions to these very problems.

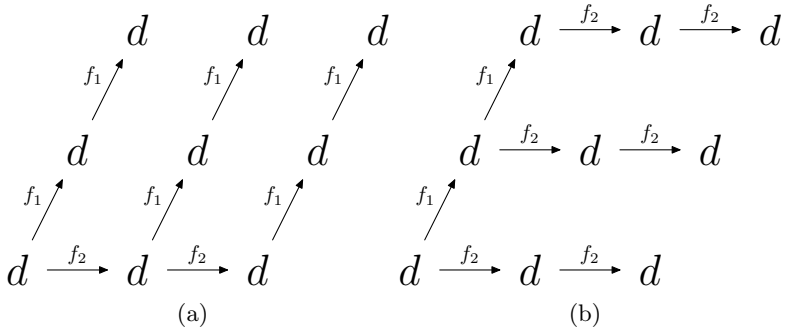


Figure 4.2: The grid diagrams for the musical wallpapers in Figure 4.1

4.3.2 Dealing with Multiply Positioned Forms

The first problem is, that denotators of a composite form F may contain several denotators of a specific **Simple** form S in their structural tree. In this case, we name S *multiply positioned* in F, or else *well-positioned*. The forms having multiply positioned subforms form a superset of the recursive forms. For example, a denotator of the recursive form *MacroScore* contains an *Onset* denotator on each recursion level and therefore *Onset* is multiply positioned in *MacroScore*. It is essential to define for a morphism, which denotators of a multiply positioned **Simple** form it processes.

In the above definition of a wallpaper (Equation 4.1), the morphism $Macro_f$ is used, so that the morphism $f : Note \rightarrow Note$ just transforms the notes at top recursion level of a *MacroScore* denotator (the *Note* denotators in the first level *Nodes*). In our generalization, the morphism f is be more flexible: it virtually maps from one arbitrary product of **Simple** forms to another (described in next section). It is now necessary to find a morphism similar to $Macro_f$ but more general, which transforms *MacroScores* as defined, but also *Scores* and denotators of all other **Power** forms, containing multiply positioned forms or not, in a useful way.

The idea is to create a morphism $First_f : F \rightarrow F$ for a form F, which uses a search algorithm to determine the **Simple** denotators to be mapped by morphism f. This morphism makes sure, that for every **Simple** form in the domain or codomain of f, the top form (with the shortest path) of this type found in the structural tree of the form F is comprised in the transformation. For the realization of this morphism, we first define the set of **Simple** subforms S_F of F, which contains all **Simple** forms present in F's structural tree, e.g. for $F = Score$, $S_F = \{Onset, Pitch, Loudness, Duration, Voice\}$.

Next, we define a function $SA(S, d) = t$, which outputs a denotator t of **Simple** form $S \in S_F$ for denotator $d : A@F$. Note that such a function is

difficult to realize, since we have no guarantee that a denotator of the specified of form S is indeed contained by d, as already described in Chapter 2, Section 2.3.2. For example, in the case where F contains a **Colimit** subform. However, the method we chose works for many possible form structures and will certainly be improved in the future.

More precisely, $SA(S, d)$ searches the form tree of F for the top level form of type S and returns the first denotator found at this position in the tree of d, i.e. if a **Power** denotator is passed, the path along its first branch is travelled. In practice, if no denotator is present at this position (this may happen when a **Colimit** denotator is passed), SA returns nothing. If at least one **Simple** denotator expected by f is not present in d's subtree, d is simply not mapped. This is described more precisely in Chapter 5.

For now, let us assume a t is always found. This is the case for *MacroScore* and *Score* forms, which is what is at least required. For a *MacroScore*, the function ensures that the same **Simple** denotators are transformed as by $Macro_f$. *Score* coordinates just contain one denotator of every **Simple** form, which are therefore the top ones to be found.

For simplicity we will later use $SA(S, d)$ in two ways, sometimes to denominate the denotator t itself, but sometimes also directly for the value morphism of t. Before $First_f$ can be defined, we need to describe the way we treat the more flexible morphism f mentioned above, which we will do in the next section.

4.3.3 Mapping Denotators by Arbitrary Morphisms

Another interesting problem is to find a way to map a **Power** denotator d by any arbitrary morphism f. Also if for instance the codomain modules of d's **Simple** denotator value morphisms do not match f's domain modules. A rubette dealing with this problem offers great flexibility and security, since users can use a specific morphism for different purposes and cannot misapply it in any way.
In order to solve this problem, *casting morphisms* can be used to convert the **Simple** denotator's values so that they can be composed with the morphism. In this section we explain this procedure for a morphism $f : V \to W$ and a denotator $d : A@P(c_1, \ldots c_n)$ of **Power** form P. The morphism's domain $V = V_1 \times \ldots V_s$ and codomain $W = W_1 \times \ldots W_t$ are assumed to be products of arbitrary modules.
We define two sequences of **Simple** forms $G. = (G_1, \ldots G_s), H. = (H_1, \ldots H_t)$ with the elements $G_j, H_k \in S_P$ to specify, which denotators in $c_1, \ldots c_n \in d$ are transformed by f. Moreover, we need three kinds of auxiliary morphisms:

- the *injection* morphisms $i_1, \ldots i_s, i_j : V_j \to V$ with $i_j(v) = v' = (0, \ldots v, \ldots 0)$, where v is at the j-th position of v',
- the *projection* morphisms $p_1, \ldots p_t, p_k : W \to W_k$ with $p_k(w) = w_k$ for $w = (w_1, \ldots w_t)$ and
- the *casting* morphisms $g_1, \ldots g_s$, $g_j : G_j \to V_j$ and $h_1, \ldots h_t$, $h_k : W_k \to H_k$.

In practice, a casting morphism is usually an embedding, if its domain module is contained by its codomain module. Otherwise, it is a rounding morphism. The previously defined morphisms are composed in the following way, to map denotator $c_i \in d$ by morphism f:

$$\phi_f(c_i, (G_j)) = f(i_1 \circ g_1 \circ SA(G_1, c_i) + \ldots i_s \circ g_s \circ SA(G_s, c_i)).$$

Finally, considering the **Simple** forms $G., H.$, we are able to define our general mapping morphism $First_f : P \to P$ for **Power** form P:

$$\begin{aligned} First_f(d) \text{ is a copy of } d, \text{ where every } SA(H_k, c_i) \text{ is replaced} \\ \text{by } h_k \circ p_k \circ \phi_f(c_i, (G_j)) \text{ for } 1 \leq k \leq t \text{ and every } c_i \in d. \end{aligned}$$

4.3.4 General Wallpapers

Using the definition of $First_f$, we are now able to describe a wallpaper for arbitrary **Power** forms. It is composed of

- a *motif* $d : A@P(c_1, \ldots c_n)$ with n coordinates $c_1, \ldots c_n$ of form C, P being any form of type Power,
- a *grid* spanned by the morphisms $f. = (f_1, \ldots f_m)$, $f_i : V^i \to W^i$, $V^i = V_1^i \times \ldots V_s^i$ and $W^i = W_1^i \times \ldots W_t^i$ being products of arbitrary modules,
- and the corresponding *ranges* $r. = (r_1, \ldots r_m) = [a_i, b_i] \subset \mathbb{Z}$, where $a_i \leq b_i$

To be able to use the function $First$, we take one configuration of **Simple** forms $G., H.$ as described above, but this time all from S_P. The general wallpaper is then defined as follows:

$$w(d, f., r.) = \bigcup_{k_i \in r_i} First_{f_1^{k_1}} \circ \ldots First_{f_m^{km}}(d).$$

Note that *First* is not a functor and therefore cannot be applied to a product of f's. In the next section, we give an example to illustrate the above definitions.

4.3.5 Example for Denotator Mapping and Wallpapers

Assume we would like to transform a *Score* denotator m by a morphism $f : \mathbb{R}^2 \to \mathbb{R}^2$ with $f(q,r) = (\frac{1}{10}qr, 2r)$. Additionally, we specify $G_1 = Onset, G_2 = Pitch, H_1 = Onset$ and $H_2 = Loudness$. The necessary auxiliary morphisms are:

$$
\begin{aligned}
&g_1 : \mathbb{R} \to \mathbb{R},\\
&g_2 : \mathbb{Q} \to \mathbb{R},\\
&i_1 : \mathbb{R} \to \mathbb{R}^2 \text{ with } i(r) = (r, 0),\\
&i_2 : \mathbb{R} \to \mathbb{R}^2 \text{ with } i(r) = (0, r),\\
&p_1 : \mathbb{R}^2 \to \mathbb{R} \text{ with } p(q,r) = q,\\
&p_2 : \mathbb{R}^2 \to \mathbb{R} \text{ with } p(q,r) = r,\\
&h_1 : \mathbb{R} \to \mathbb{R} \text{ and}\\
&h_2 : \mathbb{R} \to \mathbb{Z}.
\end{aligned}
$$

g_1 and h_1 are identity morphisms, since G_1 and H_1 have the same module as f's domain and codomain factors. Then, we define the following *Score* denotator m. For simplicity, we just specify *Onset* and *Pitch* denotators, for they are the only **Simple** denotators needed for transformation.

$$
\begin{aligned}
&m : A@Score(\{n_1, n_2, n_3\})\\
&n_1 : A@Note(o_1, p_1, l_1, d_1, v_1)\\
&n_2 : A@Note(o_2, p_2, l_2, d_2, v_2)\\
&n_3 : A@Note(o_3, p_3, l_3, d_3, v_3)\\
&o_1 : A@Onset(1.0), p_1 : A@Pitch(63)\\
&o_2 : A@Onset(2.0), p_2 : A@Pitch(60)\\
&o_3 : A@Onset(2.5), p_3 : A@Pitch(56).
\end{aligned}
$$

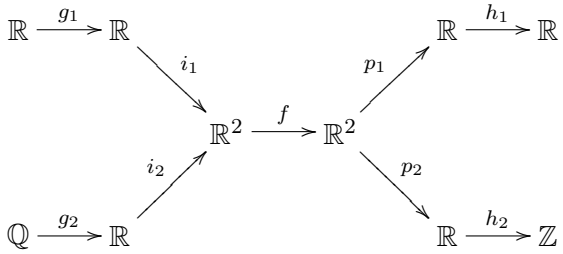
For the note n_i, the function SA returns the following **Simple** denotators:

$$
\begin{aligned}
SA(G_1, n_i) &= o_i\\
SA(G_2, n_i) &= p_i\\
SA(H_1, n_i) &= o_i\\
SA(H_2, n_i) &= l_i.
\end{aligned}
$$

We obtain:

$$
\begin{aligned}
\phi_f(n_i) &= f(i_1 \circ g_1 \circ SA(G_1, n_i) + i_2 \circ g_2 \circ SA(G_2, n_i))\\
&= f(i_1 \circ g_1 \circ o_i + i_2 \circ g_2 \circ p_i) \text{ and}\\
First_f(m)&\{SA(H_k, n_i) = h_k \circ p_j \circ \phi_f(n_i) \mid 1 \leq k \leq t, n_i \in m\}\\
&= \{o_i = h_1 \circ p_1 \circ \phi_f(n_i) \cup l_i = h_2 \circ p_2 \circ \phi_f(n_i) \mid n_i \in m\}.
\end{aligned}
$$

Here is the diagram of the composed function:



We then obtain the following transformed *Score* $m' = First_f(m)$:

$$
\begin{aligned}
&m' : A@Score(\{n'_1, n'_2, n'_3\})\\
&n'_1 : A@Note(o'_1, p_1, l'_1, d_1, v_1)\\
&n'_2 : A@Note(o'_2, p_2, l'_2, d_2, v_2)\\
&n'_3 : A@Note(o'_3, p_3, l'_3, d_3, v_3)\\
&o'_1 : A@Onset(6.3), l'_1 : A@Loudness(126)\\
&o'_2 : A@Onset(12.0), l'_2 : A@Loudness(120)\\
&o'_3 : A@Onset(14.0), l'_3 : A@Loudness(112).
\end{aligned}
$$

