Functors for Music: The Rubato Composer System

Guerino Mazzola and Gérard Milmeister
Department of Informatics, University of Zurich
guerino@mazzola.ch
milmei@ifi.unizh.ch

Abstract

In abstract mathematical music theory, the data format of denotators is associated with set-valued presheaves over module categories. In this paper, we present an implementation of this concept framework in the Rubato Composer system, a Java application comprising a GUI for manipulation and combination of rubettes. These are plug-ins that can be added and connected for communication of denotator data. Rubato Composer is a GPL software and is accessible to the computer music community for download and collaboration. In this paper, the functionality, architecture, concept framework, and the implemented mathematical operators are presented and illustrated with a “functorial composition”.

1 Introduction

In 1984, the principles of a music composition software were presented to Herbert von Karajan and led to the Atari-application¹ *presto*[®], used for the creation of a number of full-fledged compositions, e.g., the jazz CD *SYNTHESIS* (Mazzola 1991). In the sequel, our research has focused on a comprehensive environment RUBATO[®] for analysis, performance, and composition, including plug-ins for specific tasks. Such a plug-in is called RUBETTE[®], and we managed to realize rubettes for rhythmical, melodic, and harmonic analysis, as well as a rubette for expressive performance and one for prima-vista operations. Originally, these implementations ran on NEXTSTEP and were later ported to Mac OS X. See the Rubato homepage www.rubato.org for the different Rubato implementations and documentation. Rubato has been applied to musicological research, which is documented in ICMC papers (Mazzola and Zahorka 1994; Mazzola 1995; Zahorka 1995; Mazzola and Stange-Elbe 1995; Mazzola and Müller 2003; Buteau 2005; Volk 2005), papers on other conferences (Buteau 2004; Göller and Milmeister 2004; Müller 2004; Garbers and Noll 2004; Garbers 2004), and in the systematic context of a comprehensive theory (Mazzola 2002).

However, the compositional components of Rubato have not been implemented in the NEXTSTEP and Mac OS X versions. In the present implementation, called *Rubato Composer*, which is a Java application, the original *presto* features are reconsidered and implemented within the framework of functorial mathematical music theory as described in (Mazzola 2002). This gives the compositional tools an unprecedented conceptual power. Intuitively, complex musical objects, such as dodecaphonic series, chords consisting not of ordinary notes, but of collections of melodies, or ornaments of rhythmic patterns, and, more generally, macro-object constructions in the spirit of Schenker, can be handled as if they were ordinary notes. In his introduction to the OpenMusic book (Agon, Assayag, and Besson 2006), Miller Puckette notes: “In the future, computer music in general will have to deal with high-dimensional objects such as would specify a timbre. The dimensionality of a set such as the ‘set of all possible sounds’ is probably not even well defined. New techniques will be needed to describe and manipulate such quantities.” We propose the present denotator concept as a three-way approach to the solution of the problems described by Puckette: (1) The introduction of addressed denotators, enabling the control of arbitrarily parametrized objects; (2) the construction of arbitrary finite limits, e.g., direct sums of modules; (3) the definition of circular concepts of arbitrary depth, used in FM synthesis, for example. The analytical and performance-theoretical tools of the preceding Rubato implementations will be reimplemented in the near future. For the time being, we focus on composition.

After an overview of the functionality and architecture, we describe the universal space concept of forms and their ‘points’, the denotators. We then describe their implementation in Rubato Composer and give examples of denotator constructions. Next, we describe the functorial operations and illustrate their musical signification. The paper concludes with the explicit construction of a simple composition on the basis of the functorial concept framework.

¹shareware, accessible via tamw.atari-users.net/presto.htm

2 Overview

The Rubato Composer system follows the classical Rubato architecture (Mazzola and Zahorka 1994): A frame application hosts a number of instantiated plug-ins, the rubettes, which perform specialized tasks. By means of a graphical user interface, rubettes are connected through input and output slots in networks. To this end, they are dragged from a repository onto a canvas representing the network, see figure 1.

Figure 1: The Rubato Composer network window.

The distinctive feature of this network system is that the inter-rubette communication is performed via the denotator data format exclusively. This normalization is not restrictive at all since denotators realize a universal typology, which is endorsed by the most-advanced contemporary mathematics, viz. topos theory. Therefore, communication among rubettes uses a comprehensive language which fits in the mathematical universe. In this paper, we want to demonstrate that even the most abstract category theory of functors generated by the Yoneda embedding is canonically accessible on the basis of denotators. We also want to show that such tools are of practical use to composers and music theorists alike.

