Composition on Distributed RUBATO by Affine Transformations and Deformations of Musical Structures

Stefan Göller and Gérard Milmeister

MultiMedia Laboratory (MML)
Computer Science Department
University of Zürich
goeller@ifi.unizh.ch, milmei@ifi.unizh.ch

Abstract

A well-known modular software for analysis and performance has been redesigned in Java for distributed components and extended to musical composition. The new compositional component allows for boolean operations, arbitrary affine transformations and deformations on note assemblies, which instantiate a score form representing macro events of arbitrary recursive depth. The enabling framework is implemented on two levels, the data structure level based on the denotator concept with the corresponding operations, and the user interface level, where operations are performed in a 3D realm. These two components are discussed.

1 Introduction

Traditionally the toolbox of a composer contains intellectual devices that allow him to manipulate his musical material and construct musical structures, leading, hopefully, to works of art.

One such device that has been in steady use since the great francoflamic masters from the Renaissance up to the adherents of the Vienna School and contemporary composers is the collection of the various methods of counterpoint.

Even before the 19th century, but especially in the second half of the 20th, stochastic methods have been used to create scores, commonly known as aleatoric music.

These devices can be subsumed under the collective term of musical transformations, in the mathematical sense of a function mapping points from one space to another (or the same) space. In the abovementioned case of counterpoint, a simple appropriate space would span notes, characterized by their onset and pitch. Such transformations include from translations, rotations, similarities and other symmetry operations.

A good example for “composition as computation” is the first movement of Webern’s Symphony op.21. Here every musical event can be traced to a position in the original, retrograde, inversion and the retrograde inversion of the fundamental series.

The performance of such transformations using computers has long been an active topic in both research in music informatics and commercial applications [Mazzola, 1990]. For the most part, however, these efforts concentrated on specific kinds of transformations, and the supporting data structures were specially designed to support them. Thus the resulting utilities were far from attaining a powerful generality. This, among others, precluded the extension of the products to tasks unforeseen at the time of design.

Often these systems were built adhoc without much regard to future developments, and thus most of them sooner or later have become obsolete and work into them was left to software archeologists to salvage.

One composition tool is *presto*® [Mazzola, 1993] developed on the Atari ST from 1988 to 1994. It provided quite a few useful transformations particularly directed at composers. Unfortunately in this case the efforts could not carry over by simple porting to modern platforms.

However the functionality provided by *presto* is too interesting to leave it rotting. Therefore our goal was to implement its features in the setting of Distributed RUBATO® [Göller and Milmeister, 2003], the Java-based redseign of the RUBATO software for NeXT and MacOS X [Garbers et al., 1996–2002]. This allows for a new sophisticated 3D user interface and the reimplementation of the transformations in a much more general way. The key to the achievement of both objectives lies in the use of denotators and forms [Mazzola] for all high-level processing.

2 Denotators and RUBATO

In order to reduce the risk of our project becoming rapidly obsolete, we decided to implement Distributed RUBATO in the Java programming language.

Apart from assuring a high degree of platform independence (development takes place on Linux and Windows), the Java development environment provides a comprehensive set of libraries and functionality required by a project of this extent, for instance Java 3D, network programming, sound and MIDI processing.

The fundamental and pervasive data structure is the denotator, suitable for modelling high-dimensional and nested concept spaces and building on a wide range of mathematical modules such as integers, reals and string-monoid algebras.

These spaces are described by forms, which act as blueprints for the concept spaces in question. The denotators are the substance that is filled into this forms — similar to classes and instances in object-oriented programming, or to XML documents reifying XML schemas.

The notion of denotators is a spin-off from category theory, the language of modern mathematics, and a form is nothing but a directed graph with nodes being either branch nodes (limit, colimit, powerset) or leaf nodes which contain the underlying modules. For a profound introduction see Mazzola.

One such form is of particular importance for the domain we are presently interested in. This *Score* form is a variant of the one defined in Montiel-Hernandez [1999]. In the current shape it is designed to be especially suitable for representing common European scores without limitations to a particular style. The *Score* form provides for data from bibliographical notes, through dynamics to key signatures. Figure 1 has an excerpt of the *Score* form that emphasizes the part relating to notes.[1] It can be seen that a *GeneralNote* encompasses ordinary notes as well as rests; refer to Mazzola et al. [2002] for a complete reference. The circular appearance of the *GeneralNote* form allows for arbitrary complex grouping of note assemblies.

In order to keep the description short, we concentrate on the part pertaining to actual notes which have the form *SimpleNote*: [2]

```
SimpleNote:.limit[Voice, Onset, Pitch,
                  Loudness, Duration,
                  Accidential,
                  Articulation,
                  Fingering];

Voice:.simple(ZString);
Onset:.simple(R);
Pitch:.simple(Q);
Loudness:.simple(RString);
Duration:.simple(R);
```

[1]In this diagram hexagonal shapes picture product types, oval shapes power set types and diamond shapes coproduct types.