If we go one step further and map the resulting denotator, i.e. calculate $m'' = First_f(m') = First_{f^2}(m)$, we get the following result:

$$
\begin{aligned}
&m'' : A@Score(\{n''_1, n''_2, n''_3\})\\
&n''_1 : A@Note(o''_1, p_1, l''_1, d_1, v_1)\\
&n''_2 : A@Note(o''_2, p_2, l''_2, d_2, v_2)\\
&n''_3 : A@Note(o''_3, p_3, l''_3, d_3, v_3)\\
&o''_1 : A@Onset(39.69), l''_1 : A@Loudness(126)\\
&o''_2 : A@Onset(72.0), l''_2 : A@Loudness(120)\\
&o''_3 : A@Onset(78.4), l''_3 : A@Loudness(112).
\end{aligned}
$$

Now let us create a 1-dimensional wallpaper with range $r = [0, 2]$:

$$
\begin{aligned}
w(m, f, r) &= \bigcup_{0 \le k \le 2} First_{f^k}(m)\\
&= First_{f^0}(m) \cup First_{f^1}(m) \cup First_{f^2}(m)\\
&= m \cup m' \cup m''.
\end{aligned}
$$

We finally get a wallpaper with 9 notes:

$$w : A@Score(\{n_1, n_2, n_3, n'_1, n'_2, n'_3, n''_1, n''_2, n''_3\})$$

4.4 Alteration

The second main rubette, the Alteration rubette, provides the functionality for altering one composition to another. In this section, we generalize *presto* OrnaMagic's alteration concepts (see Chapter 3, Section 3.2.1) for **Power** denotators. After the definition of alteration, we reconstruct some of the *presto* examples for better comprehension.

4.4.1 Definition of Alteration

For alteration we need:

- two *compositions* $d_1 : A@P(c_{11}, \ldots c_{1m})$ and $d_2 : A@P(c_{21}, \ldots c_{2n})$, c_{ij} being of form C, P being any form of type **Power**,
- d alteration *dimensions* with:
  - $S. = (S_1, \ldots S_d)$ altered **Simple** forms, $S_i \in S_P$ and $S_i \neq S_j$ for $i \neq j$ and $0 \leq i, j \leq d$,
  - $(a_1, b_1), \ldots (a_d, b_d), a_i, b_i \in \mathbb{R}$ pairs of marginal alteration *degrees* and
  - $R. = (R_1, \ldots R_d)$ **Simple** forms, according to which the local alteration degree is calculated, $R_i \in S_P$ and embedded in $\mathbb{R}$, therefore linearly ordered,
- a *distance* function (a metric) $\delta(c_1, c_2)$ for denotators c_1, c_2 of form C
- and a *nearest neighbor* function $NN(c, d) = n$, for $d : A@P(c_1, \ldots c_k)$ and c of form C so that $n = c_i$, if $\delta(c, c_i) \leq \delta(c, c_j)$ for $j \neq i$, $1 \leq i, j \leq k$.

Now let us describe formally, how alteration works. If we alter composition d_1 towards d_2, we obtain a composition $alt : A@P(e_1, \ldots e_m)$ of form P with $|alt| = |d_1|$. Its elements $e_1, \ldots e_m$ are transformed versions of $c_{11}, \ldots c_{1m}$.

Which **Simple** denotators' values are altered is specified by the d **Simple** forms S_i, each of them standing for an alteration dimension. As with general wallpapers, we need to accept any **Power** form as input composition, also forms, the elements of which may contain several denotators of form S_i in their structural tree. Therefore also here we need to use the function SA (introduced in Section 4.3.2), so that for every S_i and every $c \in d_1$, just one denotator value is altered, namely $SA(S_i, c)$.

The amount by which a coordinate c is altered, is indicated by its *local alteration degree*. For its calculation, first the position of c within the com-

position d_1 according to R_i is determined. R_i is another **Simple** form, which is often the same as S_i and which defines the scale on which the position of c is measured. The position $position_i \in [0,1]$ of c is calculated using the function

$$position_i(c) = \frac{SA(R_i, c) - min_{R_i}}{max_{R_i} - min_{R_i}},$$

where $max_{R_i} = max(\{SA(R_i, c) \mid c \in d_1\})$ and $min_{R_i} = min(\{SA(R_i, c) \mid c \in d_1\})$ are the highest and lowest elements of R_i denotators in d_1. These elements can just be calculated if R_i is embedded in $\mathbb{R}$, which has been assumed above.

Then, the local alteration degree is calculated depending on the marginal alteration degrees (a_i, b_i). These degrees indicate, how much d_1 is altered at its lowest and highest values on the R_i scale, whereby a_i defines how much max_{R_i} is altered and b_i how much min_{R_i}. For $a_i, b_i = 0$, d_1 remains unchanged and for $a_i, b_i = 1$, every element of d_1 obtains the value in d_2 with the smallest distance to it in dimension i, as we will see in the later definitions. If for example we choose $d_1 : A@Score, R_i = Onset, a_i = 0$ and $b_i = 1$ the first *Note* of d_1 in time is not changed at all, whereas the last note obtains a new onset, taken from the note in d_2 that shows the least distance from it. For each note in between, the local alteration degree is calculated using linear interpolation, according to its position in time. We define the function for c_k's the local alteration degree $degree_i(\texttt{SA}(R_i, c_k))$ as follows:

$$degree_i(m) = a_i + (a_i - b_i)position_i(m).$$

The alteration function for dimension i then uses this degree to alterate a morphism m_1 towards another m_2:

$$Alteration_i(m_1, m_2) = m_1 + degree_i(m_1)(m_2 - m_1).$$

We have now made the necessary preparations and are ready to define the altered composition *alt*, which we already mentioned above:

$$alt(d_1, d_2) = \{Alteration(c_k, NN(c_k, d_2)) \mid c_k \in d_1\}.$$

In this definition, we use the morphism $Alteration : C \times C \to C$, where $Alteration(c_1, c_2)$ returns a copy of c_1, in which the value of the **Simple** denotator $SA(S_i, c_1)$ is replaced by the altered morphism $Alteration_i(SA(S_i, c_1), SA(S_i, c_2))$ for every $1 \leq i \leq d$.

4.4.2 Nearest Neighbor Search

As described in the previous section, to identify the target for alteration, a nearest neighbor function *NN* is used. In *presto*, nearest neighbor search

(NNS) is realized using the two-dimensional spiral search algorithm (see Chapter 3, Section 3.2.1). This is possible as *presto* deals with discrete values, i.e. the number of coordinate points visited and checked during nearest neighbor search is always finite. However, this is not the case in our actual RUBATO COMPOSER implementation, since **Simple** denotators accept values from continuous modules. We therefore chose to use a common method for NNS, the *kd-tree* method [Bent75].

The kd-tree is a multidimensional binary search tree, which organizes points of a k-dimensional space in a way that allows NNS within a time complexity of $O(\log n)$. Points can be added and removed and from these points, the one with the smallest distance from an arbitrary input point can be determined. The distance function δ normally used for space-partitioning in kd-trees is the *Euclidean distance* $\delta(A, B) = \sqrt{\sum_{i=0}^{n} (a_i - b_i)^2}$ for $A = (a_1, \dots a_n), B = (b_1, \dots b_n)$.

To build a preferably efficient kd-tree for the elements of an arbitrary **Power** denotator, we decided to cast the value of each relevant **Simple** denotator in their structural tree to the module of real numbers $\mathbb{R}$. To achieve this, we again use casting morphisms, as introduced in Section 4.3.3. We define a point $x_k = (x_{k1}, \dots x_{kd}) \in \mathbb{R}^d$ for every $c_k \in d_2$, with $x_{ki} = g_i(SA(S_i, c_k))$, using the casting morphism $g_i : S_i \to \mathbb{R}$ and add all x_k to the kd-tree. This is possible due to the assumption in the previous section, that the R_i are embedded in $\mathbb{R}$. During NNS, the Euclidean distance δ can then be calculated within $\mathbb{R}^d$. Note that this NNS just considers the values of the denotators $SA(S_i, c_k)$. If values of other denotators should be respected in NNS but no morphing should take place in their direction, it is recommended to define a dimension h with $(a_h, b_h) = (0, 0)$. A few special cases of alteration (some of them using this method) and a thorough example will be discussed in the next sections. More details about the implementation of NNS will be given in Chapter 5.

4.4.3 Imitation of the OrnaMagic rails

In *presto*, the direction of alteration can be limited by selecting one of four different rails, as described in Chapter 3, Section 3.2.1. With our definition of alteration, two of these rails can be imitated. We assume a common two-dimensional *presto* alteration space of $Onset \times Pitch$ and therefore define two alteration dimensions with $S_1 = Onset, S_2 = Pitch, R_1 = Onset$ and $R_2 = Onset$. For the two rails we need the following restrictions:

- horizontal rail: $(a_2, b_2) = (0, 0)$
- vertical rail: $(a_1, b_1) = (0, 0)$

There is no way to simulate the two diagonal rails with our present definitions. They would have to be extended for this.

4.4.4 Alteration Examples

We give two examples for alteration, both of them inspired by *presto*'s Easy Ornaments module described in Chapter 3, Section 3.3. For *tonal alteration* increasing in time, we choose d_1 to be a composition and d_2 to be a musical scale, the generation of which we describe later, and perform a one-dimensional alteration with:

$$\begin{aligned} S_1 &= \mathit{Pitch}, \\ (a_1, b_1) &= (0, 1) \text{ and} \\ R_1 &= \mathit{Onset}. \end{aligned}$$

For *quantizing* decreasing in time, we need composition d_2 to be a rhythmical grid of the same length in time as d_1. Such a grid can be created using a wallpaper. Furthermore, we define the following one-dimensional alteration:

$$\begin{aligned} S_1 &= \mathit{Onset}, \\ (a_1, b_1) &= (1, 0) \text{ and} \\ R_1 &= \mathit{Onset}. \end{aligned}$$

4.5 Melodies, Rhythms and Scales

All three auxiliary rubettes written for this thesis, output basic musical elements such as motifs and scales in form of *Score* denotators. If *MacroScores* are needed, the rubettes can easily be combined with the ScoreToMacro rubette (see [Mors]). Most of the examples in this thesis (see Chapter 6) have been created utilizing these rubettes.

4.5.1 Melody Rubette

The Melody rubette is used to generate simple random melodies fulfilling specific criteria. The generation procedure is similar to Karlheinz Essl's random generator 'brownian' [Essl96], although some additional restrictions can be defined. A random melody can for example be used as a motif for the Wallpaper rubette and consists of n notes on a scale with step size $s \in \mathbb{Q}$ and starting pitch $p_{start} \in \mathbb{Q}$. In Rubato Composer, pitch is commonly

interpreted as MIDI pitch within a range of 0 to 127, where every integer represents a semitone. So for example the current ScorePlay rubette plays only notes on the chromatic scale. However, the Melody rubette has microtonal capabilities, e.g. for $s = \frac{1}{2}$, in MIDI, the steps consist of quarter tones.

During creation of a melody, the following boundaries are respected:

- $p = (p_{min}, p_{max}) \in \mathbb{Q}^2, p_{min} \leq p_{max}$, the pitch boundaries,
- $u = (u_{min}, u_{max}) \in \mathbb{N}^2, 0 \leq u_{min} \leq u_{max}$, the interval up boundaries and
- $d = (d_{min}, d_{max}) \in \mathbb{N}^2, 0 \leq d_{min} \leq d_{max}$, the interval down boundaries,

where the interval boundaries are expressed in number of steps of size s. From these values, a melody $m : 0@Score(c_1, \ldots, c_n)$ is generated based on the following pitch sequence:

$$p_1 = p_{start}$$
$$p_{k+1} = p_k + i_k,$$

where $i_k \in \mathbb{Z}$ is an interval randomly selected from the subsequence $\hat{a}.$ of $a. = -d_{max}, \ldots - d_{min}, u_{min}, \ldots u_{max}$, which contains only the elements a_j with $p_{min} \leq p_k + a_j \leq p_{max}$. The random selections from $\hat{a}.$ are equally distributed.