2.1 Functionality

A priori, the functionality of rubettes is only limited by the programmer's skill. The essential difference to the original rubette concept is the following. In the original context, we had four types of rubette tasks in mind: (1) analysis, (2) composition, (3) performance, and (4) logico-geometric operators. The fourth group was born from the insight or rather

suspicion that many tasks in special rubettes can be decomposed into standard procedures of a logical and geometric nature.

In the present setup, the functionality of rubettes is not straitjacketed into such a rigid categorization. The main objective is reuse via the universal denotator interface. This component-oriented principle dissolves the original logico-geometric operator group since any rubette may define components of this type. But it also externalizes a number of functionalities which were hardcoded in the original Metro-, Melo-, or Performance-rubettes. The difference to general component architectures is that denotators present a clear formal conceptualization and therefore offers an interoperability beyond ad hoc interfaces.

2.2 Architecture

The Rubato Composer GUI and the Rubato framework are completely implemented in pure Java (JDK version 1.5), using various standard components of the JDK such as Swing for the graphical user interface, XML for the file data format and the Sound and MIDI APIs for score playing. Figure 2 visualizes the most important parts of the architecture.

Figure 2: The architecture of Rubato Composer.

The bottom to top direction of the diagram reflects the passage from lower to higher levels in the architecture itself. Thus, at the bottom is the general implementation system, the JDK. The first level of Rubato Composer proper consists of the triple hierarchy of mathematical modules, elements of module spaces, and morphisms between them. The italic

typeface in this picture indicates the existence of a class or an interface of the same name. The background of this part is discussed in section 3.2.

On this concrete foundation sits the implementation of the core of Rubato Composer, namely denotators and forms. Section 3.1 deals with the details of this part. These classes are supplemented by tools for creating and manipulating forms and denotators, such as, for example, Boolean operations on denotators of type Power.

The last level, most visible to application-oriented users, contains the implementation of Rubato Composer GUI including the rubette Java interface and a number of core rubettes.

Essentially, the rubette interface exhibits several methods that define the operation of a rubette, namely the flow of data (denotators) from input to output. Additional, but optional, features include the definition of documentation strings and the export of a properties dialog to customize settings, as well as a view dialog for audio-visual representation of the process data.

3 Universal Spaces: Forms and Denotators

Forms and denotators are the data format on which Rubato communication is based. This format is the conceptual basis of mathematical music theory. We refer to (Mazzola 2002; Mazzola 2004), where this setup has been described in extenso. We first discuss the general concept architecture (section 3.1) and then the mathematical basis (section 3.2).

3.1 Forms, Denotators and their Implementation

Forms

The space format is called a form. The form F is specified by the expression

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

The form name Name_F is a denotator (to be defined below), the form identifier Id_F is a monomorphism $Id_F : \text{Space}_F \rightarrow \text{Frame}_F$ of presheaves², Type_F is one of four types: Simple, Limit, Colimit, or Power. And Dia_F is a diagram of forms³. Depending on the type, the data is specified as follows.

²Presheaves are contravariant functors $P : \mathbf{Mod} \rightarrow \mathbf{Sets}$ from the category $\mathbf{Mod}$ of modules and diaffine morphisms into the category $\mathbf{Sets}$ of sets

³If F, G are two forms, a morphism $f : F \rightarrow G$ is by definition a natural transformation of their underlying spaces.

For those readers that are not completely familiar with the mathematics and its notation used here, statements are included that point to the analogous constructs of the data type model in the programming language Pascal and XML.

Type Simple: The diagram degenerates to a module M , standing for the representable presheaf $\text{Hom}(-, M) = @M$. This space type is the mathematical basis. In Pascal, this corresponds to the primitive datatypes such character, integer, or real, but also arrays of such primitive data types. But notice, that type Simple is much more comprehensive, for example, there are also polynomials. We shall describe the category of modules and its implementation in section 3.2. In XML, only character data (PCDATA) are primitive.

Type Limit: Frame_F is the limit of the form spaces of the given diagram. In particular, if the diagram is discrete (i.e. no arrows), then we have the Cartesian product of a list of factors, representing a conjunction of attributes. The Pascal analogue is the record data type, which is equivalent to the special case of the Cartesian product. In the XML content model, it is known as the sequence type. If arrows are present in the diagram, the limit construction formalizes constraints upon the coordinates of the Cartesian product. This is in fact akin to constraint programming.

