Elody : a Java+MidiShare based Music Composition Environment

Yann Orlarey, Dominique Fober, Stéphane Letz
(orlarey, fober, letz)@rd.grame.fr

Grame, 9 rue du Garet, BP 1185, 69202 Lyon Cedex 01, France

Abstract

This paper introduces Elody, a MidiShare compatible music composition environment developed in Java. The heart of Elody is a visual functional language derived from the $\lambda$-calculus. The languages expressions are handled through visual constructors and Drag & Drop actions allowing the user to play in realtime with the language.

1 Introduction

Elody is music composition environment based on a visual functional language, a direct-handling user interface and Internet facilities. It is written in Java and uses the real-time MIDI services of MidiShare [1].

Elody allows algorithmic descriptions and transformations of musical structures and compositional processes. Its design tries to promote a creative and experimental attitude (including for the programming activity), as well as Internet users collaborations.

Working with Elody mainly consists in building new musical expressions: musical objects as well as programs, by combining or composing other musical expressions. The user interface is based on drag & drop and visual functionalities. Each user action results in an immediate sound and graphical feedback.

Programming using Elody is a natural and direct extension of the music composition activity and doesn’t require a distinct programming language. This approach is called homogeneous programming. An Elody program is a generalized musical expression based on the $\lambda$-calculus concept of abstraction: it allows to use the exact same means to describe, combine, edit or represent musical objects and programs (see [2]).

The Elody environment have been developed using recent Internet technologies in order to facilitate its spread on the Web. The implementation is developed in Java, a programming language similar to C++ which allows to write classical programs and small programs called ”applets” that can be embedded in Web pages. Elody can run either as a stand-alone application or as an applet.

Elody documents are saved in HTML. This allows to:

1. publish musical sequences which can be directly displayed in a Web browser. HTML pages a user wants to share will be available for all Elody users through a central server.
2. musical expressions can be used to add musical content to a page which contains an Elody player applet. This applet will load, evaluate and play the expression contained in the Web page itself.
3. every published expression can be used by other Elody users. An Elody expression can reference others Elody expressions using URL links, which will be automatically fetched and loaded by the language parser.
4. Elody can use musical resources already available on the Web like MIDIFile for example.

2 User interface

Figure 1 gives an overview of the Elody environment. All the functionalities are available by way of visual constructors. A constructor is a particular way to create new expressions by combining existing ones, in sequence or in parallel for example. Visual constructors are represented by one or several argument boxes where the user drop expressions, and a result box where he can get the resulting expression.

The figure 2 shows the use of the sequence constructor (S). The arguments are dropped in the left and middle boxes and the resulting sequence appears in the result box on the right. The constructors perform no real computation. The resulting expression
is a tree whose root node is labeled with the constructor name, and branches are the used sub-expressions. Therefore an expression always keeps track of all its components and of the way to combine them: the intentional description. The evaluation of an expression, the transition from the intentional description to the corresponding extensional description, is only done for graphical or Midi rendering. Users always handle non-evaluated, intentional expressions.

Figure 1: Overview of the user interface

The chord constructor, figure 4, is used to put notes on the pitch scale and to create chords. Concentric circles represent the octaves and radius the degrees within an octave.

Figure 4: The chord constructor

The rules window, figure 5, includes the 8 main constructors: abstraction (λ), recursive abstraction (∞) and application (@), used for programming, sequence (S) and mix (M) for temporal organization of expressions, begin (B) and rest (R) to copy and cut part of an expression and duration (D) to adjust its duration.

Figure 5: The rules window

More sophisticated constructors are also available, like the expressions sequencer figure 6. It works like an old analog sequencers: it includes 8 steps where one can drop musical expressions. Pitch, velocity and duration can be adjusted for each step. The expression sequencer works in real-time: when the Play option is selected the steps are played in a loop, a red dot indicates the current step and the user can dynamically change the settings or drop new expressions.

Figure 2: The sequence constructor

As figure 3 shows, expressions have a piano like representation, with a non-linear (arc-tangent) timescale in order to represent the whole expression in a limited space. Lazy evaluation techniques used in graphical and Midi rendering, allow to handle infinite objects, like the presented sequence C, E, G, repeated ad infinitum.