The k-th note c_k of the melody is then defined as follows:

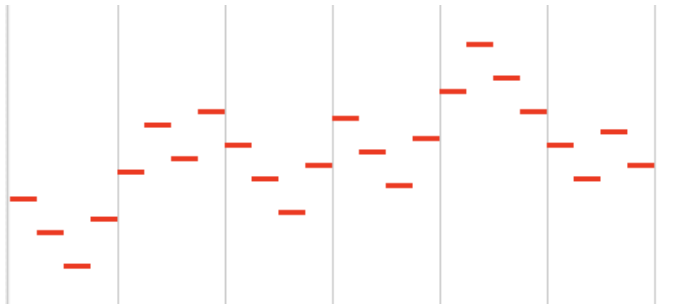
$$c_k : 0@Note(0@Onset(k), 0@Pitch(p_k), \ldots).$$

The remaining three coordinates obtain standard values: loudness is set to 120, duration to 1.0 and voice to 0.

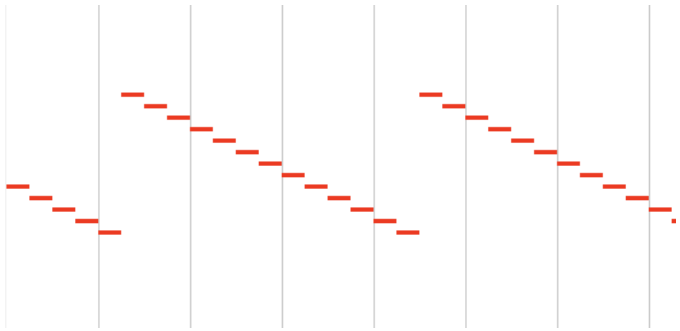
The interval boundaries are very useful to obtain melodies of a certain linearity. Furthermore, they can be used to build melodies of specific patterns, e.g. a melody with $s = 1, u = (7, 7), d = (5, 5)$ travels the circle of fifths while moving up and down randomly (see Figure 4.3 (a)) or $s = 1, p = (60, 72), u = (12, 12), d = (1, 1)$ builds a melody with a sawtooth-wave-like [Puck07] structure (Figure 4.3 (b)). Figure 4.3 (c) shows an example of a random melody generated with $n = 10, p = (0, 127), u = (3, 6), d = (1, 5)$.

4.5.2 Rhythmize Rubette

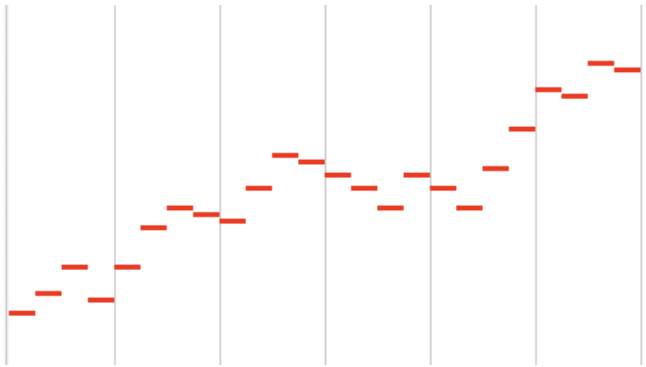
It is much easier to recognize melodic motifs in a composition acoustically, when these motifs have conspicuous rhythmical characteristics. So the task



(a)



(b)



(c)

Figure 4.3: Three example melodies generated by the Melody rubette.

of the second helper rubette, the Rhythmize rubette is to provide a composition with a new rhythm, while respecting a global meter. According to [Coop60], *rhythm* is defined as "the way in which one or more unaccented beats are grouped in relation to an accented one" and *meter* as "the number of pulses between the more or less regularly recurring accents". Inspired by these definitions, the Rhythmize rubette primarily changes onset and duration of the input composition's notes and optionally their loudness, if a meter or emphasizing period is specified.

The output of the Rhythmize rubette is monophonic, i.e. one note at a time, and therefore it can best be used to rhythmize monophonic material, e.g. melodies generated by the Melody rubette. The rubette generates a sequence of notes $s' : 0@Score$, where all notes of the input composition $s : 0@Score(c_1, \ldots, c_m)$ are aligned according to their playing order (see [Mazz02], Section 6.8 for more about the ordering of denotators), where every subsequent note starts, when its preceding note stopped. Their durations are calculated based on very simple rules, which we explain here.

In music, note values (durations) are often multiples of each other, e.g. a sixteenth note has half the value of an eighth note. We therefore define a base $b \in \mathbb{R}$, on which the different possible note values are calculated exponentially. To restrict the possible note values, we specify the number of exponents n and a maximum exponent value $x \in \mathbb{Z}$. We then get $e_1, \ldots e_n$, the sequence of possible exponents, where $e_j = x - j + 1$ and the sequence of note values $v_1, \ldots v_n$ with $v_j = b^{e_j}$. For example, we assume $b = 2, n = 4$ and $x = 0$ and we get $v_1 = 1, v_2 = \frac{1}{2}, v_3 = \frac{1}{4}, v_4 = \frac{1}{8}$, the most common note values in music, from the whole note to the eighth note.

For each note c_k of the input composition s, a duration d_k is randomly selected from the possible note values v_j with equal selection probability for each value. The onset o_k of the same note is then calculated with the following sequence:

$$o_1 = 0$$
$$o_{k+1} = o_k + d_k,$$

Furthermore, as described above, some of the notes can be emphasized. For this, we specify a meter $m \in \mathbb{R}, m \geq 0$, an accented loudness $l_a \in \mathbb{Z}$ and an unaccented loudness $l_u \in \mathbb{Z}$. If $m > 0$, a mechanism is used for guaranteeing, that for each meter point, i.e. every non-negative integer multiple am of m, where $a \in \mathbb{N}_0$, there is a note $c' \in s'$ with onset am. We therefore define an adjusted duration $\hat{d}_k$:

$$\hat{d}_k = \begin{cases} am - o_k, \text{ if } \exists a \in \mathbb{N}_0 \text{ with } o_k < am < o_k + d_k \\ d_k, \text{ otherwise.} \end{cases}$$

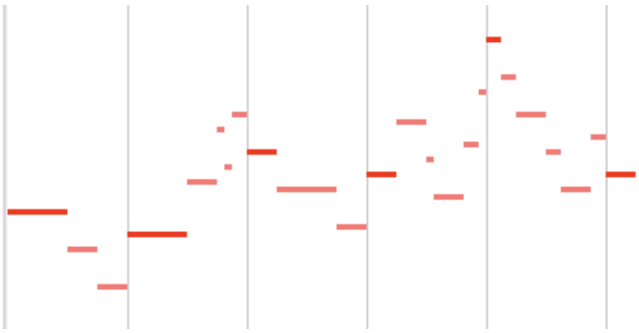


Figure 4.4: The example melody of Figure 4.3 (a) processed by the Rhythmize rubette.

If we use $\hat{d}_k$ instead of d_k for calculating o_{k+1}, whenever the first case of $\hat{d}_k$'s definition occurs, the next onset will be $o_{k+1} = o_k + \hat{d}_k = o_k + am - o_k = am$, which is what we wanted.

Finally, we define the loudness l_k of the k-th note in s':

$$l_k = \begin{cases} l_a, \text{ if } \exists a \in \mathbb{N}_0 \text{ with } \hat{o}_k = am \\ l_u, \text{ otherwise,} \end{cases}$$

which means that the notes on the meter points am are emphasized. We obtain the new note c'_k

$$c'_k : \begin{cases} 0@Note(0@Onset(o_k), \ldots 0@Duration(d_k), \ldots), \text{ if } m = 0 \\ 0@Note(0@Onset(o_k), \ldots 0@Loudness(l_k), 0@Duration(\hat{d}_k), \ldots), \\ \quad \text{otherwise.} \end{cases}$$

Figure 4.4 shows an example for the rhythmization of the melody shown in Figure 4.3 (a). The specified parameters are $b = 2, n = 4, x = 1, m = 4, l_a = 120$ and $l_u = 80$.

4.5.3 Scale Rubette

Above, we mentioned the example of tonal alteration, which can be realized using the Alteration rubette. For this purpose, we need to specify a scale, towards which we would like to alterate tonally. Such a scale can be generated using the Scale rubette.

In his renowned work, Thesaurus of Scales and Melodic Patterns [Slon47], Nicolas Slonimsky defines a musical scale as "a progression, either diatonic

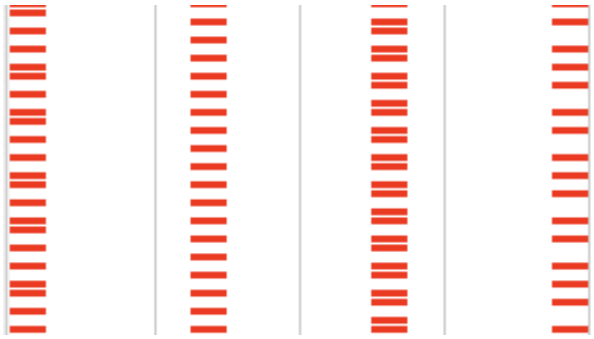


Figure 4.5: An excerpt showing four scales (ionian, whole tone, diminished and minor pentatonic) individually displaced using the ModuleMap rubette and united using the Set rubette.

or chromatic, that proceeds uniformly in one direction, ascending or descending, until the termination point is reached". In our context, the order of progression is not relevant, i.e. the onset of the involved notes can be ignored. We define a scale in a more abstract way as a sequence of n positive intervals $i. = i_1, \ldots i_n \in \mathbb{R}_+$, where the value of 1 is commonly interpreted as a semitone, as described above. Furthermore, we define a pair $(p_{min}, p_{max}) \in \mathbb{Q}^2$, which denotes the pitch range of the output scale, and $p_{root} \in \mathbb{Q}$ with $p_{min} \leq p_{root} \leq p_{max}$, the root note to which the intervals are seen relatively.

We then generate the pitches for the coordinates c_j of the output denotator $s : 0@Score(c_1, \ldots, c_m)$ as follows:

$$\begin{aligned} p_0 &= p_{root} \\ p_{k+1} &= p_k + i_{(k \mod n)+1} \\ p_{k-1} &= p_k - i_{(n-k-1) \mod n}, \end{aligned}$$

where the p_{k+1} are calculated as long they do not exceed p_{max}, whereas the p_{k-1} are calculated as long as they are not smaller than p_{min}. For each p_l generated like this, we then define a note $0@Note$ with pitch $0@Pitch(p_l)$ and add it to the output scale s. Again, the other coordinates of the hereby defined notes, are standard values, i.e. onset is set to 0, loudness to 120, duration to 1.0 and voice to 0. Figure 4.5 shows a score with some example scales, displaced in time. The intervals for the whole minor pentatonic scale, for example, are $i. = 3, 2, 2, 3, 2$. The Scale rubette is designed in such a way that a list of scales can be predefined so that its elements can be selected by runtime as well as any scale can be defined spontaneously in the rubettes properties (see Appendix A, Section A.1). Furthermore, for simplicity, the above specified (p_{min}, p_{max}) are currently not changeable by the user.