Type Colimit: Frame_F is the colimit of the form spaces of the given diagram. In particular, if the diagram is discrete (i.e. no arrows), then we have the disjoint union of a list of cofactors, representing a disjunction of attributes. In Pascal, there exists a variant record data type, which discriminates on an additional record field and corresponds to the special case of the colimit as a disjoint union. The DTD specification of XML calls this the choice type. The presence of arrows creates glueing condition on the cofactors.

Type Power: The diagram has a single vertex form G and no arrow. The frame space is $\text{Frame}_F = \Omega^{\text{Space}_G}$, where Ω is the subobject classifier⁴. This corresponds to a collection of instances (i.e., denotators, see below) of the basic form G . Sets are not generally available in Pascal, except for the special case of a bitset, i.e., a data type of subsets of the set of integer values from 0 to 31. In XML, this type is absent.

Denotators

Based upon these form types, a denotator D is an instance of such a space given by the following data:

$$\text{Name}_D : \text{Add}_D @ \text{Form}_D(\text{Coord}_D).$$

Here, the denotator name Name_D is again a denotator, the address Add_D is a module A , the Form_D is a form F , and the

⁴At a given address A , this presheaf has the set of sieves, i.e., the subpresheaves of the representable presheaf $@A : A @ \Omega = \text{Sub}(@A)$.

coordinate Coord_D is an element of the evaluation $A@Space_F$ of the form space $Space_F$ of the form F of D .

Observe that the definition of forms and denotators is circular in that names of forms are denotators, while denotators refer to forms. Most programming languages, including Java, accept circular definitions. However, for the implementation of these data types, circularity has two aspects: The instantiation of a form-object of circular type, i.e., such that in the form's diagram, the form reappears, creates no problem, whereas operationally, the instantiation of denotators of circular type may cause endless loops.

Implementation

The present implementation is subject to some restrictions with respect to the mathematical theory: To begin with, the modules are not available in full generality, see the following section 3.2. Moreover, the instantiation of general Limit and Colimit typed forms is possible, but the exhibition of denotators of such forms becomes difficult if constraints or glueing conditions by diagram arrows are imposed. Currently, routines for the treatment of such constraints are not yet implemented. The novelty of the present implementation is the fairly complete realization of addressed denotators, including the techniques of address changes and space morphisms.

The implementation of presheaves P works for specified arguments, i.e., the definition of sets $A@P$ as well as set maps $f@P : B@P \rightarrow A@P$ for given maps $f : A \rightarrow B$ of address modules. Also, the morphisms between presheaves are given address-wise, i.e., we must specify set maps $A@P \rightarrow A@Q$ for given presheaves P, Q in order to define a morphism $P \rightarrow Q$.

Although the definition of presheaf morphisms is not feasible in complete generality, a number of cases can be realized. To begin with, the Yoneda lemma guarantees that morphisms $f : @M \rightarrow @N$ are bijectively related to morphisms of the underlying modules, i.e., $f = @g$ for precisely one diaffine morphism $g : M \rightarrow N$.

Moreover, any morphism $f : \text{colim } \mathcal{D} \rightarrow F$ is universally defined by the associated inductive system of morphisms $f_i : \mathcal{D}_i \rightarrow F$, whereas a morphism $f : F \rightarrow \text{lim } \mathcal{D}$ is universally determined by its associated projective system $f_i : F \rightarrow \mathcal{D}_i$. Conversely, one may define morphisms $f : F \rightarrow \text{colim } \mathcal{D}$ by a factorization through one of the spaces $\mathcal{D}_i$, and morphisms $f : \text{lim } \mathcal{D} \rightarrow F$ may be defined through the factorization via one of the projection spaces $\mathcal{D}_i$. Although this is not the general solution, we get universally defined morphisms. For example, all affine morphisms $f : \bigoplus M_i \rightarrow \bigoplus N_j$ between finite direct sums of modules can be constructed by the universal properties of limits and colimits. Moreover, in the present implementation, some general, not necessarily affine, set maps can be constructed, e.g., polynomial maps, thereby

invoking the larger category of modules and set maps. In contrast to the affine case, this was done without systematically accounting for category theory.