[2]For those people not familiar with the DenoteX notation: `F:.limit[`$F_1, \ldots, F_n$`]` defines a form with name `F` that is a *limit*, or *product* of the n forms with names F_1 to F_n. `F:.simple(M)` defines a *simple* form, i.e. a form that is backed with a mathematical module `M`; see Müller [2002] for the EBNF specification.

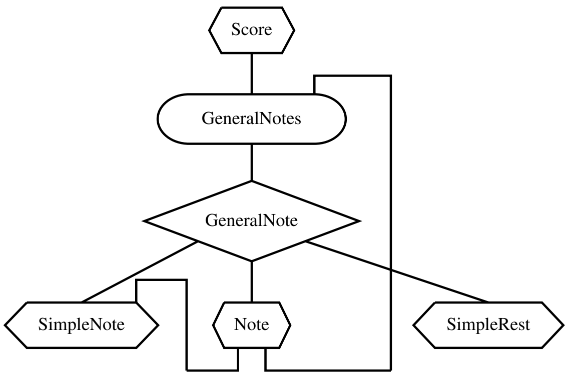

Figure 1: Excerpt from the *Score* form.

Essentially, a note is represented as having a voice, an onset (a real number denoting quarter notes), pitch (a rational number normalized to MIDI key numbers), loudness (a string with a real factor, e.g. “60.0*Midi”) and duration (analogous to onset).

3 Operations on Denotators

We thus presented the structural aspect of denotators. The other aspect deals with operations that can be used to manipulate denotators. The design of these operations is oriented on functional programming methods. In what follows we show a few such useful tools.

On first needs a mechanism that wades through denotators and collects those parts that satisfy a particular condition. This task is assumed by the `Select` class, that does just that: it takes a denotator d and predicate p and returns a list of denotators of denotators g dwelling in d with $p(g) = true$:

```
List select(Predicate p, Denotator d);
```

A second important operator is `Map` which applies a function to all denotators that satisfy a predicate:

```
Denotator map(Denotator d,
              Predicate p,
              Function f);
```

These examples hardly scratch the surface of the functionality provided by these and the other available operation classes, including a wide ranging choice of parameters to influence their behaviour.

3.1 An Example: Topological Deformation

presto incorporates a little tool, called *Ornamagic*, that can be likened to a deformation of musical structures.

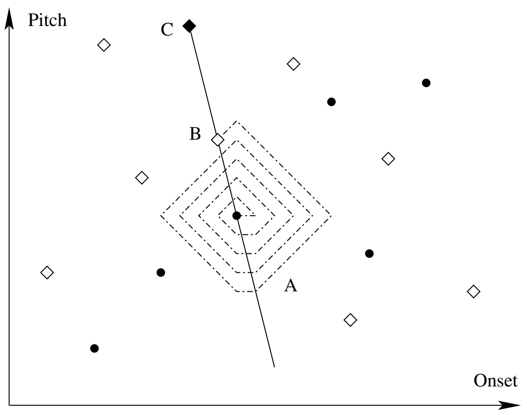

Figure 2: Deformation of a note by *presto*'s Ornamagic

In figure 2 the notes to be deformed are shown as filled circles, the structure that serves as the force field as open diamonds. For each point *A* in the space (in this case the two-dimensional space of onset and pitch), the nearest neighbour *B* in the force field is found. Then a similarity transformation with center *B* is applied, resulting in a new point *C*.

In RUBATO we implemented more general deformations, where Ornamagic emerges immediately as a special case. First we define the force field as a *neighbour* function that returns for each point in the space the nearest neighbour. Second we provide a *deformer* function: Given a point and its neighbour it returns a new point based on an arbitrary transformation on these two points.

The general form of the deformation function looks like this:

```
List deform(Function neighbour,
            List points,
            Function deformer);
```

Ornamagic is implemented as a new function, using the *deform* function:

```
List ornamagic(List neighbours,
               List points,
               Function distance,
               Function deformer);
```

In this case *distance* computes the common Euclidean distance between two points. The one with the smallest distance is taken as a second argument to *deformer*.

The reason we choose a function for modelling the force field instead of, say, a matrix, is that in general a space is not two-dimensional and discrete, as in the example, but may be a module of any dimension over a ring supported by RUBATO. In this case, however, finding the nearest neighbour using exhaustive search may not be feasible. On the other hand, a function returning the nearest neighbour for a point may encapsulate a search algorithm hand-tailored to the domain in question.

4 The 3D Graphical User Interface

While the *presto* software as well as the first version of RUBATO had conventional 2D GUIs, the present Distributed RUBATO has an immersive 3D GUI steered by a 6-degree-of-freedom input device (SpaceMouse^{TM}). The GUI is used to

- visualize all denotators
- manipulate them and
- control all parts and components (Rubettes) of RUBATO.

The most common set-theoretic operations, those already driving much of the power of *presto*, selection, union, intersection and difference, have also a central place in RUBATO and are implemented as interactive operations in the graphical interface, which is shown in the following.

4.1 Selection

Besides the `Select` class described in section 3 there is also a need for selecting parts of a denotator graphically.