Chapter 5

Design and Implementation for Rubato Composer

The generalized concepts of our work have now been formalized to the point where their implementation can proceed in a straightforward way. As described in Chapter 2, Section 2.4.3, the Rubato Composer framework provides all necessary classes and methods for modeling mathematical issues. In this chapter, we give an example for this direct translation from formulas to code and explain how such code is wrapped in a Rubato Composer rubette. Apart from this, we discuss the overall implementation structure as well as some selected reusable utility classes which we implemented within the scope of this thesis.

5.1 Programming a Rubette

Initially we briefly describe how mathematical definitions like those in the previous chapter can be translated to rubette code. There are four rules that have been respected for the rubettes in this thesis:

- Given denotators become rubette inputs. The number of inputs can be defined using the `setInCount(int n)` method. To get the input denotators in the `run()` method, the `getInput(int i)` method is used.
- All other given data become rubette properties. They are implemented as a `javax.swing.JComponent` returned by the rubette's method `getProperties()`, so that users can define them at runtime. Additionally, the methods `applyProperties()` and `revertProperties()` have to be implemented. They are used to save and load the properties in the component to and from the rubette's instance variables, which

are read when the rubette is run.

- The calculations are initialed in the rubette's `run()` method.
- The final definition or result becomes the rubette output. This can be specified using the methods `setOutCount(int n)` and `setOutput(int i, Denotator d)`.

All methods mentioned here, are methods of the class `AbstractRubette`. There could also be other translation procedures, for example some of the properties data (point 2) could be wrapped in denotators and instead be defined as an input denotator. This would lead to more flexibility for building networks, but it would probably also require new denotator definitions.

Here is a code excerpt from the Alteration rubette, as a short example for the aplication of these rules.

Listing 5.1: AlterationRubette.java

```
...
public class AlterationRubette extends AbstractRubette
    implements ActionListener, FocusListener {
  ...
  private JPanel propertiesPanel;

  public AlterationRubette() {
    this.setInCount(2); //input compositions d_1 and d_2
    this.setOutCount(1); //altered composition alt(d_1, d_2)
    ...
  }
  ...
  public void run(RunInfo arg0) {
    PowerDenotator input0 = (PowerDenotator) this.getInput(0);
    PowerDenotator input1 = (PowerDenotator) this.getInput(1);
    if (this.inputFormAndDenotatorsValid(input0, input1)) {
      try {
        PowerDenotator output = this.getAlteration(input0, input1);
        this.setOutput(0, output);
      } catch (RubatoException e) {
        e.printStackTrace();
        this.addError("Error during rubette execution.");
      }
    }
  }
  ...
  public JComponent getProperties() {
    if (this.propertiesPanel == null) {
      this.propertiesPanel = new JPanel();
      ... //add all necessary components
    }
    return this.propertiesPanel;
  }
  public boolean applyProperties() {...}
  public void revertProperties() {...}
  ...
}
```

Now here is an example for a translated mathematical expression. It is a method used in the Wallpaper rubette that first creates a casting morphism, as described in Chapter 4, Section 4.3.3, from the codomain *oldC* of the given morphism m to the given codomain *newC*, i.e. the morphism $c : oldC \rightarrow newC$. Then, it returns the composition of the two morphisms $c \circ m$.

Listing 5.2: WallpaperRubette.java

```
...
/*
 * composes the input morphism with a casting morphism for the
 * specified codomain. example: m:Q->Q, c:R, then return m:Q->R
 */
private ModuleMorphism getCastedMorphism(ModuleMorphism m,
      Module newC) throws CompositionException {
  Module oldC = m.getCodomain();
  if (!newC.equals(oldC)) {
    ModuleMorphism castingM = CanonicalMorphism.make(oldC, newC);
    m = castingM.compose(m);
  }
  return m;
}
...
```

During the development of the rubettes for this thesis, an even simpler implementation possibility for rubettes has been proposed and created by Gérard Milmeister. The class `SimpleAbstractRubette` needs less specifications (abstract methods to be implemented) and provides the functionality of automatic generation of the properties component and apply/revert functionality. This possibility has been tried out on two of the new rubettes, the Melody and the Rhythmize rubettes, as they have just a few basic properties. The problem that appeared during these adaptations, is that there is no way to define relations between properties. However, this will be soon implemented.

5.2 Package Structure and Classes

As outlined in the previous chapter, the contribution of this thesis are five plugin software modules called rubettes. Each of these rubettes has a corresponding class extending the class `AbstractRubette`, namely `WallpaperRubette`, `AlterationRubette`, `MelodyRubette`, `RhythmizeRubette` and `ScaleRubette`. As shown below, these classes have been been organized in three packages, separating the rubettes processing or generating *Score* denotators from the **Power** denotator rubettes.

`org.rubato.rubettes.wallpaper`	Wallpaper rubette classes and tests
`org.rubato.rubettes.alteration`	Alteration rubette classes and tests
`org.rubato.rubettes.score`	Additional rubettes and tests
`org.rubato.util`	Shared classes and superclasses

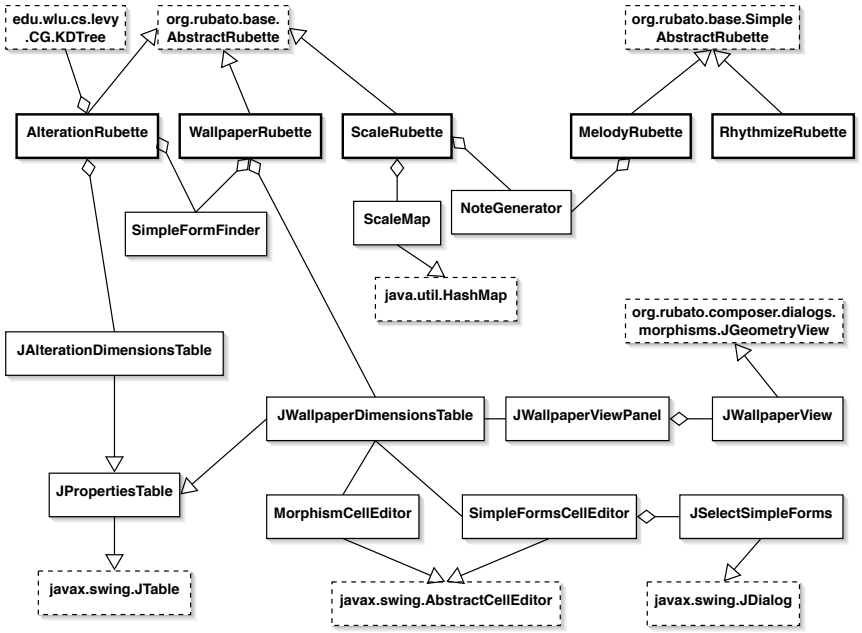


Figure 5.1: A simplified UML diagram without the testing classes.

The fourth package contains classes shared by classes in different packages. Some of them are described in a detailed way in Section 5.4.

5.3 The Rubette Classes

Here is a short overview of every package's contents. The simplified UML diagram in Figure 5.1 visualizes their relationships.

`org.rubato.rubettes.wallpaper`:

`WallpaperRubette`	Main rubette class
`JWallpaperDimensionsTable`	Table in the rubette's properties
`MorphismCellEditor`	Editor for the table's morphism column
`SimpleFormsCellEditor`	Editor for the table's **Simple** forms column
`JSelectSimpleForms`	Dialog opened by `SimpleFormsCellEditor`
`JWallpaperView`	Wallpaper visualization
`JWallpaperViewPanel`	The Wallpaper rubette's view
`WallpaperRubetteTest`	Tests for all the classes in this package

`org.rubato.rubettes.alteration`:

`AlterationRubette`	Main rubette class
`JAlterationDimensionsTable`	Table in the rubette's properties
`AlterationRubetteTest`	Tests for all the classes in this package

`org.rubato.rubettes.score`:

`MelodyRubette`	Melody rubette class
`RhythmizeRubette`	Rhythmize rubette class
`ScaleRubette`	Scale rubette class
`MelodyRubetteTest`	Tests for the Melody rubette
`RhythmizeRubetteTest`	Tests for the Rhythmize rubette
`ScaleRubetteTest`	Tests for the Scale rubette

`org.rubato.util`:

`JPropertiesTable`	Superclass for the W./A. rubette's tables
`SimpleFormFinder`	Finds all forms of type **Simple** in a form tree
`SimpleFormFinderTest`	Tests for SimpleFormFinder
`NoteGenerator`	Easily creates *Note* and *Score* denotators
`ScaleMap`	Holds a number of predefined musical scales
`PerformanceCheck`	Used for rubette performance tests

5.4 Shared and External Classes

In this section, we present some of the classes shared by different rubettes. At the time, they have to be added to the plugin jar file of every rubette using them, so in fact different copies of them are used at the same time. But they may be developed further and added to the Rubato Composer framework someday.

5.4.1 SimpleFormFinder

The most important of the shared classes is `SimpleFormFinder`, which is used by both the Wallpaper rubette and the Alteration rubette. It refers to the function SA and to the set S_F defined in Chapter 2, Section 4.3.2. It is used to search the form tree of a specified form F to find the set S_F of present **Simple** forms and for each $S \in S_F$ the shortest path to the root. It features the following public methods among others:

```
public SimpleFormFinder(Form parentForm) {...}
public void setMaxRecursion(int maxRecursion) {...}
public List<SimpleForm> getSimpleForms() {...}
public List<List<Integer>> getSimpleFormPaths() {...}
...
```

`SimpleFormFinder`'s constructor takes a `Form` as an argument, which is the form the tree of which has to be searched for `SimpleForms`. Furthermore, a maximum recursion level can be defined using the the method `setMaxRecursion(...)`. This defines the depth of the search, which is necessary to define, since else for recursive forms like *MacroScore*, the search would never stop.

As soon as one of these parameters changes, the algorithm searches the whole tree up to the specified depth to find all **Simple** forms present. For every found form, the shortest path to reach it from the root is remembered. These forms and paths can then be fetched by calling the methods `getSimpleForms()` and `getSimpleFormPaths()`.

The found forms can be directly used in the properties windows of the concerning rubettes, where users have to specify one of the subtree forms. With the help of `SimpleFormFinder`, users can be given a list containing just the present forms and therefore cannot make a wrong selection.

The method used for implementation produces the same results as *breadth-first search*, due to the fact that the first and shortest path found for every denotator is kept.

5.4.2 NoteGenerator

`NoteGenerator` is a useful class for generating simple notes melodies and is used by the Melody and the Scale rubette. It provides the following public methods among others:

```
public Denotator createNoteDenotator(double onset,
 double pitch, int loudness, double duration) {...}
public PowerDenotator createSimpleMelody(double noteDistance,
 double...  pitches) {...}
...
```

The first of them is a simple way for generating a zero-addressed denotator of form *Note*. The returned note holds the specified values as well as voice 0.

The second creates a zero-addressed *Score* denotator containing an arbitrary number of notes. As parameters, we need to specify a note distance d and a sequence of pitches $p_0, \ldots p_m$. Every note n_i of the returned score is defined by onset id and pitch p_i, for $0 \leq i \leq m$. The other three coordinate denotators *Loudness, Duration* and *Voice* are set to the standard values 120, 1 and 0.

If a melody needs to be created iteratively in a generative process, the following three methods can be used:

```
public void startNewMelody(double noteDistance) {...}
public void addNoteToMelody(double pitch) {...}
public PowerDenotator createScoreWithMelody() {...}
...
```

The first method starts a new melody for a specified note distance (as above), the second is used to add notes to the melody and the third method can anytime be called to return a *Score* denotator containing the notes defined so far.

5.4.3 ScaleMap

The `ScaleMap` class is a special `java.util.HashMap`, holding a number of predefined musical scales for the use in Rubato Composer. It is currently used by the Scale rubette. In its constructor `ScaleMap()`, scales can be defined as `java.lang.Object[]` records, holding the scale name and an array of halftone steps, for example: {`"ionian", new double[]` {`2,2,1,2,2,2,1`}}. The map is then filled with these records, the names becoming the keys and the steps the values.