With respect to the Power type forms, a very special, but usually sufficient case of denotators can be constructed: The general A -addressed denotator D with the form

$$\text{Name}_F : Id_F.\text{Power}(G)$$

has the coordinate

$$c \in A@ \Omega^{\text{Space}_G} = \text{Sub}(@A \times \text{Space}_G).$$

It can be shown (Mazzola 2002, ch. 7.3) that every set S of A -addressed denotators in G gives rise to a so-called objective subpresheaf $\hat{S}$ of $@A \times \text{Space}_G$. Coordinates of shape $c = \hat{S}$ can be constructed in the present implementation.

Construction of Forms and Denotators in Rubato Composer

In the discussion of the architecture (section 2.2) a collection of tools on the programming level for comfortable construction of forms and denotators is mentioned. The Rubato Composer GUI exposes a user interface to these tools. As an example consider the construction of a form of type Limit with several factors, say already defined forms with names E (onset, type Simple with module $\mathbb{R}$), H (pitch, type Simple with module $\mathbb{Q}$), L (loudness, type Simple with module $\mathbb{Z}$) and ∂E (duration, type Simple with module $\mathbb{R}$). Then the dialog for the construction of a form called F looks like figure 3.

Figure 3: The form construction dialog.

The structure of the dialog is mostly self-explaining. One first selects the type of the form to create. The layout changes according to this choice. In the case of type Limit the factor

forms are added from a list of already defined forms (button **Add**) or newly created ones (**Add new**). The order can be changed using the **Up** and **Down**. Optionally a label may be assigned to a factor for easier referencing, here these are, in order, *onset*, *pitch*, *loudness*, and *duration*.

Figure 3 also shows how one could enrich the diagram of the Limit form: A domain (**From** list) and a codomain (**To** list) are selected and a morphism between the two is created using **Set morphism**, here between the first factor E and the third factor L .

Now, with the form F defined, the next step is to construct a denotator of this form. The intention is to build a $\mathbb{Z}^{11}$ -addressed denotator, therefore, the module of the form H being $\mathbb{Q}$, a morphism $m : \mathbb{Z}^{11} \rightarrow \mathbb{Q}$ needs to be defined. To this end the module morphism builder tool is used (figure 4).

Figure 4: The module morphism construction dialog.

Figure 5: The simple denotator construction dialog.

The domain and codomain of the morphism are selected, then the type of morphism, here the choice is “affine morphism”. The morphism with name m is of the form $m(x) = Ax + b$, where A is an integer (1×11)-matrix and b an integer. Similarly a morphism $\mathbb{Z}^{11} \rightarrow \mathbb{R}$ for the form E is defined. For the two remaining forms (representing loudness and duration)

constant morphisms are used. Figure 5 shows the construction of the simple denotator $h1$ with form H . After the form H for the new denotator has been chosen, the address type is set to *non-null* and the previously defined morphism m is selected. Only those morphisms compatible with the form (in this case only those with codomain $\mathbb{Q}$, the module of the form H) are shown in the list. Denotators $e1$ of form E , $l1$ of form L and $de1$ of form ∂E are created similarly.

Figure 6: The Limit denotator construction dialog.

The last step is the definition of the denotator $n1$ of form F (figure 6) and is straightforward from the above discussion.

3.2 The Module Category and its Implementation

The mathematics as implemented in Rubato Composer comprises the rings $\mathbb{Z}$, $\mathbb{Z}_n$, $\mathbb{Q}$, $\mathbb{R}$, $\mathbb{C}$, of integers, modular integers, rationals, reals, and complexes, as well as word monoids $\langle \text{Unicode} \rangle$ over the alphabet Unicode (effectively strings with Unicode characters) and monoid algebras $R\langle \text{Unicode} \rangle$ over a ring R , and in particular the polynomial rings $R[X]$ in one variable. Finally, Cartesian products $R_1 \times \dots R_n$ of rings $R_1, \dots R_n$ are also supported.

Given a finite sequence $M_1, \dots M_k$ of modules over a ring R , the direct sum $M_1 \oplus \dots M_k$ can be built. In particular, the rings as one-dimensional modules over themselves give rise to free modules R^k of dimension k , for example the real vector spaces $\mathbb{R}^k$.