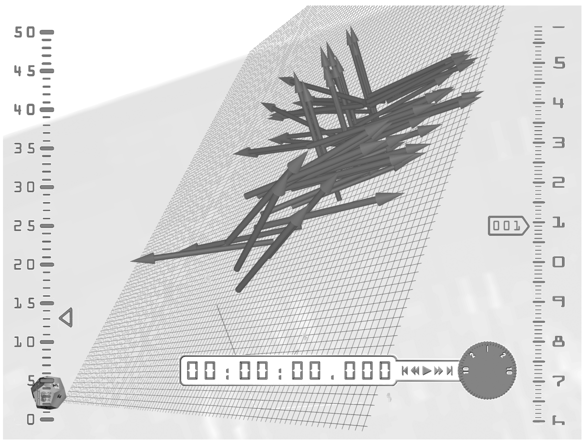

Figure 3: Selection using a plane.

In figure 3 a set of complex notes is visualized by a field of arrows. The user selects a subset of the set in question by moving a fan plane through the space. The intersected arrows are selected.

4.2 Intersection

A basic operation on sets is intersection.

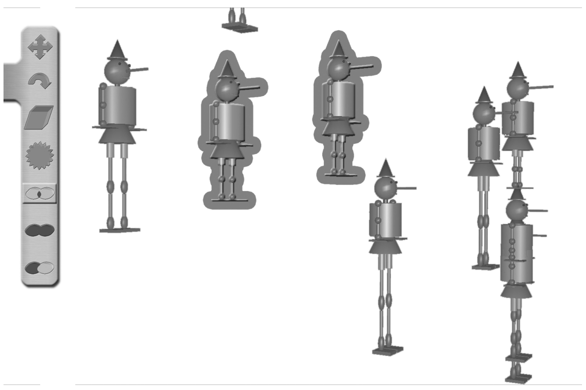

Figure 4: Intersection of two sets.

In figure 4 two sets of data are visualized by Pinocchio puppets. The highlighted Pinocchios represent the intersection of two sets selected by the above procedure. This Boolean operation is chosen from the panel shown on the left side.

4.3 Affine Transformation

Most of the operations used by composers are special affine transformations.

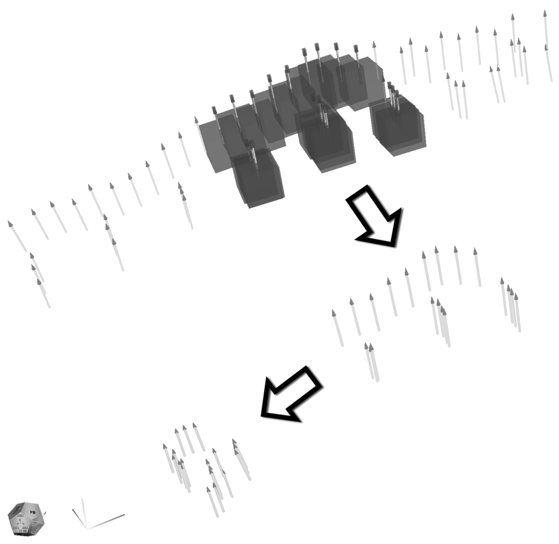

Figure 5: Affine Transformation of a subset.

In figure 5 a short piano piece from Czerny is visualized by arrows in the space of `limit[Onset, Pitch, Duration]`. The user selects a small part and transforms it by rotation.

5 Composition

Using the described interactions, the composer can create musical scores in many ways. He starts either

- by generating denotators from scratch along the *Score* form
- or by loading MIDI or DenoteX files which are automatically cast to denotators of the *Score* form.

Then the composer proceeds by applying the operations from the last section. The visualized musical material can always be played directly either using MIDI output our by remotely steering corresponding output modules like the one described in Mazzola and Müller [2003].

References

Jörg Garbers, Guerino Mazzola, and Oliver Zahorka. RUBATO, 1996–2002. http://www.rubato.org.

Stefan Göller and Gérard Milmeister. Distributed RUBATO: Foundation and multimedialization. In G. Mazzola, T. Noll, and T. Weyde, editors, *Proceedings of the 3rd International Seminar on Mathematical Music Theory*, Osnabrück, 2003. EPOS. To be published.

Guerino Mazzola. *Topos of Music*. Birkhäuser, Basel.

Guerino Mazzola. *Geometrie der Töne*. Birkhäuser, Basel, 1990.

Guerino Mazzola. *presto — Kompositions-Software für den Atari*. Dübendorf, 2nd edition, 1993.

Guerino Mazzola, Gérard Milmeister, and Stefan Müller. Score Form, 2002. http://www.ifi.unizh.ch/mml/musicmedia/downloads.php4.

Guerino Mazzola and Stefan Müller. Constraint-based Shaping of Gestural Performance. *ICMC*, 2003.

Mariana Montiel-Hernandez. El Denotador: Su Estructura, construcción y Papel en la Teoría Matemática de la Musica. Master's thesis, UNAM, Mexico City, 1999.

Stefan Müller. DenoteX EBNF specification, 2002. http://www.ifi.unizh.ch/mml/musicmedia/downloads.php4.