`ScaleMap` provides one additional public method to the original `HashMap` methods:

```
public Object[] getScaleNames(int numberOfNotes) {...}.
```

This method is used to get the scale names (keys) for a specified number of notes. To get all keys or a corresponding value, the usual `HashMap` methods `keySet()` or `get()` can be used.

5.4.4 KDTree

Some early experiments were made with a nearest neighbor search implementation using a standard algorithm which calculated the distance to each neighbor and selected the shortest (time complexity $O(n^2)$). We quickly noticed that already for $n = 1000$ (two scores with a thousand notes each), alteration took too long (16 seconds, approximately) and we decided to use a common nearest neighbor algorithm, the *kd*-tree ($O(n \log n)$). The alteration time for $n = 1000$ could be reduced to two seconds.

In our implementation, we reused *kd*-tree java package programmed by Prof. Simon D. Levy of Washington and Lee University, Lexington, Virginia [KDTree], released under the GNU General Public License [GnuGPL]. It consists of the three classes

```
edu.wlu.cs.levy.CG.KDTree
edu.wlu.cs.levy.CG.KeyDuplicateException
edu.wlu.cs.levy.CG.KeySizeException
```

and exactly covers the needs described in Chapter 4, Section 4.4.2. One problem that appeared with the use of *kd*-trees, is that one functionality that was taken over from *presto* cannot be implemented fully anymore. It is the possibility to define in which direction a denotator should altered, if there are several nearest neighbors (described in Chapter 3, Section 3.2.1).

Chapter 6

Practical Examples for Wallpapers and Alterations

In the previous chapters we already showed some outputs of single rubettes like the Melody, Rhythmize or Scale rubette (mostly in piano roll notation). In this chapter we present a few musical and non-musical examples showing the application possibilities of the rubettes generated for this thesis. We would like to focus on designing reasonable rubette networks using the new rubettes in combination with Rubato Composer's built-in rubettes.

6.1 Wallpapers

We start with some examples using the Wallpaper rubette. The first one shows how the two simpler application functions of OrnaMagic, namely union and intersection, can be simulated with Rubato Composer. The second one shows how non-musical denotators can be processed by the Wallpaper rubette. Finally, the latter two are more complex musical examples that we created in the context of this thesis and presented at two conferences in 2006.

6.1.1 Union and Intersection

Two of the three applications of ornaments in OrnaMagic have not been adapted to Rubato Composer for their emulation is possible using two built-in rubettes. Figure 6.1 shows an example of a network, where first an excerpt is selected from an input MIDI file using the *Select2D* rubette,

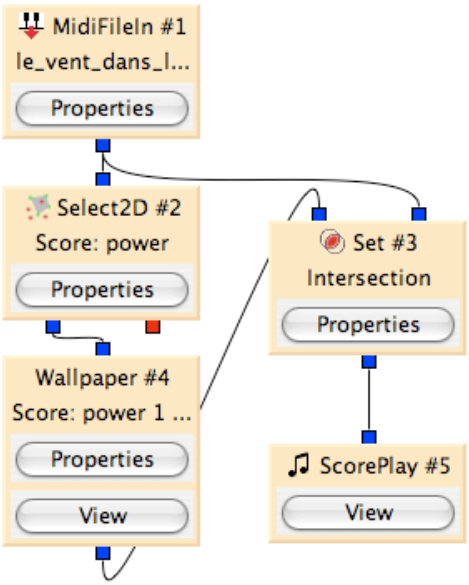


Figure 6.1: The network setup for an emulation of the intersection application.

analogous to a *presto* local score. From this excerpt coming out of the left outlet of the Select2D rubette, a wallpaper is generated and finally, using the *Set* rubette the intersection of the original score and the wallpaper is created and finally passed on to the ScorePlay rubette.

The union application can be realized with the same network, just by selecting the Set rubette's 'Union' option.

6.1.2 Non-Musical Wallpapers

Since the Wallpaper rubette accepts denotators of any form of type **Power**, there are many interesting applications if suitable forms are defined, e.g. for numeric calculations or graphical uses. Here, we give a very simple example using the form *RealSet*, which is predefined in RUBATO COMPOSER. Figure 6.2 shows its network consisting of four rubettes.

The Source rubette feeds a *RealSet* denotator named *reals* with two *Real* coordinates with values 2 and 5 into the network. This denotator is then processed by the Wallpaper rubette and finally visualized using the Display rubette (right part of the picture). For comparison, the original denotator coming from the source is shown using another Display rubette.

The wallpaper's grid is $f. = f_1 : \mathbb{R} \to \mathbb{R}, f_2 : \mathbb{R} \to \mathbb{R}$, with $f_1(r) = 2r - 1$ and $f_2(r) = \frac{1}{2}r + 1$ and the corresponding ranges $r_1 = [0, 1]$ and $r_2 =$

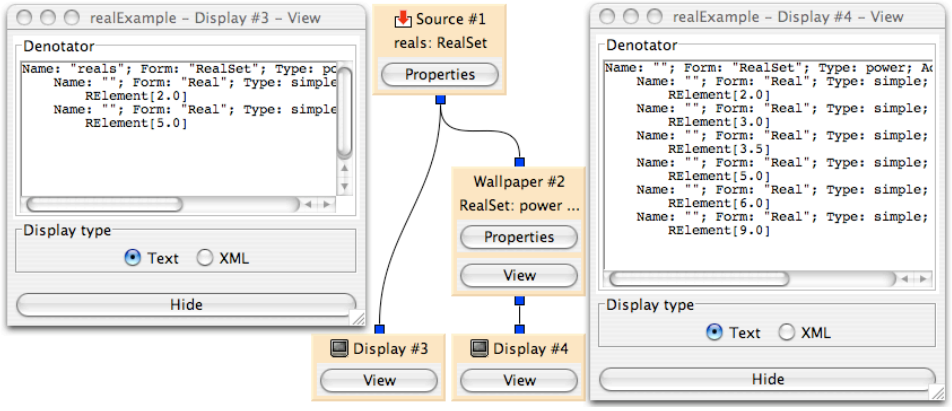


Figure 6.2: An example for a non-musical application of the Wallpaper rubette.

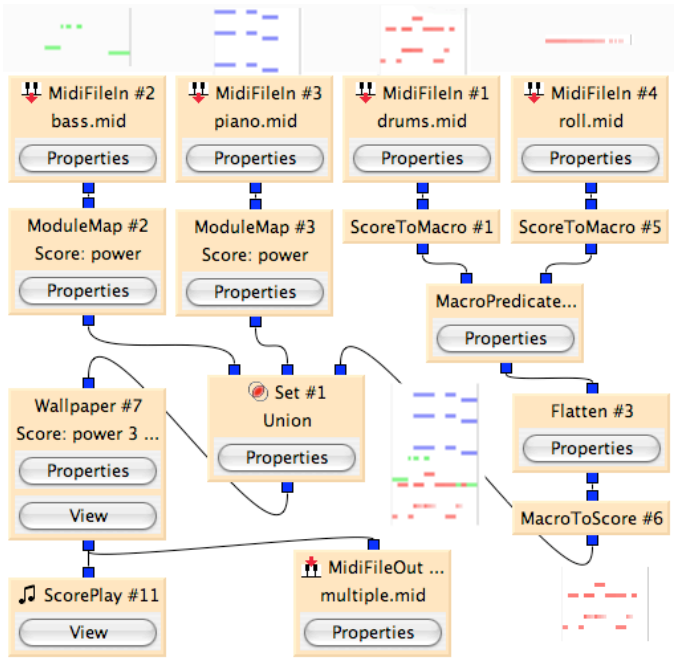
$[0, 1]$. Note that the wallpaper just contains 6 instead of 8 elements. This is because $f_1^0 \circ f_2^1(2) = 2$ and $f_1^1 \circ f_2^0(2) = f_1^1 \circ f_2^1(2) = 3$ and the wallpaper therefore should contain two copies of the *Real* denotator with value 2 and two copies with value 3. However, these would be equal and therefore, the **Power** denotator *reals* just accepts a single copy of each of them.

6.1.3 Polyphonics

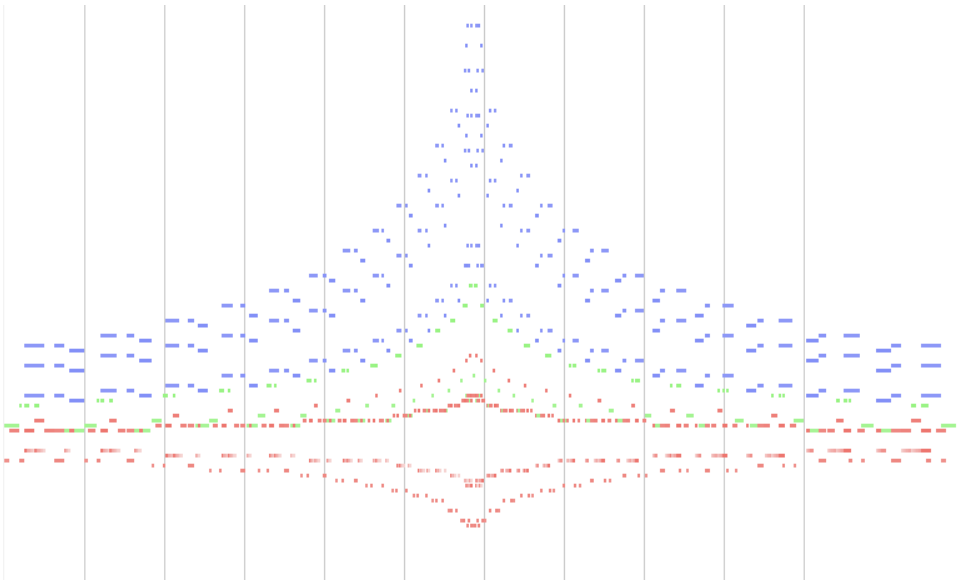
The next example, we presented at the *Digital Art Weeks Conference 2006* in Zürich. It demonstrates RUBATO COMPOSER's polyphonic possibilities by building a wallpaper from several voices. These are imported from midi files and finally exported again, e.g. to be played played by synthesizers in a sequencing software.

Figure 6.3 (a) shows the network for the example. The top four rubettes are the MidiFileIn rubettes, by means of which the four original MIDI files are imported and converted to the denotator format. Above each of them, a piano roll representation of the their output denotator is shown. The two leftmost denotators are then given separate voices (1 and 2) using ModuleMap rubettes, whereas the two other denotators keep the voice 0.

The latter two are converted to *MacroScores* by ScoreToMacro rubettes and combined using a MacroPredicateSelect rubette, which is set to embellish all notes with pitch 38 in the left input denotator ('drums') by the right input denotator ('roll'). In this procedure, the notes of the 'roll' denotator are automatically compressed in time to fit the duration of the said 'drums' notes and added to them as satellite notes. This causes all snare drum notes (pitch 38, here) to be played as a roll. Because of the multi-level



(a)



(b)

Figure 6.3: A polyphonic example: its network (a) and the ScorePlay rubette view (b).

structure of the resulting *MacroScore* denotator, we have to flatten using the Flatten rubette it before converting it back to a *Score* denotator. The embellished 'drums' denotator is shown graphically below the MacroToScore rubette.