The modules are located in a class hierarchy starting from the Java interface *Module*. Further down the hierarchy, interfaces like *FreeModule* and *Ring* are found that reflect the properties of particular types of modules. Thus, in contrast to the data types of most programming languages (Smalltalk being the most notable exception), a module in the Rubato

framework is itself an object. Some module classes, such as *ZRing* ($\mathbb{Z}$), have only the singleton instance, others, such as *ZnRing* ($\mathbb{Z}_n$), may have many instances. The *Module* interface exports several useful methods, including those for querying attributes, such as whether the module is a ring, or for creating objects such as the zero element in the specific module.

Strictly parallel to the class hierarchy of modules, there is a hierarchy of classes representing elements in modules. These are the values, numbers and vectors, their classes reflecting the corresponding classes in the module hierarchy, for example, *ZElement* or *FreeElement*. The *ModuleElement* interface exports a comprehensive set of operations on module elements, such as addition and scalar multiplication.

A third hierarchy is rooted at the interface *ModuleMorphism* and collects morphisms between modules. Diaffine morphisms feature prominently, as well as special morphisms such as complex conjugation. Morphisms may be added, composed and raised to a natural power, as long as respective domains and codomains satisfy the requirements.

Although the functionality of this class framework is extensive, at least regarding module theory, it is not meant to aspire to a full-featured computer algebra system. Apart from the usual linear algebra (e.g., determinants or Gauss elimination), no algorithms for the solutions of systems are implemented, and polynomials support only the module theoretic operations as well as evaluation. Of course such functionality is useful and may be provided by a rubette that embodies a direct implementation or connects to packages such as Maple, Mathematica, or Maxima.

4 Functoriality and its Application to Musical Operations

The functorial approach is a threefold methodology, providing us with a powerful toolbox for the manipulation of musical objects.

First method: address changes

A given denotator D with data $N_D : A @ F(c)$ and any address change $f : B \rightarrow A$ are used to generate a new denotator D_f with $N_{D_f} : B @ F(c \circ f)$, whose coordinate is the image $c \circ f$ of c under the map $f @ \text{Space}_F$. A typical example is the evaluation of a denotator at elements $a \in A$ of its address. Such an element is addressed by the constant map $f : 0 \rightarrow A : f(0) = a$ from the zero address $B = 0$. For example, we take $A = \mathbb{Z}^n$, $F : \text{Id.Limit}(E, H, L, \partial E)$, the Cartesian product

form composed of the four above Simple forms:

$$\begin{aligned} E &: \text{Id.Simple}(\mathbb{R}) \text{ (onset)}, \\ H &: \text{Id.Simple}(\mathbb{Q}) \text{ (pitch)}, \\ L &: \text{Id.Simple}(\mathbb{Z}) \text{ (loudness)}, \\ \partial E &: \text{Id.Simple}(\mathbb{R}) \text{ (duration)}, \end{aligned}$$

over the $\mathbb{Z}$ -modules $\mathbb{Z}, \mathbb{Q}, \mathbb{R}$. If the address change is one of the canonical basis vectors $e_i = (0, 0, \dots, 0, 1, 0, \dots, 0)$, $i = 1, \dots, n$ or the zero vector $e_0 = 0$, then the $n + 1$ evaluations $c_i = c_{e_i}$ yield the images of the basis vectors in F . This means that we obtain a sequence of $n + 1$ canonical “note” denotators $D_i = D_{e_i}$ associated with the $\mathbb{Z}^n$ -addressed denotator D . Conversely, every such $n + 1$ -tuple $(D_i)_i$ defines a unique $\mathbb{Z}^n$ -addressed denotator D giving rise to the $n + 1$ -tuple, since $\mathbb{Z}^n$ is free. Musically, this means that one may now consider $n + 1$ -tuples of ordinary notes as one “note” living at the free address $\mathbb{Z}^n$. If we take as address change any automorphism f of $\mathbb{Z}^n$ induced by a permutation of the $n + 1$ vectors e_i , $i = 0, \dots, n$, then the new denotator c_f is just a reordering of the given sequence of ordinary, i.e., zero-addressed, notes following the given permutation.