Figure 3: C E G sequence repeated ad infinitum

The basic musical expressions are notes and rests. They include a duration and a color. A note also includes a pitch, a velocity and a channel.
3 Programming

One of the main purposes of a music composition environment is to allow to express intentional descriptions with all the required generality, and to maintain the relation with the corresponding extensional descriptions.

These two types of descriptions are necessary. Extensional description is needed to render the objects and to work on their details. The intentional description presents many interests:

Conciseness. Compared to its extensional description, an intentional description can be very concise. In this case, it is a fast and efficient way to describe an object.

Structure. An intentional description allows to clarify an object underlying idea, its general structure, its building principles. It is open to analogy, generalization, systematization and reusing in other contexts.

Experimentation. Intentional description modifications have generally a large impact. A small modification of the intentional description can produce a large modification of the extensional description. That promotes experimental approaches like what happens if I use this in place of that within my intentional description.

3.1 Homogeneous languages

Reaching generality and abstraction within intentional description presuppose a kind of programming. The homogeneous language approach we have adopted tries to insert the programming activity in a conceptual framework directly related to the existing skills and knowledge of the user. Instead of providing the user with a separate programming language with specific concepts, syntax, semantic and editing tools, the key idea of the homogeneous language approach is to extend in a consistent way the domain of objects the user manipulates with his editing tools to include user-defined programs. This is achieved mainly by introducing the concepts of abstraction and application of λ-calculus. The results are highly specialized functional languages.

Elody is based on this approach. User-defined programs are abstractions: generalized musical expressions obtained by making variable parts of an existing expression. These abstractions can be applied to other objects to produce a result, and composed to build new programs. Therefore Elody makes no real distinction between musical objects and programs. Programs are just musical objects with variable parts. They are visualized, manipulated and assembled in a consistent way like any other musical objects, without resorting to a separate programming language.

3.2 Building abstractions

Let’s see how to build some simple abstractions. Dropping the same note in the two argument boxes of the sequence constructor (S) as in figure 7.1, we build a repetition. This is a specific example of repetition. We can now generalize this specific repetition by making variable the note used, thus defining the general concept of repetition. To do that we use the abstraction constructor (λ). We drop in the middle box the expression we want to generalize and in the left box the element we want to make variable. The resulting Double abstraction appears in the result box on the right (figure 7.2).

Using the same method we can now define a Triple function which repeats its argument three times (figure 8.1), and apply it on another expression (figure 8.2).

These simple examples illustrate two important points. First, the musical data language itself is used as programming language. Second, we don’t have to write abstractions from scratch, using a priori variables. We can start from a concrete example, even a
sequence played on a keyboard, and showing the parts we want to make variable, let the generalization algorithm of the system compute the actual abstraction.

### 3.3 Composing Abstractions

An abstraction like any other musical expression has a duration. Abstractions can be time-stretched, mixed and organized in sequence. For example we can take the previous Double and Triple functions, time-stretch them to a particular duration and then create a sequence alternating them several times. Figure 9 shows the result of alternating 8 times the Double and Triple functions, a sequence of 16 abstractions.

![Figure 9: Alternating Double and Triple functions](image)

This sequence of abstraction is also a program and it can be applied to some argument. When a sequence is applied, every element is applied according to its duration to the corresponding part of the argument. Therefore if we time-stretch this sequence to the duration of an object and we apply it to this object, the result is the first 1/16 of our object repeated twice, the second 1/16 of our object repeated three times, the third 1/16 of our object repeated twice, etc. as shown in figure 10.

### 3.4 Generalized Abstractions

What we can make variable is not limited to single notes. In the previous example we have used the Triple function. We can generalize the previous result by making variable the Triple function.

![Figure 10: Application of a sequence of abstractions](image)

Figure 10: Application of a sequence of abstractions

The result is a new abstraction we can apply to another function, for example a three voices canon as in figure 11.2. The canon function is obtained by abstracting a note in an expression where this note appears with three different pitches and with different time shift. (figure 11.1).

![Figure 11:](image)

### References