The three denotators with different voices generated so far are then united using the Set rubette (the corresponding score is shown on the right hand side of the Set rubette) and passed on to the Wallpaper rubette. The wallpaper used in this example is slightly more complex than the previous ones. We have a grid of two affine morphisms $f_1 : \mathbb{R} \to \mathbb{R}$ and $f_2 : \mathbb{R}^3 \to \mathbb{R}^3$ with $f_1(c) = -c + 46.9$ and

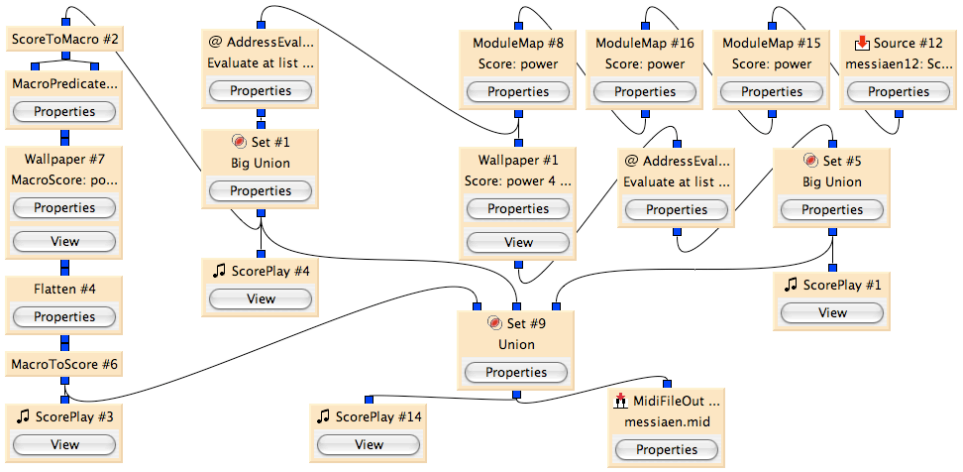
$$f_2(c) = \begin{pmatrix} 0.85 & 0 & 0 \\ 0 & 1.12 & 0 \\ 0 & 0 & 0.85 \end{pmatrix} c + \begin{pmatrix} 4 \\ -4.8 \\ 0 \end{pmatrix}$$

as well as the ranges $r_1 = [0, 1]$ and $r_2 = [0, 12]$. The coefficients in f_2's matrix and shift are the result of experimentation with the purpose of obtaining an auditory satisfying result. f_2 compresses onset and duration by a factor of 0.85 and stretches pitch by a factor of 1.12 ($G. = (Onset, Pitch, Duration), H. = (Onset, Pitch, Duration)$). Additionally, it translates the motif by 4 in time, which is the original length of the motif, i.e. the translated motif starts directly after its predecessor. By doing this twelve times in a row (for the whole range r_2), we obtain the left half of the wallpaper shown in Figure 6.3 (b). Its right half is created by the second morphism, which reflects the onsets ($G. = (Onset), H. = (Onset)$) of the left half at point 0 and translates it by 46.9, which is twice its length. We hereby obtain a final wallpaper, where the first half is followed by its retrograde. This wallpaper is then handed on to the ScorePlay rubette and exported by the MidiFileOut rubette.

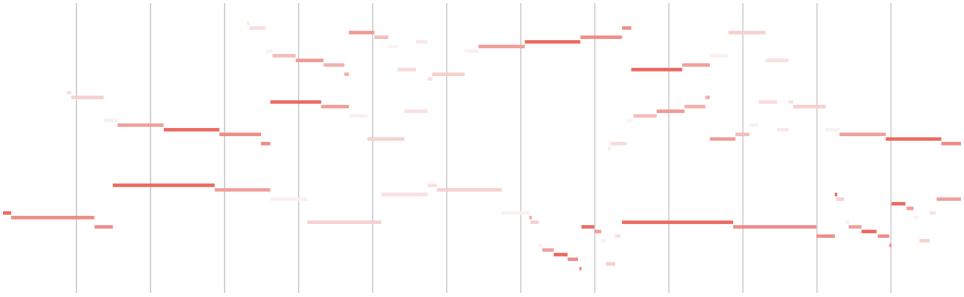
6.1.4 Functorial Dodecaphonics

Rubato Composer has a functorial background and therefore our last example for the Wallpaper rubette shows how the theory of functors can be used to easily produce a small composition from a dodecaphonic series, which is taken from Olivier Messiaen's *Mode de valeurs et d'intensités*. It is the same series Pierre Boulez processed in his composition *Structures Ia*. We presented this example at the *International Computer Music Conference 2006*.

We use two Wallpaper rubettes here, one in combination with *MacroScores* and one for functorial denotators. The network for the composition is shown in Figure 6.4 (a). The Source rubette in the top right corner holds a $\mathbb{Z}^{11}$-addressed denotator $m : \mathbb{Z}^{11}@Score$ that contains just one note $\mathbb{Z}^{11}@Note$,



(a)



(b)

Figure 6.4: The network (a) of a functorial example and an excerpt of its piano roll representation (b).

which represents the whole twelve-tone series. m is then processed by three ModuleMap denotators to adapt onset, pitch and duration for the later needs. In the network's left part, m is evaluated at the basis vectors of the address space $\mathbb{Z}^{11}$ by an AddressEval rubette (see [Mazz06] for more information about this). We obtain a *Score* of twelve notes which is passed on to a network of *MacroScore* rubettes, where a bass line is built by replacing several notes of it by the whole score itself (a fractal construction). Within this subnetwork, the bass line is translated several times (range [1,3]) by a Wallpaper rubette.

However, what's interesting in the context of this thesis, is the right part of the network. There, the original $\mathbb{Z}^{11}$-addressed m is passed on to the Wallpaper rubette, where we generate a wallpaper, which in this case will be a score of 12 $\mathbb{Z}^{11}$-addressed notes. We use a grid of four morphisms $f_1, \ldots f_4$, the precise definitions of which will not be given here. Nevertheless, we

quickly explain their function (in reverse order, because this is their order of application):

- $f_4 : \mathbb{R}^3 \to \mathbb{R}^3$ halves onset and duration of the series and translates the series up in pitch and onset, so that it sounds higher and ends exactly at the same time as the original series. The corresponding range is $r_4 = [1, 1]$, so the original series is not included in the wallpaper.
- $f_3 : \mathbb{R}^4 \to \mathbb{R}^4$ again halves onset and duration of the single series obtained after the transformation using f_4 and again translates it in the same way. Furthermore, its loudness is decreased. This time, we use a range $r_3 = [0, 1]$, so the result is a wallpaper with two series.
- $f_2 : \mathbb{R}^2 \to \mathbb{R}^2$ creates an inversion of the wallpaper created so far, so that the inversion of the compressed series takes the place in pitch of the original, and the inversion of the original takes the place of the compressed. The inversions are also translated to the end of the wallpaper generated so far and both of them are kept ($r_2 = [0, 1]$).
- $f_1 : \mathbb{R}^2 \to \mathbb{R}^2$ finally creates two copies of the four-series wallpaper also displaced in time and slightly down in pitch ($r_1 = [0, 1]$), so that the returned wallpaper includes twelve transformed copies of the original.

The resulting score is then evaluated at the basis vectors of the address space $\mathbb{Z}^{11}$ and finally merged with the original series and the bass line in a Set rubette. An excerpt of the resulting *Score* is shown in Figure 6.4 (b). The middle and top voices show the beginning of what the second wallpaper created, whereas the lower voice is part of the result from the *MacroScore* subnetwork.

6.2 Alterations

Next, we present two examples for alteration. They are the two most common in musical applications, namely tonal alteration and quantizing.

6.2.1 Tonal Alteration

Tonal Alteration, as explained in the context of *presto* in Section 3.3.2, can be easily reproduced in RUBATO COMPOSER. Figure 6.5 shows the minmal network settings. The Scale rubette is ideal to provide a scale towards which the notes of the composition are altered. In the Alteration rubette preference window, the necessary specifications for an alteration gradually increasing in time can be seen (low degree 0, high degree 1).

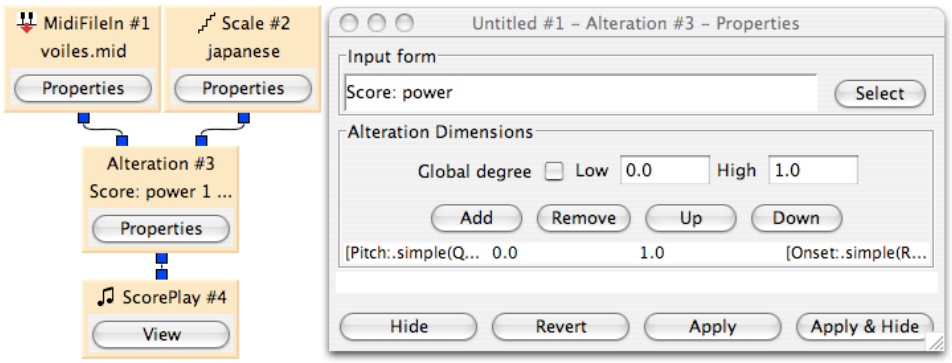


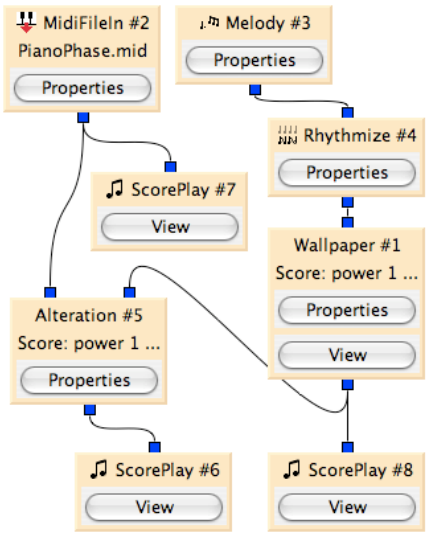
Figure 6.5: The settings for performing tonal alteration on a given composition.

6.2.2 Quantizing

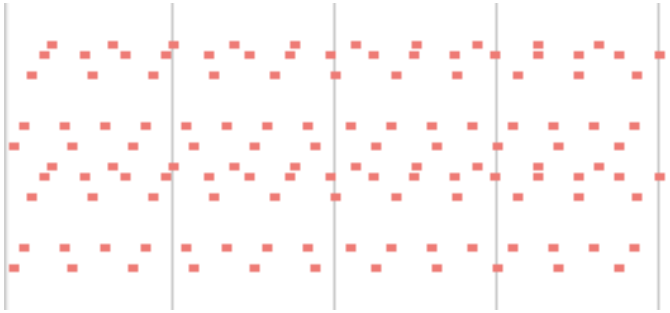
To perform quantizing on a given composition, we need to define a rhythmical grid. This can be done using the Rhythmize rubette. We first configure a Melody rubette to create a simple melody, each of its note having the same pitch (this does not matter, since do not alter pitch). Then, we pass the melody on to a Rhythmize rubette, which outputs the rhythmic motif for our grid. We set the Rhythmize rubette's properties so that this motif's length is exactly one bar (here, 2.0 time units).

The grid is then produced using a Wallpaper rubette, which just loops the motif, i.e. uses one morphism displacing the motif's onsets by 2.0. It is important to make sure here, that the obtained wallpaper exceeds the length in time of the composition to be quantized. The quantizing itself is finally managed by an Alteration rubette, where we define one alteration dimension for onset.

Figure 6.6 shows a composed network for quantizing the starting pattern of Steve Reich's *Piano Phase* (a typical wallpaper, by the way) with a gradually increasing degree (low 0, high 1). Part (b) of the same figure shows the resulting score. Even though visually the differences are minimal, they are clearly audible. The originally even rhythm is disturbed and virtually absorbed by the rhythm produced by the Rhythmize rubette.



(a)



(b)

Figure 6.6: An example for quantizing. The network (a) and the resulting score (b).

Chapter 7

Conclusion

In this chapter we evaluate the software written for this thesis by contrasting its features with those of its predecessor. Subsequently, we present the difficulties we encountered during implementation and the limitations of our new product. Finally, we outline future and other possible projects either using the new Rubato Composer modules or developing them further.