Second method: space transformations

In this case, not the domain address A of D is changed, but its form space. Given a morphism $h : F \rightarrow G$ of forms, the denotator coordinate $c \in A @ \text{Space}_F$ is mapped to a coordinate $hc = A @ h(c)$ of a new denotator ${}_h D$ defined by $N_{{}_h D} : A @ G(hc)$. In our previous example, of course each projection $p_E : F \rightarrow E$, $p_H : F \rightarrow H$, $p_L : F \rightarrow L$, and $p_{\partial E} : F \rightarrow \partial E$ gives rise to the corresponding coordinate denotators $E_D = {}_{p_E} D$, $H_D = {}_{p_H} D$, $L_D = {}_{p_L} D$, and $\partial E_D = {}_{p_{\partial E}} D$. A second typical example is a standard transformation in music on the form F , such as a retrograde inversion $R I$ defined by (1) a pitch inversion $I(x) = h - x$ on $\mathbb{Q}$ and (2) a time inversion $R(x) = t - x$ on $\mathbb{R}$, and leaving the other coordinates fixed. This approach is in fact independent of the given address and works at any address. Therefore the musical application may be thought as an operation on A -addressed denotators as if they were ordinary zero-addressed notes.

Summarizing, every couple of an address change $f : B \rightarrow A$ and a form morphism $h : F \rightarrow G$ gives rise to a new denotator ${}_h D_f$ with address B and form G .

Third method: set-theoretic constructions

On Power type forms, one has the common Boolean operations $C \cup D$, $C \cap D$, and $C - D$ for denotators C, D at the same address. Attention: absolute negation and implication of Boolean logic are not possible within the context of finite power denotators. Moreover, if a denotator D has Power type

form $F : Id.Power(G)$, where G is also a Power type, then the union of the elements of D (which are sets) yields a denotator $\bigcup D$ of form G , the address being unchanged.

Suppose that we are given an A -addressed denotator D in the form F . If we are given a Power-typed denotator S with $N_S : B@H(c)$, where $H : Id.Power(G)$ for a Simple form G with $N_G : Id.Simple(A)$, then we have $c \subset B@A$, i.e., $c = \{c_1 : B \rightarrow A, \dots, c_k : B \rightarrow A\}$ is a set of address changes from A to B . Therefore, we may apply each such change to D and then get the Power-typed denotator D_S with coordinate set $\{D_{c_1}, \dots, D_{c_k}\}$ at address B over the form F . This method is particularly useful in order to generate zero-addressed Power-typed denotators ($B = 0$) from more abstract A -addressed ones. For example, the manipulation of $A = \mathbb{Z}^{11}$ -addressed denotators, which may represent dodecaphonic series, will then yield sets of “ordinary notes” associated with the given series.

5 A Functorial Composition with Dodecaphonic Series

It remains to demonstrate how the theoretical musings of section 4 are put into practice. The basis for the example developed presently are the forms as constructed in the example of section 3.1. However, the names are chosen to be a little longer for the sake of clarity: $Pitch = H$, $Onset = E$, $Loudness = L$, $Duration = \partial E$, and $Note = F$. Assuming that the forms have been created we proceed in several steps. The reader is invited to compare these steps to the procedures described in section 4.

1. Two $\mathbb{Z}^{11}$ -addressed denotators n_1 and n_2 of form $Note$ are created. To this end, a pair of denotators for each of the forms $Pitch$, $Onset$, $Duration$ and $Loudness$ has to be constructed. $Duration$ and $Loudness$ can be dealt with quickly, i.e., by using a meaningful, but for this purpose immaterial, constant morphism.

For the denotators of form $Onset$, a diaffine morphism $h : \mathbb{Z}^{11} \rightarrow \mathbb{R}$ is defined as the composition $h_2 \circ h_1$, where $h_1 : \mathbb{Z}^{11} \rightarrow \mathbb{Z}$, such that $h_1(x) = Ax + b$, where

$$A = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11] \text{ and } b = 0,$$

and $h_2 : \mathbb{Z} \rightarrow \mathbb{R}$ is the embedding of integers in the reals. The morphism for the pitch of the first note n_1 is $f(x) = A_f x + b_f$, where

$$A_f = [1, -3, 2, 3, 5, 7, 4, -1, -2, -4, -5], b_f = 60.$$

The pitch for the second note n_2 is defined by $g(X) = A_g x + b_g$, where

$$A_g = [1, 2, 3, 6, 5, 4, -1, -2, -3, -4, -5], b_g = 67.$$

2. A $\mathbb{Z}^{11}$ -addressed denotator s of form $Score$ (set of $Note$ -denotators in a Power-type form deduced from $Note$) with elements n_1 and n_2 is created. Illustrative screenshots are omitted since this was already shown above.
3. On a new network a *Source* rubette that outputs denotator s is placed (figure 7).

Figure 7: A *Source* rubette.