7.1 Evaluation

The new implementation for grid structure generation and application is more flexible in many ways, as shown in Table 7.1. We summarize and discuss the improvements.

Due to its extendable structure and the rubette concept, Rubato Composer promotes a modular way of thinking, where many functional parts act together in every conceivable way. To maximize the flexibility of use, the functionality of the OrnaMagic module has been separated in two main and three additional modules. The combination of these modules leads to new ways of using grid structures for composition.

Furthermore, the two main modules, the Wallpaper and the Alteration rubette can be used for processing any arbitrary structures, not just musical ones, since they accept any denotators of type **Power** as inputs.

All note coordinate values in *presto* are positive integers. Pitch, loudness and duration are moreover restricted to a maximal value of 72. In Rubato Composer, the processed structures can have elements of many different modules as values. Our present implementation of wallpapers and alteration includes the modules $\mathbb{Z}, \mathbb{Q}, \mathbb{R}$ and $\mathbb{Z}_n$.

	presto OrnaMagic	Rubato Composer
design	serial	modular
input	set of notes	any **Power** denotator
values	discrete integer values	$\mathbb{Z}, \mathbb{Q}, \mathbb{R}$ or $\mathbb{Z}_n$ values
space	onset×pitch/loudness/duration	any arbitrary space
grid vectors	2-dim translations	n-dim morphisms

Table 7.1: Differences between the two implementations.

The space for grid structure generation and application is two-dimensional in *presto* and consisting of onset and a second coordinate. The new implementation allows the selection of any conceivable n-dimensional space.

This difference also applies to the grid vectors. In *presto*, these were in the two-dimensional and additionally restricted to simple translations. Our new implementation features affine and other morphisms of any dimension.

7.2 Open Issues

Despite the benefits of our new implementation, there are some open issues that have not been implemented yet:

- In *presto*, users can make the decision which neighbour should preferably be chosen as an alteration goal, in case there are *multiple nearest neighbors*. In our early implementation, we found a solution for integrating this issue in Rubato Composer. However, since we use a *kd*-tree now for nearest neighbor search this is not possible anymore.
- The *diagonal rails* that could be selected to restrict alteration direction in *presto* could not be adapted to Rubato Composer. For their realization, the mathematical definition of alteration has to be extended.
- The present implementation is restricted to process denotators of **Simple** forms that are embedded in $\mathbb{R}$. Forms of other modules like string rings or polynomial rings are not yet supported

7.3 Future Work

There are many possibilities for future uses and further developments of the contributions of this thesis. Just a few of them are stated here:

- At the first international conference of the *Society for Mathematics and Computation in Music*, a demonstration of the possibilities of

wallpapers and alteration will be presented [Mazzar].

- In a composition project with Guerino Mazzola's research group, Pierre Boulez' *Structures Ia* is reconstructed and altered with RUBATO COMPOSER. In this context, the Wallpaper rubette has to be developed further to be able to generate wallpapers made of multiple motifs.
- Using special denotators defining a function, the alteration degrees could be defined more precisely than just by two boundary values. For the generation of such functions, a rubette could be created where users can draw functions by hand or select a preset function.
- The concepts of alteration can easily be extended for creating musical morphings, analogous to visual morphing in the domain of image processing. A so-called morph of two compositions d_1 and d_2 can be defined as a selection of elements of the two altered compositions $alt(d_1, d_2)$ and $alt(d_2, d_1)$. This leads to a more complete implementation of musical morphings than for example described in [Oppe95].
- A further extension could be to produce morphings between an arbitrary number of compositions $d_1, \ldots d_n$ using multidimensional degree functions.

Appendix A

Manuals

This appendix includes the extensions to the RUBATO COMPOSER User's Manual [Milm06b], which is available on [RubatoC]. The rubettes created for this thesis are described using the schema proposed in Section 4 of the manual.

A.1 Score Rubettes

Melody

Group: Score

Summary: Generates a random melody.

Outputs: #1: A denotator of form *Score*.

Description: The generated melody consists of a specified number of subsequent notes of duration 1.0, loudness 120 and voice 0. The first note is at the initial pitch. Every subsequent note's pitch is calculated randomly, where the distance to its previous note never exceeds the indicated interval boundaries, which are interpreted in steps, e.g. step size 1 stands for halftones, $\frac{1}{2}$ for quartertones. Furthermore, the stated lowest and highest pitch are never exceeded.

Properties: The number of notes must be specified, as well as the highest, lowest and initial pitches. Furthermore, the step size and the minimum and maximum intervals up and down in steps have to be indicated (see Figure A.1).

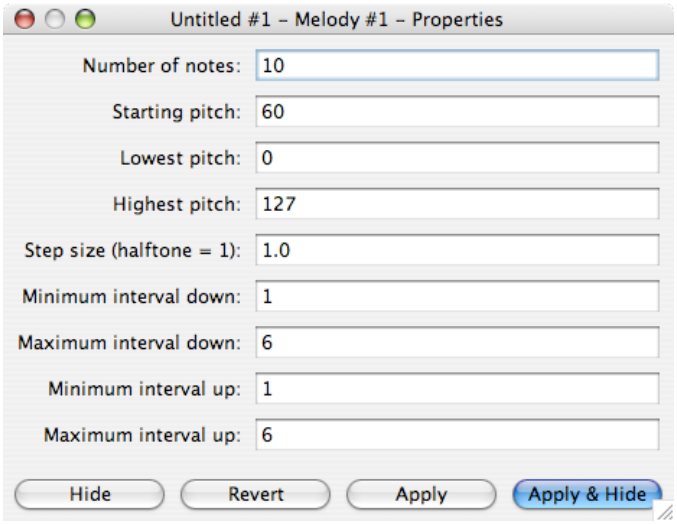


Figure A.1: The properties window of the Melody rubette.

Rhythmize

Group: Score

Summary: Provides the input denotator with a new rhythm.

Inputs: #1: A denotator of form *Score*.

Outputs: #1: A denotator of form *Score*.

Description: A new score is created taking the notes from the input denotator in their order of appearance and giving them new onsets and durations. The resulting score is monophonic (one note at a time) and its durations are randomly taken from the exponentially calculated values $b^x, b^{x-1}, \ldots b^{x-n+1}$, where b is the specified base, x is the specified exponent and n is the specified number of values.
If the specified metric period is greater than 0, it is made sure that there is a note at every meter point (multiple of the metric period). These notes obtain the accented loudness, whereas all other notes obtain the unaccented one.

Properties: The number of different notes values and the value base and maximum exponent have to be specified. Additionally, a meter period can be specified. If it is greater than 0, the loudnesses of accented and unaccented notes must be indicated, else it is

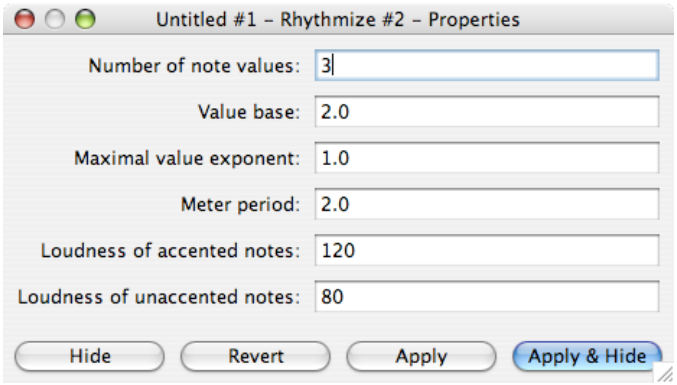


Figure A.2: The Rhythmize rubette's properties window.

ignored. The properties dialog is shown in Figure A.2.

Scale

Group: Score

Summary: Generates a musical scale.

Outputs: #1: A denotator of form *Score*.

Description: Either a predefined scale or a custom scale is generated, the pitches of its notes ranging from 0 to 127 (MIDI pitches). Onset is set to 0, loudness to 120, duration to 1 and voice to 0 for every note in the scale. Starting at the specified root note pitch, the defined sequence of intervals is stepped through repeatedly, until the above mentioned boundaries are reached. For each pitch reached this way, a note is added to the output denotator.

Properties: The number of notes has to be specified using the slider (see Figure A.3). Then, the pitch of the root note has to be indicated and using the combo box, either one of the predefined scales or 'custom' has to be selected. If 'custom' is selected, the desired intervals between subsequent notes have to be specified in the interval text fields.

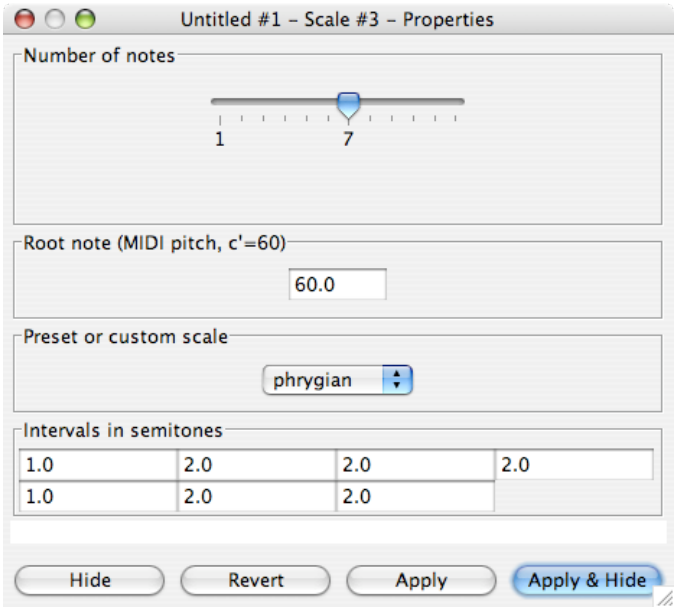


Figure A.3: The properties window of the Scale rubette.

A.2 Core Rubettes

Wallpaper

Summary: Generates a wallpaper from an input motif.

Inputs: #1: A denotator of type **Power**.

Outputs: #1: A denotator of the same form as input #1.

Description: All elements of the input denotator are mapped repeatedly using the specified morphisms and finally, part of the mappings are united in a new denotator of the input form and returned. The first morphism to map is the one of the wallpaper dimension lowest in the table, then the next higher and so on.
The indicated range defines for a dimension, which mappings are added to the result **Power** denotator. $[2, 5]$ for example, means that from the denotators after the second mapping to the ones after the fifth mapping, all denotators are added to the result. The last column of the table defines the forms of the **Simple** denotators mapped in each element.

Properties: First, the **Power** form of the input denotator has to be specified using the standard form entry. Then, using the 'Add' but-

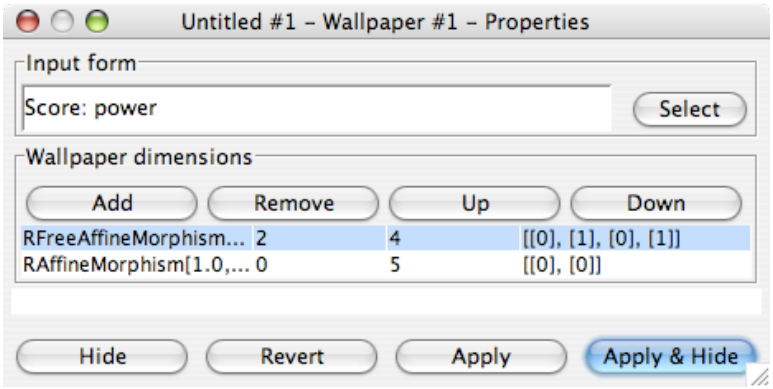


Figure A.4: The Wallpaper rubette properties window.

ton, any number of wallpaper grid dimensions can be added (see Figure A.4). These are shown in the table below the button row. When the 'Remove', 'Up' or 'down' button is pressed, the dimensions selected in the table are removed, moved one position up or one down in the table.
For every dimension, a module morphism has to be specified in the dialog that opens, when a field in the leftmost column of the table is double-clicked. In the two middle columns, the dimension's range has to be defined. Finally, by double-clicking a field of the rightmost column, the **Simple** forms to be transformed by the morphism indicated above, have to be specified. The appearing dialog is shown in Figure A.5 (a). For every module of the morphism's domain and codomain, a **Simple** form needs to be selected using the combo boxes.

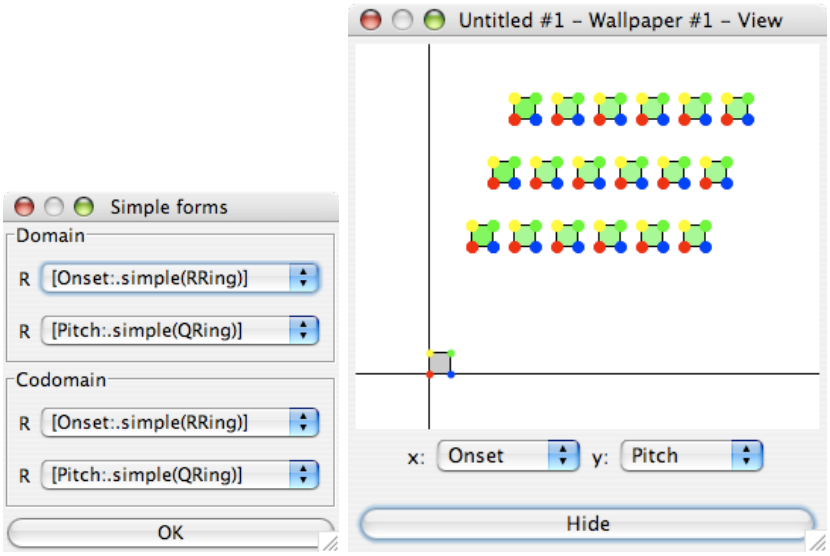
View: The wallpaper rubette provides a view (see Figure A.5 (b)), which is similar to the graphical representation of geometrical morphisms in the 'Edit module morphism dialog'. It simulates the creation of a wallpaper from a square at $(0, 0)$ visually. The coordinates shown can be selected using the two combo boxes.

Alteration

Summary: Alters a first composition towards a second.

Inputs: #1: The denotator of type **Power** to be altered.

#2: The denotator towards which the alteration takes place. Of the same form as input #1.



(a) (b)

Figure A.5: The **Simple** forms dialog (a) and the Wallpaper rubette view (b).

Outputs: #1: The altered denotator of the same form as input #1.

Description: The Alteration rubette moves each element of the first input denotator towards its nearest neighbor of the second composition. For each dimension and element, the **Simple** denotator affected by the alteration is of the form specified in the leftmost column of the dimensions table and it happens to be the first one of that form found in the element's tree. Furthermore, for each dimension and element, a local alteration degree is calculated according to its position relative to all other elements, which is calculated from the two indicated alteration degrees and the value of the first denotator found of the form specified in the fourth column of the table.

Properties: First, using the standard form entry, the **Power** form of the input denotator has to be specified (see Figure A.6). Using the buttons 'Add', 'Remove', 'Up' and 'Down', alteration dimensions can be added to removed from or moved in the dimensions table, in the same way as in the Wallpaper rubette's table.
In the first column of the table, the altered form has to be specified for each dimension. The next two columns hold the alteration degrees, the second is the degree for the lowest value

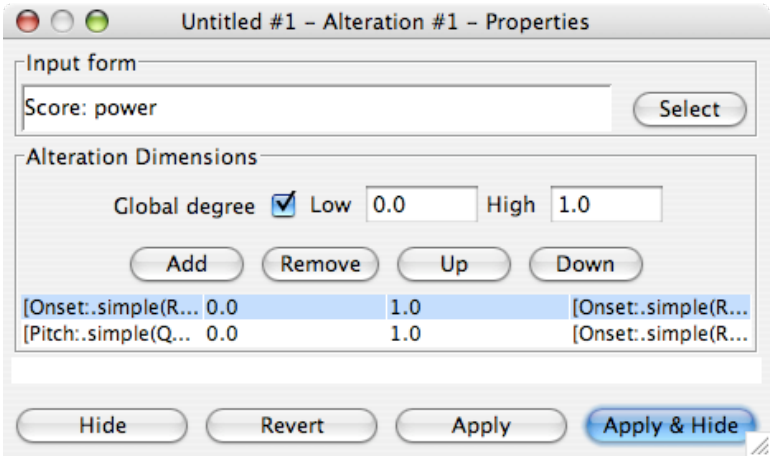


Figure A.6: The Alteration rubette properties window.

and the third the one for the highest value. In the last column, the form has to be specified, relative to the values of which the local alteration degrees are calculated.
If the alteration degrees are meant to be the same for all dimensions, the 'Global degree' checkbox can be selected. All degrees will then be set to the values in the two text fields at the right hand side of the checkbox.

Bibliography

[Andr01] Moreno Andreatta, Thomas Noll, Carlos Agon, and Gerard Assayag. "The Geometrical Groove: rhythmic canons between Theory, Implementation and Musical Experiment". In: *Actes des Journes dInformatique Musicale*, pp. 93–98, Bourges, 2001.

[Andr02] Moreno Andreatta, Carlos Agon, and Emmanuel Amiot. "Tiling problems in music composition: Theory and Implementation". In: *Proceedings of the International Computer Music Conference*, pp. 156–163, ICMA, Göteborg, 2002.

[Babb92] Milton Babbitt. *The Function of Set Structure in the Twelve-Tone System*. PhD thesis, Princeton University, 1946, accepted 1992.

[Bent75] Jon Louis Bentley. "Multidimensional binary search trees used for associative searching". *Communications of the ACM*, Vol. 18, No. 9, pp. 509–517, Sep. 1975.

[Bres05] Jean Bresson, Carlos Agon, and Gerard Assayag. "OpenMusic 5: A Cross-Platform Release of the Computer-Assisted Composition Environment". In: *10th Brazilian Symposium on Computer Music*, SBCM, Belo Horizonte, 2005.

[Burr04] Dave Burraston, Ernest Edmonds, Dan Livingstone, and Eduardo Reck Miranda. "Cellular Automata in MIDI based Computer Music". In: *Proceedings of the International Computer Music Conference*, Miami, 2004.

[Coop60] Grosvenor W. Cooper and Leonard B. Meyer. *The Rhythmic Structure of Music*. University of Chicago Press, Chicago, 1960.

[Cubase] "Cubase 4". http://www.steinberg.de/.

[Essl91] Karlheinz Essl. "Computer Aided Composition". *herbst-ton*, 1991.

[Essl96] Karlheinz Essl. "Strukturgeneratoren". *Beitrge zur Elektronischen Musik*, 1996.

[Eule39] Leonard Euler. *Tentamen novae theoriae musicae*. 1739.

[Finale] "Finale 2007". http://www.finalemusic.com/finale/.

[Fux65] Johann Joseph Fux. *The Study of Counterpoint (Gradus ad Parnassum). Translated by Alfred Mann*. W. W. Norton & Co., New York, 1965.

[GnuGPL] "GNU General Public License". http://www.gnu.org/copyleft/gpl.html.

[Gold84] Robert Goldblatt. *Topoi, the Categorial Analysis of Logic*. North-Holland, New York, 1984.

[Huda96] Paul Hudak, Tom Makucevich, Syam Gadde, and Bo Whong. "Haskore Music Notation - An Algebra of Music". *Journal of Functional Programming*, Vol. 6, No. 3, pp. 465–483, 1996.

[JMusic] "Piano Phase - After Steve Reich". http://jmusic.ci.qut.edu.au/jmtutorial/PianoPhase.html.

[KDTree] "edu.wlu.cs.levy.CG.KDTree". http://www.cs.wlu.edu/~levy/kd/.

[Lewi87] David Lewin. *Generalized Musical Intervals and Transformations*. Yale University Press, New Haven, 1987.

[Lewi93] David Lewin. *Musical Form and Transformation: Four Analytic Essays*. Yale University Press, New Haven, 1993.

[Logic] "Logic Pro". http://www.apple.com/logicpro/.

[Mazz02] Guerino Mazzola. *The Topos of Music*. Birkhäuser, Basel, 2002.

[Mazz06] Guerino Mazzola, Gérard Milmeister, Karim Morsy, and Florian Thalmann. "Functors for Music: The Rubato Composer System". In: *Digital Art Weeks Proceedings*, ETH, Zürich, 2006.

[Mazz89] Guerino Mazzola. *The Presto Manual*. SToA Music, Zürich, 1989.

[Mazz90] Guerino Mazzola. *Geometrie der Töne: Elemente der Mathematischen Musiktheorie*. Birkhäuser, Basel, 1990.

[Mazz94] Guerino Mazzola and Oliver Zahorka. "The RUBATO Performance Workstation on NEXTSTEP". In: *ICMC 94 Proceedings*, ICMA, San Francisco, 1994.

[Mazzar] Guerino Mazzola and Florian Thalmann. "Grid Diagrams for Ornaments and Morphing". In: *First International Conference of the Society for Mathematics and Computation in Music*, MCM, Berlin, 2007, to appear.

[MIDI96] *The Complete MIDI 1.0 Detailed Specification.* The MIDI Manufacturers Association, Los Angeles, 1996.

[Milm06a] Gérard Milmeister. *The RUBATO Composer Music Software: Component-based Implementation of a Functorial Concept Architecture.* PhD thesis, University of Zürich, 2006.

[Milm06b] Gérard Milmeister. "RUBATO COMPOSER User's Manual". Tech. Rep., University of Zürich, 2006.

[Mors] Karim Morsy. "Denotatorbasierte Makroevents und Methoden der musikalischen Komposition". Ongoing research project, University of Zürich.

[Nien03] Han-Wen Nienhuys and Jan Nieuwenhuizen. "LilyPond, a system for automated music engraving". In: *Proceedings of the XIV Colloquium on Musical Informatics*, CIM, Firenze, 2003.

[Oppe95] Daniel V. Oppenheim. "Demonstrating MMorph: A System for Morphing Music in Real-time". In: *Proceedings of the International Computer Music Conference*, ICMA, Banff, 1995.

[Pier91] Benjamin C. Pierce. *Basic Category Theory for Computer Scientists.* MIT Press, Cambridge, 1991.

[Presto] "*presto*® for Atari ST". http://tamw.atari-users.net/dwnloads.htm.

[Puck07] Miller Puckette. *The Theory and Technique of Electronic Music.* World Scientific Publishing Co. Pte. Ltd., 2007.

[RubatoC] "RUBATO COMPOSER". http://www.rubato.org/rubatocomposer.html.

[RubatoN] "Rubato for NEXTSTEP". ftp.ifi.unizh.ch/pub/projects/rubato.

[RubatoX] "Rubato for OS X". http://flp.cs.tu-berlin.de/MaMuTh/Downloads/Rubato/.

[Sibelius] "Sibelius 4". http://www.sibelius.com/products/sibelius/index.html.

[Slon47] Nicolas Slonimsky. *Thesaurus of scales and melodic patterns.* Coleman-Ross Company, New York, 1947.

[Ston80] Kurt Stone. *Music Notation in the Twentieth Century: A Practical Guidebook*. W. W. Norton, New York, 1980.

[Taub91] Heinrich Taube. "Common Music: A Music Composition Language in Common Lisp and CLOS". *Computer Music Journal*, Vol. 15, No. 2, pp. 21–32, 1991.

[Thal06] Florian Thalmann and Markus Gaelli. "Jam Tomorrow: Collaborative Music Generation in Croquet Using OpenAL". In: *Proceedings of the Fourth International Conference on Creating, Connecting and Collaborating through Computing*, University of California, Berkeley, 2006.