4. To perform an inversion, a morphism $I : \mathbb{Q} \rightarrow \mathbb{Q}$ is created with $I(x) = 126 - x$. A *ModuleMap* rubette is introduced that performs I on the $Pitch$ part of s (figure 8).

Figure 8: A *ModuleMap* rubette.

5. To perform a retrograde, a morphism $R : \mathbb{R} \rightarrow \mathbb{R}$ is created with $R(x) = 11 - x$. Another *ModuleMap* rubette is placed in the network that performs R on the $Onset$ part of the previous result.
6. The next computation involves the evaluation of the result at an address. The address space is $\mathbb{Z}^{11}$. We choose to evaluate at the basis vectors $e_0, e_1 \dots e_{11}$ using an *AddressEval* rubette (figure 9). The result is a set of $Score$ denotators, i.e., a denotator of the Power-typed form $ScoreSet$.
7. To get a $Score$ denotator again, the previous result is piped into a *Set* rubette to apply the big union operator described in section 4 (third method).
8. To make things a little more interesting, the original denotator s is evaluated at the same address and united with the transformed one through a union.

Figure 9: An *AddressEval* rubette.

9. The result can be displayed in a piano roll representation and played using the *ScorePlay* rubette.

The whole setup is shown in figure 10.

This example is simple, but illustrates the purpose and power of Rubato Composer. Behind the simplicity displayed by the network of rubettes lies a lot of precisely formulated mathematics.

References

- Agon, C., G. Assayag, and J. Besson (2006). *The OM Composer's Book, Vol. One*. Paris: Delatour.
- Buteau, C. (2004). Motivic Spaces of Scores through RUBATO's MeloTopRUBETTE. In *Perspectives in Mathematical Music Theory*, pp. 330–342. epOs music, Osnabrück.
- Buteau, C. (2005). Automatic Motivic Analysis Including Melodic Similarity for Different Contour Cardinalities: Application to Schumann's 'Of Foreign Lands and People'. In *Proceedings of the International Computer Music Conference*. International Computer Music Association.
- Garbers, J. (2004). User Participation in Software Configuration and Integration of OpenMusic, Humdrum and Rubato. In *Perspectives in Mathematical Music Theory*, pp. 262–281. epOs music, Osnabrück.
- Garbers, J. and T. Noll (2004). Harmonic Path Analysis. In *Perspectives in Mathematical Music Theory*, pp. 399–431. epOs music, Osnabrück.

Figure 10: The complete network and the *ScorePlay* view.

- Göller, S. and G. Milmeister (2004). Distributed RUBATO: Foundation and Multimedia Rendering. In *Perspectives in Mathematical Music Theory*, pp. 282–299. epOs music, Osnabrück.
- Mazzola, G. (1991). *SYNTHESIS*. Zurich: STOA music.
- Mazzola, G. (1995). Inverse Performance Theory. In *Proceedings of the International Computer Music Conference*, pp. 533–540. International Computer Music Association.
- Mazzola, G. (2002). *The Topos of Music*. Basel: Birkhäuser.
- Mazzola, G. (2004). Towards a Galois Theory of Concepts. In *Perspectives in Mathematical Music Theory*, pp. 78–88. epOs music, Osnabrück.
- Mazzola, G. and S. Müller (2003). Constraint-based Shaping of Gestural Performance. In *Proceedings of the International Computer Music Conference*. International Computer Music Association.
- Mazzola, G. and J. Stange-Elbe (1995). Cooking a Canon with RUBATO—Performance Aspects of J.S. Bach's *Kunst der Fuge*. In *Proceedings of the International Computer Music Conference*, pp. 179–186. International Computer Music Association.
- Mazzola, G. and O. Zahorka (1994). The RUBATO Performance Workstation on NeXTSTEP. In *Proceedings of the International Computer Music Conference*, pp. 102–108. International Computer Music Association.
- Müller, S. (2004). Parametric Gesture Curves: A Model for Gestural Performance. In *Perspectives in Mathematical Music Theory*, pp. 106–116. epOs music, Osnabrück.
- Volk, A. (2005). Modeling a Processive Perspective on Meter in Music. In *Proceedings of the International Computer Music Conference*. International Computer Music Association.
- Zahorka, O. (1995). PrediBase—Controlling Semantics of Symbolic Structures in Music. In *Proceedings of the International Computer Music Conference*, pp. 203–206. International Computer Music Association.